Al-Specific Code Smells: From Specification to Detection

BRAHIM MAHMOUDI, École de technologie supérieure (ÉTS), Canada NAOUEL MOHA, École de technologie supérieure (ÉTS), Canada QUENTIN STIÉVENART, Université du Québec à Montréal (UQAM), Canada FLORENT AVELLANEDA, Université du Québec à Montréal (UQAM), Canada

The rise of Artificial Intelligence (AI) is reshaping how software systems are developed and maintained. However, AI-based systems give rise to new software issues that existing detection tools often miss. Among these, we focus on AI-specific code smells, recurring patterns in the code that may indicate deeper problems such as unreproducibility, silent failures, or poor model generalization. We introduce *SpecDetect4AI*, a tool-based approach for the specification and detection of these code smells at scale. This approach combines a high-level declarative Domain-Specific Language (DSL) for rule specification with an extensible static analysis tool that interprets and detects these rules for AI-based systems. We specified 22 AI-specific code smells and evaluated *SpecDetect4AI* on 826 AI-based systems (20M lines of code), achieving a precision of 88.66% and a recall of 88.89%, outperforming other existing detection tools. Our results show that *SpecDetect4AI* supports the specification and detection of AI-specific code smells through dedicated rules and can effectively analyze large AI-based systems, demonstrating both efficiency and extensibility (SUS 81.7/100).

CCS Concepts: • Software and its engineering — Software maintenance tools; Automated static analysis; Domain specific languages; • Computing methodologies — Artificial intelligence.

Additional Key Words and Phrases: AI, AI-specific code smells, AI-based systems, Static analysis, DSL, Specification, Detection

1 Introduction

The rapid adoption of Artificial Intelligence (AI) is reshaping the landscape of software engineering, fundamentally altering how applications are designed, developed, and maintained [21, 38, 43]. Broadly defined, an AI-based system encompasses software that integrates AI techniques, including but not limited to machine learning (ML), to perform tasks typically requiring human intelligence, such as decision-making, perception, or adaptation to evolving environments. Particularly, ML, a prevalent subset of AI, enables software to learn from data patterns, moving away from traditional rule-based programming [48]. AI-based systems inherently differ from traditional software due to their probabilistic, data-driven nature, dependence on dynamic data inputs, and tight integration with continuously evolving libraries and frameworks [30]. These characteristics make software reasoning, testing, and maintenance significantly more challenging [43]. While general-purpose linters like Pylint [33], flake8 [32] and the more recent iSMELL [46] effectively identify syntactic or stylistic issues, they overlook subtler, context-specific patterns associated with AI [42, 48]. This shortcoming has led to the introduction of AI-specific code smells, which are defined, following Fowler's notion of code smells [15], as recurrent source code patterns in AI-based systems that, although syntactically correct, indicate deeper potential issues related to correctness, efficiency, or maintainability. Such patterns do not necessarily cause immediate failures, but they often reflect underlying methodological flaws or a misapplication of AI development practices.

For example, the *Data Leakage* smell occurs when preprocessing steps are applied to the entire dataset before it is split into training and test subsets. This causes data information from the test set to leak into the training phase,

Authors' Contact Information: Brahim Mahmoudi, École de technologie supérieure (ÉTS), Montréal, Canada, prenom.nom@etsmtl.ca; Naouel Moha, École de technologie supérieure (ÉTS), Montréal, Canada, prenom.nom@etsmtl.ca; Quentin Stiévenart, Université du Québec à Montréal (UQAM), Montréal, Canada, prenom.nom@uqam.ca; Florent Avellaneda, Université du Québec à Montréal (UQAM), Montréal, Canada, prenom.nom@uqam.ca.

artificially inflating performance metrics and producing overly optimistic results that may not generalize to real-world deployments [42, 48].

Recently, CodeSmile [35] and mlpylint [19] are two tools developed to detect AI-specific code smells. CodeSmile implements 12 of the 22 smells from Zhang et al. [48], while mlpylint implements 20 as Pylint plugins. However, both rely on hardcoded heuristics, which makes them rigid to evolving best practices and difficult to maintain or extend. Mlpylint also suffers from sparse documentation, complicating practical adoption. To the best of our knowledge, these are the only specialized, publicly available tools that target a broad set of AI-specific smells.

To address these identified challenges, namely the need for a flexible, scalable detection tool capable of adapting swiftly to the rapid advancements in AI, we propose *SpecDetect4AI*, a tool-based approach designed for the specification and detection of AI-specific code smells through a Domain-Specific Language (DSL) achieving the best precision and recall trade-off and less than half the analysis time of CodeSmile and mlpylint under identical settings. Unlike existing detection tools, *SpecDetect4AI* adopts an extensible declarative approach based on reusable semantic predicates, empowering practitioners to specify and dynamically extend detection rules in alignment with evolving AI practices. We empirically demonstrate that new smell rules from recent literature [42, 47] can be added with minimal effort using the DSL, as opposed to hand-coded Python rules.

The key contributions of this paper are:

- (1) A Declarative DSL at the core of our tool-based approach We introduce a declarative DSL, expressed with EBNF grammar and first-order AST predicates, that forms the foundation of our tool-based approach SpecDetect4AI for specifying and detecting AI-specific code smells in a modular and extensible way (§3).
- (2) An Implementation and an Empirical Evaluation We implement our approach as the tool *SpecDetect4AI*, specify 22 AI-specific code smell rules, and evaluate the tool on a 826 AI-based systems. This evaluation combines large-scale detection, validation on a mixed ground truth, and comparative analysis with existing tools, as well as an extensibility analysis (§4).
- (3) **A Usability / Extensibility study** We design and conduct a user study (SUS) evaluating the usability and extensibility of *SpecDetect4AI* (§4).
- (4) **A Prevalence study** We assess the prevalence of AI-specific code smells in the evaluation corpus and analyze their distribution to understand underlying causes (§4).
- (5) **An Open replication package** [3] We release the full tool-based approach, rule specifications, ground truth, and evaluation scripts for reproducibility and future research.

The rest of this paper is structured as follows: Section 2 presents an overview of the code smell catalog; Section 3 describes our tool-based approach; the validation and the results of *SpecDetect4AI* are presented in Section 4; and Section 5 discusses related work. We conclude in Section 6.

2 Code Smells Overview

To provide context, Table 1 presents the 22 AI-specific code smells identified across the literature [48], detailing their identifier, name, description, effect, smell type, and the corresponding stage of the ML pipeline. To illustrate these code smells in real-world ML workflows, we also highlight one representative instance per *Effect* category, *Maintainability*, *Correctness*, and *Efficiency*.

R5 Hyperparameter Not Explicitly Set (Listing 1)

Context: Hyperparameters influence the training dynamics and outcomes of ML models, playing a crucial role in

Table 1. Al-specific code smells from Zhang et al. [48]. Abbreviations: Effect (C: Correctness, E: Efficiency, M: Maintainability), Type (G: Generic, A: API-specific), Stage (MT: Model Training, ME: Model Evaluation, DC: Data Cleaning, FE: Feature Engineering).

ID	Code Smell Description	Effect	Type	Stage
R1	Broadcasting Feature Not Used: Manual data duplication instead of using efficient library features, causing memory waste.	Е	Ģ	MT
R2	Randomness Uncontrolled: Not controlling random processes explicitly, causing inconsistent and irreproducible results.	С	G	MT/ME
R3	TensorArray Not Used: Inefficient handling of arrays in iterative operations, causing slow execution.	Е	Α	MT
R4	Improper Train/Eval Mode Toggling: Incorrect switching between training and evaluation modes, causing flawed training procedures.	C	G	MT
R5	Hyperparameter Not Explicitly Set: Relying on default parameters of ML models, causing poor reproducibility and potential performance issues.	M	G	MT
R6	Deterministic Algorithm Option Not Used: Omitting settings that ensure consistent results, making debugging and reproduction difficult.	C	G	MT
R7	Missing Mask of Invalid Value: Not properly handling invalid inputs to certain operations, causing difficult-to-debug errors.	С	G	MT
R8	PyTorch Call Method Misused: Using incorrect methods to pass data through neural networks, potentially causing unexpected behavior.	ı C	A	MT
R9	Gradients Not Cleared: Failing to reset calculation states between training steps, leading to incorrect training results.	С	Α	MT
R10	Memory Not Freed: Failing to free up unused memory resources, leading to system crashes or slowdown.	E	G	MT
R11	Data Leakage: Accidentally mixing testing and training data during preprocessing, leading to misleadingly high performance measures.	e C	G	ME
R12	Matrix Multiplication API Misused: Misuse of mathematical functions, causing confusion or incorrect calculations.	С	Α	DC
R13	Empty Column Misinitialization: Incorrect initialization of empty columns, making future data processing steps error-prone.	. C	G	DC
R14	DataFrame Conversion API Misused: Using outdated or inappropriate data conversion methods, causing unpredictable results.		Α	DC
R15	Merge API Parameter Not Explicitly Set: Unclear or ambiguous data merging, leading to difficult-to-track errors.	M	G	DÇ
R16	In-Place APIs Misused: Incorrect assumptions about whether operations modify data directly, leading to unintended outcomes.		G	DC
R17	Unnecessary Iteration: Using loops instead of more efficient built-in methods, causing slow performance.	E	Ģ	DÇ
R18	NaN Equivalence Comparison Misused: Incorrect handling of comparisons involving missing data, leading to unexpected program behavior.	ı C	G	DC
R19	Threshold-Dependent Validation: Using evaluation methods that depend heavily on subjective thresholds, causing unstable evaluation results.	C	G	ME
R20	Chain Indexing: Inefficient and potentially confusing method of accessing data in <i>DataFrame</i> , causing performance issues and errors.	ı M	Α	DC
R21	Columns and DataType Not Explicitly Set: Not clearly specifying data structure when importing data, causing confusion or subtle bugs downstream.	· M	G	DC
R22	No Scaling Before Sensitive Operation: Failing to scale data properly before analyses, resulting in inaccurate results.	C	G	FE

performance and reproducibility.

Problem: When left to default values, hyperparameters may be suboptimal for the specific task and dataset, and may change across library versions, undermining result consistency and reproducibility.

Inappropriate Behavior: Instantiating ML models without explicitly specifying key hyperparameters (line 8).

Solution: Always set hyperparameters explicitly during model instantiation, and tune them appropriately to the task at hand (line 9).

Listing 1. R5 - Hyperparameter Not Explicitly Set - Maintainability

R11 Data Leakage (Listing 2)

Context: In ML pipelines, data leakage refers to the inclusion of information from the test or validation set during training, often caused by improper preprocessing sequences.

Problem: This leads to artificially high performance during evaluation, resulting in misleading conclusions and poor generalization to unseen data.

Inappropriate Behavior: Applying preprocessing operations (e.g., scaling or encoding) on the entire dataset before performing the train-test split (lines 4-5).

Solution: Perform all data transformations after splitting the dataset, or use pipeline abstractions to encapsulate preprocessing and training steps safely (lines 7-10).

Listing 2. R11 - Data Leakage- Correctness

R1 Broadcasting Feature Not Used (Listing 3)

Context: ML libraries such as PyTorch and TensorFlow offer native broadcasting for efficient tensor operations without manual dimension matching.

Problem: Manually tiling tensors to align shapes leads to unnecessary memory allocations and degrades performance. **Inappropriate Behavior:** Using explicit tiling instead of leveraging built-in broadcasting mechanisms (line 3). **Solution:** Prefer native broadcasting operations to minimize memory usage and computational overhead (line 4).

```
import tensorflow as tf

-tensors = [tf.tile(x, [1, 10]) for x in inputs] # Smell: manual tile in loop
+tensors = inputs *tf.constant([1, 10]) # No smell: native broadcasting
```

Listing 3. R1 - Broadcasting Feature Not Used - Efficiency

3 SpecDetect4AI: A Tool-based approach

SpecDetect4AI is a tool-based approach that enables practitioners to specify informal descriptions of AI-specific code smells, such as those found in the literature, into executable detection rules. Figure 1 presents an overview of the workflow. In our approach, the rules are first specified using a declarative Domain-Specific Language (DSL) (Step 1). They are then automatically compiled and generated by a parser (Step 2), before being processed by a static analyzer that detects AI-specific code smells in AI-based systems (Step 3). We describe the three-step approach using the Hyperparameter Not Explicitly Set code smell as a running example.

3.1 Step 1: Specification

Input: A textual description of an AI-specific code smell.

Output: A declarative DSL rule, manually defined by the rule author, which specifies detection criteria through high-level predicates.

Implementation: We manually specify each smell description as a declarative detection rule in our DSL, where each rule corresponds to a quantified first-order formula composed of semantic predicates. We provide 52 reusable semantic predicates that abstract over low-level AST details to capture higher-level ML-specific intent [3]. For example, isMLMethodCall() identifies calls to ML-relevant methods (e.g., fit(), train()), even when aliased (e.g., clf.fit) or wrapped in helper functions. Other predicates encode structural or temporal relationships, such as

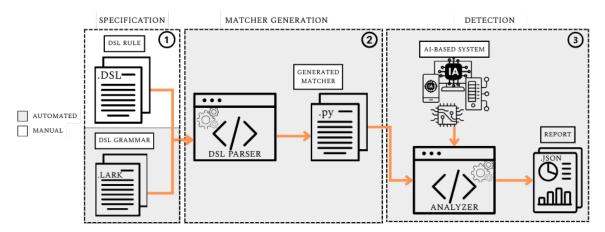


Fig. 1. Overview of SpecDetect4Al's three-step approach for specifying informal Al-specific code smell descriptions into executable detection rules.

usedBeforeTrainTestSplit(), which checks whether preprocessing occurs before dataset partitioning. We define the DSL in Lark's Extended Backus-Naur Form (EBNF) [39, 45]. Listing 4 illustrates a simplified excerpt of the grammar, showing how rules are composed of an identifier, a logical condition (including quantifiers and boolean operators), and an action to execute when the condition is satisfied.

Scope and limitations: SpecDetect4AI operates on Python ASTs at the per-file level. Its semantic predicates (e.g., isMLMethodCall(), hasExplicitHyperparameters()) allow rules to match developer intent rather than surface syntax. They handle aliased calls, helper wrappers, comprehensions, and nested preprocessing defined in the same file without hand-crafting AST templates. This lightweight design keeps rules easy to extend (see Section 4.6.4). By design, we do not perform whole-program interprocedural value-flow across files, nor do we analyze pipelines assembled at runtime. Any behavior requiring cross-class or cross-module flows is out of scope for our static analyzer. We acknowledge that this approach is inherently "soundy" [20, 24], favoring pragmatic expressiveness, predictable performance, and extensibility over complete soundness, a trade-off commonly adopted in practical static analysis [19, 33, 35]. As shown in Section 4.6, this abstraction covers the majority of ML idioms we observe and accounts for much of our recall advantage over rigid templates. As future work, we plan to investigate bounded data-flow and lightweight tracing to further improve analysis accuracy, we outline this direction in Section 6.

```
rule_definition: "rule" IDENTIFIER STRING ":"

"condition:" cond_expr

"action:" action_expr

cond_expr: "exists" IDENTIFIER "in" "AST" ":" "(" cond_expr ")"

predicate_call

| "not" cond_expr
| "not" cond_expr
| cond_expr "or" cond_expr

| cond_expr "or" cond_expr
| "(" cond_expr ")"

predicate_call: IDENTIFIER "(" argument_list? ")"

action_expr: "report" STRING
```

Listing 4. Excerpt of the SpecDetect4AI DSL grammar (EBNF-like)

The grammar comprises only 72 lines and organizes every rule into three core blocks:

- (1) Rule identifier and description (rule) assigns a unique identifier and a short textual name to the smell.
- (2) **Detection condition** (condition) is a first-order logic formula over semantic predicates and quantifiers that defines when the smell occurs.
- (3) Reporting action (action) specifies the message to report upon detecting a smell.

Listing 5 presents a skeleton of a rule template.

```
1 rule RULE_ID "Name of the smell":
2 condition : condition : condition : report <information and location of the smell>
```

Listing 5. Generic template of a rule in the DSL

Each DSL rule captures the semantic conditions under which a smell occurs, abstracting away implementation details, enabling users to focus purely on domain logic.

Running Example: Listing 6 is the DSL rule for the smell R5 Hyperparameter Not Explicitly Set. In this example, the rule explicitly states that a smell occurs when a ML method call (isMLMethodCall) is detected without explicitly specified hyperparameters (hasExplicitHyperparameters). Both identifiers are semantic predicates provided by our predicate library. Users can extend this library with new predicates as explained in Section 4.

```
rule R5 "Hyperparameter Not Explicitly Set":
condition:
sexists call in AST; (
isMIMethodCall(call) and not hasExplicitHyperparameters(call) )
action: report "Hyperparameter not explicitly set at line{lineno}"
```

Listing 6. DSL definition of Hyperparameter Not Explicitly Set

Why a Custom DSL?. We chose to design a dedicated DSL rather than adopting existing general-purpose tools such as CodeQL [17] or Semgrep [34], primarily for four reasons: (1) Domain-specific semantics: existing tools operate on low-level code representations and require users to manually reason about ASTs, including node types, nesting structures, and traversal logic. In contrast, our DSL directly exposes AI-relevant predicates, significantly reducing the cognitive burden for practitioners; (2) Concise readability: our DSL's intuitive, logic-based syntax lets practitioners rapidly define new rules without advanced software-engineering expertise; (3) Semantic expressiveness: while our rules are ultimately expressed over syntactic structures (ASTs), they rely on reusable semantic predicates that encode higher-level AI-specific concepts (e.g., detecting whether a model is trained before a data split, or whether a hyperparameter was explicitly set). These predicates abstract away low-level syntax to reflect domain intent. We acknowledge that this expressiveness is achieved in a best-effort manner: our analysis remains static and does not offer full alias resolution or interprocedural guarantees. Thus, our approach aims for practical precision over theoretical completeness; and (4) Maintainability and extensibility: our modular predicate design facilitates fast rule evolution, essential for accommodating rapidly changing ML libraries. We favor a DSL over a Python-only analyzer [19, 35] because DSL improves (i) audience reach (readable by non-experts, shareable as specifications), (ii) stability and reuse (rules are implementation-agnostic; the analysis backend can be swapped, whether AST-based, data-flow, or even dynamic analysis; and targeting another language only affects the parser), and (iii) maintainability (short, declarative specifications with less repetitive setup code) (see Section 4.6.4).

3.2 Step 2: Matcher Generation

Input: A DSL rule, as defined in Step 1.

Output: A generated matcher file that implements the DSL-defined detection logic.

Implementation: Our custom Lark-based parser reads each DSL rule, compiles it into an AST-traversal function that calls reusable predicates, and writes the result to a matcher file.

Algorithm 1: Matcher File Example for R5 (*Hyperparameter Not Explicitly Set*)

Running Example: Algorithm 1 depicts the generated matcher file for Rule R5 Listing 6. It relies on reusable semantic predicates that encapsulate domain-specific knowledge.

3.3 Step 3: Detection

Input: Generated matcher file from Step 2 and the source code of an AI-based system.

Output: A structured report of the detected code smells and the line number where they occur.

Implementation: We parse each file from the AI-based system into an Abstract Syntax Tree (AST) using Python's built-in ast module. We then load the matcher file generated in Step 2. Each matcher encapsulates the rule's logic via reusable semantic predicates and traverses the AST to detect code smell patterns. We then execute all matchers on each project's AST and aggregate their outputs into a structured report. By parallelizing independent rule matchers on a cached AST, analysis time is bounded by the slowest rule rather than the sum of rules ($T = \max_i T_i$ vs $T = \sum_i T_i$), a design choice that materially improves analysis time.

Running Example: Applying the matcher (e.g., Rule R5 from Listing 6) produces the report illustrated in Listing 7.

```
1 "Path/to/AI-based/system": {
2    "Path/to/python/file/example.py": {
3    "R5": ["Hyperparameter not explicitly set at line 13"],
4    ...
```

Listing 7. Output Report (JSON file)

This output identifies the affected file, the rule identifier (e.g., R5), and a human-readable description of the issue, including the exact line number where the problem occurs. This format is machine-readable and can be integrated into continuous integration (CI) pipelines, automated reporting dashboards, or developer feedback loops. Once the rule is specified in the DSL, the rest of the approach is fully automated: *SpecDetect4AI* generates the matcher file, compiles it, and applies it to the target project without further manual intervention.

4 Validation Design and Results

In this section, we present the study that aims to validate our tool-based approach. We follow a mixed-method methodology through quantitative and qualitative data collection and analysis.

4.1 Research Questions

We aim to respond to the following research questions:

• *RQ*₁: **Does** *SpecDetect4AI* **effectively detect AI-specific code smells?** This question evaluates detection effectiveness in terms of smell instances across files and systems.

- RQ₂: Does SpecDetect4AI achieve better precision and recall compared to existing detection tools? This question assesses whether SpecDetect4AI outperforms existing tools in accurately detecting AI-specific code smells.
- RQ₃: Does SpecDetect4AI scale efficiently across diverse AI-based systems, regardless of system size? This question measures the runtime performance and scalability of the tool across a large set of AI-based systems.
- *RQ*₄: What is the required effort to extend *SpecDetect4AI* with new AI-specific code-smell rules? This question measures the developer effort needed to extend *SpecDetect4AI* with new detection rules.

4.2 Subjects of the Validation

The validation is conducted primarily on our tool, SpecDetect4AI. We also benchmark against two static analysis tools: CodeSmile [35] and mlpylint [19]. Comparability is straightforward because (a) all three tools are Python static analyzers (no training, runtime traces, or dynamic instrumentation), (b) their taxonomies overlap substantially with ours, and (c) each emits per-file, per-rule detection. For CodeSmile, which covers only 12 AI-specific code smells from Zhang et al. [48], we compute results on the 12 code smells intersection using the adjudicated ground truth described Section 4.4. For mlpylint, which overlaps with 20 of our 22 code smells, we compare using the identical evaluation protocol as CodeSmile (20 code smells intersection). In addition, we compare against two families of LLM baselines. First, pretrained open-source models (deepseek-r1 [9], llama-3-70B [1]), as recent literature increasingly investigates LLMs for code quality assessment and defect detection [2, 10, 40]. Second, we include a lightweight hosted model from OpenAI gpt-4.1-mini [29], to represent widely used, production-grade, closed-source LLMs. Our selection rationale is threefold: (i) strong recent performance on code-centric benchmarks [10]; (ii) broad accessibility (either open-source licenses or ubiquitous API availability); and (iii) practical cost/throughput characteristics that enable rigorous, reproducible evaluation under an academic budget. We intentionally did not include larger models because their substantially higher per-token prices and tighter token-per-minute quotas would have constrained fair, end-to-end runs within our evaluation time window. This reflects a growing trend toward leveraging pretrained models as general-purpose analyzers and enables a direct comparison between traditional detection tools and language-model-based approaches using our replication-package prompts [3]. We used a fixed per-rule template to avoid leakage and simplify replication: for each smell, the model received (i) the smell name, (ii) Zhang et al.'s definition [48], (iii) illustrative examples, and (iv) the target snippet to avoid leakage and keep evaluation comparable across all models.

4.3 Objects of the Validation

We apply *SpecDetect4AI* to a large-scale corpus of 826 GitHub AI-based systems, selected using the following multi-step sampling protocol based on prior studies [11, 12, 26], using GitHub metadata to filter and stratify our corpus. First, we retrieved systems tagged AI, ML, or DL (Deep Learning) via a single API query, applying four inclusion criteria: (1) **Recency**: Updated since April 2023 to reflect active maintenance; (2) **Popularity**: At least 100 stars as a proxy

for community interest; (3) **Engagement**: At least 20 forks indicating collaborative development; and (4) **Language**: Primarily Python, matching our tool-based approach.

To complement this metadata-based filtering, we added 87 systems identified in prior literature [11, 13, 43, 48] as emblematic of real-world AI engineering challenges. To facilitate independent verification and follow-up studies, we provide all the metadata in our replication package [3].

4.4 Sampling Design and Ground-Truth Construction

As shown in Table 5, our full evaluation corpus contains 826 systems, 142,747 source files and more than 20 M LOC. Producing a complete, line-level ground truth for all these files would require well over 5,000 human-hours (extrapolated from the annotation rate reported in [27]), clearly infeasible for a single study. We therefore focus the deep-labeling effort on a representative system, mlflow [8], a widely used AI-based system (643 Python files, 120 K LOC, 19.7 K stars). To assess the relative complexity of mlflow within our corpus, we computed a set of structural and architectural metrics across the whole corpus (mean \pm std. dev.) and for mlflow. Mlflow has significantly more functions per file (8.53 vs. 3.42 ± 4.10 , $+1.25 \sigma$) and more classes (0.96 vs. 0.25 ± 0.69 , $+1.02 \sigma$) than the corpus average, with moderately higher LOC and assignment counts. It also exhibits a comparable frequency of explicit pipeline idioms (0.067 vs. 0.083 ± 0.465 , -0.03σ), indicating that its use of pipeline constructs is in line with the corpus. These indicators place mlflow above the corpus mean in several complexity dimensions, supporting its challenging case study of an AI-based system.

To keep the manual workload tractable while preserving statistical validity, we drew a **stratified random sample** [4] of 241 files (47,561 LOC). Strata were keyed on factors known to influence smell density [48]: (i) functional directory, (ii) file-size quartile, and (iii) test vs non-test code. The sample size was calculated with Cochran's formula [7] for proportions (p = 0.5, 95% confidence, ± 5 percentage points (pp)), and post-hoc checks show that smell frequencies deviate by at most 2.3 pp from the population, confirming representativeness despite the reduction.

Because no public ground truth covering the full set of 22 AI-specific code smells currently exists, three Master's-level graduate students (each with 2-3 years of hands-on ML/AI development and prior exposure to the 22 smells) independently labeled the 241 files using a two-pass protocol adapted from prior work [27, 30]. Blind double-labeling followed by consensus yielded Fleiss' $\kappa = 0.85$ (per-rule 0.78–0.89), indicating near-perfect agreement. The annotators were blind to SpecDetect4AI's internal rules and implementation and followed a structured guideline with a controlled form. We reconciled disagreements in a two-step process: (i) pairwise discussion with guideline references; (ii) if unresolved, a third annotator arbitrated. We release the full guidelines, code templates, and raw labels in the replication package to enable auditability and reuse [3]. To ensure comparability with prior work, we also leveraged the CodeSmile dataset [35], which covers only 12 smells on a stratified sample of 100 files. We initially observed definition misalignments and potential label inconsistencies relative to canonical descriptions in the literature [48] (e.g., borderline cases and omission/commission errors). Following content-analysis best practice [22], we ran a structured re-annotation with the same three Master's students using a three-step protocol: (i) independent re-labeling with our written guideline and controlled form; (ii) disagreement discussion and arbitration with citations to the canonical definitions; and (iii) consensus consolidation. In a subsequent joint adjudication session with the first author of CodeSmile [35], we found that several smell definitions admit reasonable interpretations, which plausibly explains part of the divergence across the two annotation efforts. We therefore adopted a consensus strategy that synthesizes labels from both sides: starting from the intersection of agreements, revisiting disagreements with shared instances (file excerpts, rule identifiers, justifications), and documenting binding decisions. To further stress-test borderline instances, we incorporated an LLM-assisted validation (GPT-5 Thinking DeepSearch) using chain-of-thought style reasoning to surface arguments for

and against each label. Crucially, human adjudicators retained final authority, and LLM suggestions were accepted only when consistent with the written guideline and the adjudication rationale. This process produced an adjudicated subset. Importantly, the adjudicated subset is one component of our comprehensive ground truth and is used strictly for intersection-only comparisons with CodeSmile. Our main evaluations rely on the larger ground truth that aggregates (i) the adjudicated CodeSmile subset and (ii) the statistically sampled mlflow corpus, yielding coverage of all 22 code smells.

4.5 Evaluation Metrics

We compare the approach's detections (D) with the ground-truth set (G). Table 2 summarises the four outcomes; V denotes validated positives, V^c validated negatives.

Table 2. Outcome categories (TP: True Positives, FP: False Positives, FN: False Negatives, TN: True Negatives). V: smell instances, V^c : clean instances.

Reported	V (smell)	V ^c (clean)
D (detected) D^c (missed)	$TP = D \cap V$ $FN = D^c \cap V$	$FP = D \cap V^c$ $TN = D^c \cap V^c$

As summarized in Table 2, *precision* is the proportion of validated positives among all detections (|TP|/|D|), *recall* is the proportion of validated positives among all ground-truth instances (|TP|/(|TP| + |FN|)), and F_1 -score is the harmonic mean of the two, $2 \times \text{Prec} \times \text{Rec}/(\text{Prec} + \text{Rec})$.

4.6 Results

In this section, we study and answer the research questions.

4.6.1 RQ₁: Does SpecDetect4AI effectively detect AI-specific code smells at scale? This research question investigates the detection effectiveness of SpecDetect4AI across our validation corpus. SpecDetect4AI detected 86,910 instances of code smells across 25,311 files (see Table 3). Detection volume varies substantially across rules, with the top three (R2 Randomness Uncontrolled, R11 Data Leakage, R6 Deterministic Algorithm Option Not Used) accounting for 68.38% of all detections. This suggests that a small number of poor practices dominate the landscape. On average, we observe 3.45 instances of code smells per file (median: 2, max: 477), suggesting that a few files concentrate a disproportionate number of instances of code smells. These findings emphasize the need to prioritize Correctness-related checks in analyzers. However, covering the full spectrum, including low-frequency but high-impact Efficiency and Maintainability smells, is essential for long-term code robustness, especially as AI-based systems scale or transition to production.

To evaluate how smell detection scales with system size, we compute the Pearson [5] correlation between lines of code (LOC) and instances of code smells count across systems, obtaining r = 0.71 with p < 0.001. This strong positive correlation suggests that larger systems tend to accumulate proportionally more instances of code smells.

However, this distribution is highly skewed. While 54 systems (6.5%) were entirely smell-free, the most affected system contained 5,350 instances of code smells. We quantify this inequality using the Gini coefficient [16], a standard metric from economics that ranges from 0 (perfect equality) to 1 (perfect inequality). Our computed Gini index of 0.78 suggests that code smells are unevenly distributed, with a notable concentration in a subset of systems.

Smell co-occurrence is frequent: 85% of systems contain at least two distinct types of code smells, suggesting interdependent or compounding design issues. Only 7% of systems are completely clean, and 9% are affected by only one code smell. These results confirm that *SpecDetect4AI* effectively detects AI-specific code smells.

Table 3. Al-Specific Code Smells Detected by *SpecDetect4Al*, Grouped by Effect and Sorted by Frequency.

Rule ID	Instances	Instances (%)	Affected Files	Affected Systems	Median/Systems
			Correctness		
R2	32,472	37.39	9,209	654	4
R11	16,619	19.04	4,594	216	5
R6	10,410	11.95	10,410	279	9
R12	5,336	6.12	1,532	262	2
R4	4,664	5.35	2,128	289	3
R9	1,561	1.79	777	118	2
R22	1,082	1.27	340	106	1
R7	336	0.39	185	61	2
R13	66	0.08	46	27	1
R18	53	0.06	25	21	1
R19	17	0.01	17	12	1
			Efficiency		
R17	5,335	6.11	1,639	348	2
R1	1,142	1.31	451	90	2
R10	338	0.39	338	72	2
R3	297	0.34	184	58	2
			Maintainabilit	y	
R5	3,850	4.50	1,491	267	2
R21	1,950	2.28	1,015	217	2
R14	696	0.82	261	118	1
R8	403	0.46	184	64	2
R20	257	0.30	101	66	1
R16	20	0.02	17	15	1
R15	6	0.01	4	4	1

Grouping the 22 rules by their primary *Effect* category (Table 3) reveals a strong skew:

Correctness-related smells represent 83.45% of all detections. This category is led by R2 Randomness Uncontrolled, R11 Data Leakage, and R6 Deterministic Algorithm Option Not Used, which together account for 68.38% of all smell instances. While these code smells do not typically produce immediate runtime failures, they can silently undermine the reproducibility, stability, and scientific credibility of AI experiments [11].

Maintainability-related smells account for just 8.39% of all instances and are more heterogeneous in nature. Examples include R5 Hyperparameters Not Explicitly Set and R14 DataFrame Conversion API Misuse. These smells tend to indicate informal or improvised coding practices, often associated with early-stage prototyping, which may explain their rarity in mature systems.

Efficiency-related smells make up only 8.15% of detections. These often signal redundant computations or unoptimized resource usage, problems that may degrade performance in large-scale or production environments. Their lower frequency may reflect either better engineering practices or the inherent difficulty of detecting performance issues statically.

Type and Pipeline Stage: Following Zhang et al. [48], we also categorize smells by Type (Table 2), either Generic (common across libraries) or API-specific (tied to frameworks like TensorFlow or pandas), and by their typical Stage in the ML pipeline (e.g., training, preprocessing). Our analysis shows that generic smells dominate, with 15 out of 22 rules accounting for 81.72% of all detections. In terms of pipeline stage, model training emerges as the most error-prone phase, responsible for 43.69% of smell instances. This suggests that ML model construction remains a fragile and error-prone activity, deserving focused attention in tooling and review processes.

Implications for Static Analysis and Practice: These findings suggest that analyzers for AI-based systems should prioritize detection of correctness-related smells, particularly those tied to randomness, training logic, and data leakage. However, addressing only high-frequency smells is insufficient: low-frequency but high-impact smells, especially related to

Table 4. Detection performance on the annotated sample. Operation = detection strategy, Coverage = number of supported smells, Positives_in_GT = number of annotated, detectable instances. TP = true positives, FP = false positives, FN = false negatives; P = precision, R = recall, $F_1 = F_1$ -score; Median Time = median analysis time per file (seconds).

Approach	Operation	Coverage	Positives_in_GT	TP	FP	FN	P (%)	R (%)	F ₁ (%)	Median Time (s)
SpecDetect4AI	DSL + predicates + AST traversal	22	387	344	44	43	88.66	88.89	88.77	0.22
deepseek-r1	prompts by rule	22	387	187	151	200	55.33	48.32	51.59	17.54
gpt-4.1-mini	prompts by rule	22	387	163	165	224	49.70	42.12	45.59	28.37
llama3-70b-8192	prompts by rule	22	387	124	177	263	41.20	32.04	36.05	9.63
mlpylint	AST traversal	20	361	132	13	229	91.03	36.57	52.17	0.51
CodeSmile	AST traversal	12	285	216	47	69	82.13	75.79	78.83	0.74

resource management and maintainability, remain critical for robustness, especially as systems scale or move to production.

RQ₁: **SpecDetect4AI effectively detects AI-specific code smells.** Across 826 systems, SpecDetect4AI reports 86,910 code smells with a skew toward correctness-related rules. The top three (R2, R11, R6) account for 68.38% of detections. Multi-code smell co-occurrence is common (85% of systems), while efficiency and maintainability code smells are rarer but remain consequential. These patterns motivate a detector that prioritizes correctness while preserving coverage of low-frequency, high-impact cases.

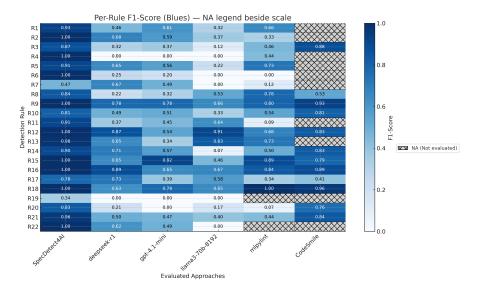


Fig. 2. Per-rule F₁-scores on the intersection of supported smells (darker = higher). "NA" marks rules outside an approach's coverage.

4.6.2 RQ $_2$: Does SpecDetect4AI perform better in terms of precision and recall than existing detection tools

? Table 4 summarizes performance across approaches. All values in Table 4 are computed on the intersection of code smells supported by each approach to ensure fair comparisons (e.g., CodeSmile on the 12 code smells intersection with the adjudicated subset). Also, the reported F_1 -score matches the macro mean over the smells covered per approach.

SpecDetect4AI achieves the best overall trade-off. High precision (88.66%) and the best recall (88.89%) with the best F_1 -score (88.77%), and it finds the most true positives while missing the fewest instances among full-coverage approaches.

Mlpylint attains the highest precision (91.03%) but a low recall due to the high number of instances not detected (229 FNs), which drags down its F_1 -score (52.17%) despite conservative matching (only 13 FPs).

CodeSmile performs strongly within its coverage (12 code smells) with a F_1 -score of 78.83%. CodeSmile remains subsecond (0.74 s per file) even if the analysis time is the worst compared to the two other static analyzers. LLMs trail static analyzers on this benchmark. The best of them (deepseek-r1) achieves a middling F_1 -score (51.59%), while others either under-detect or over-flag (notably llama3-70b-8192 with 177 FPs), which depresses precision (41.20%). *SpecDetect4AI* is the fastest per file (0.22s), substantially ahead of other approaches in this setup. LLMs incur around 9.6s to 28.4s per file, compared to 0.22s to 0.74s for static analyzers (around 13–129 x slower), which substantially limits their scalability in practice. These results come from per-rule parallelism on a cached AST (vs. serial rule passes in CodeSmile/mlpylint) and from avoiding LLM per-inference/API-call latency [23].

Figure 2 shows the F_1 -score per-rule comparison. All per-rule F_1 values are computed on the intersection of smells supported by each approach. Cells marked NA indicate smells outside an approach's scope. Figure 2 reveals a clear pattern, SpecDetect4AI forms the darkest and most uniform column, with high F_1 -score on most rules. Only two rules (R7 $Missing\ the\ Mask\ of\ Invalid\ Value\ (0.47)$ and R19 $Threshold\ Validation\ Metrics\ Count\ (0.34)$) seem difficult to detect with accuracy, both tied to cross-helper, value-dependent semantics that exceed bounded intra-file reasoning.

Within its 12 code smells scope, CodeSmile performs well overall with strong per-rule F_1 -score, though two rules remain weaker (R8 *PyTorch Call Method Misused* (0.53) and R17 *Unnecessary Iteration* (0.41)). Nevertheless, the mean F_1 -score across this 12 code smells intersection is still lower than that of *SpecDetect4AI* (91.58% vs 78.83%).

Mlpylint and LLMs show brittle or dispersed behavior. Mlpylint alternates between high values (e.g., R18 NaN $Equivalence\ Comparison\ Misused$) and zeros when templates miss minor syntactic variation in the codebase (e.g., R6 $Deterministic\ Algorithm\ Option\ Not\ Used$). LLMs exhibit variable, black-box behavior. F_1 -score oscillates across rules. Among LLMs, deepseek-r1 is the most stable. It obtains consistently mid-to-high F_1 -score on rules such as R9 $Gradients\ Not\ Cleared\ before\ Backward\ Propagation\ (0.78)$ or R14 $Dataframe\ Conversion\ API\ Misused\ (0.71)$ but it collapses on value-or flow-sensitive rules (e.g., R19 $Threshold\ Validation\ Metrics\ Count$). Llama3-70b shows few high scores (only R12 $Matrix\ Multiplication\ API\ Misused\ (0.91)$ and R13 $Empty\ Column\ Misinitialization\ (0.83)$) but zeros out on multiple difficult rules (e.g., R7 $Missing\ the\ Mask\ of\ Invalid\ Value$, R22 $No\ Scaling\ Before\ Scale-Sensitive\ Operations$). Finally, gpt-4.1-mini exhibits a pattern comparable to deepseek-r1, achieving similarly strong F_1 on rules with localized instances while encountering similar failures on flow-sensitive code smells.

What LLMs tend to get right? Rules whose instances are local and explicit (clear API calls, in-scope literals, or short patterns with unambiguous surface markers) (e.g., R17 Unnecessary Iteration, R18 NaN Equivalence Comparison Misused or R21 Columns and DataType Not Explicitly Set) are more LLM-friendly. Figure 2 shows that these rules yield coherent mid/high bands across the LLM columns, indicating that pattern recognition from prompt context is often sufficient. Where LLMs struggle? The other code smells are more flow-dependent or value-sensitive. Cases where correctness depends on thresholds, masks, or derived values that propagate through helpers (e.g., R7 Missing the Mask of Invalid Value, R20 Chain Indexing) or where aliasing/wrappers obscure the decisive call sites (e.g., R4 Training / Evaluation Mode Improper Toggling, R6 Deterministic Algorithm Option Not Used). LLMs show unstable scores with unpredictable troughs, reflecting the absence of symbolic name resolution and data-/control-flow reasoning. Attempting to compensate with longer prompts or broader context often inflates FP and the cost of analysis time.

LLMs provide useful coverage on easy to localize code smells but remain fragile on semantics that require even lightweight program analysis. This dispersion contrasts with the uniform, low-variance profile of *SpecDetect4AI* and explains the gap in macro-F₁-score.

Finally, a Wilcoxon signed-rank [44] test on the per-rule F_1 -score confirms that SpecDetect4AI outperforms other approaches (W = 0, p = 0.011). Non-overlapping 95% bootstrap CIs on precision, recall, and F_1 -score corroborate the finding.

 RQ_2 : SpecDetect4AI delivers the best precision—recall trade-off and scalability. It pairs high precision (88.66%) with the highest recall (88.89%), yielding the top macro F1-score (88.77%) and the most TPs (344) with the fewest misses (43) among full-coverage tools. Compared to mlpylint, which has the peak precision but low recall (229 FNs), SpecDetect4AI achieves a far better balance. On CodeSmile's 12 code smells intersection, it still leads (91.58% vs. 78.83%). It is also the fastest (0.22 s/file), roughly 13–129x faster than LLMs in our setup. A Wilcoxon signed-rank test on per-smell F_1 -scores confirms its superiority (p = 0.011).

4.6.3 RQ_3 : Does SpecDetect4AI scale efficiently across diverse AI-based systems, regardless of system size? We evaluated the runtime performance and scalability of SpecDetect4AI by applying it to 826 AI-based GitHub systems (142,747 Python files, 20.5M LOC). The tool detected 86,910 AI-specific instances of code smells across 25,311 files (17.7% of all files). Table 5 summarizes the analysis statistics.

Table 5. General Statistics of the Analysis

Metric	Value
Number of GitHub systems analyzed	826
Size of all systems	29.44 GB
Number of .py files	142,747
Lines of code (LOC)	20,454,021
Files containing at least one code smell	25,311
Total number of code smells detected	86,910
Total analysis time	15,180.6 s (4h 13min)
Average analysis time per system	18.37 s
Median analysis time per system	3.78 s
90 th -percentile time per system	6.20 s
99 th -percentile time per system	14.5 s
Mean time / 1,000 LOC	0.19 s

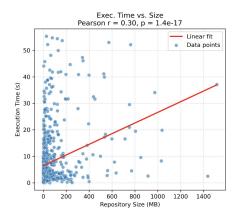


Fig. 3. Execution time vs. system size (values above the 95th percentile capped)

Overall throughput: Across the validation corpus, SpecDetect4AI finished in 4h13min (Table 5), corresponding to 22 k LOC s⁻¹. Hence an average-sized system is analyzed in a few seconds.

Per-system distribution: The timing distribution is strongly right-skewed. Most systems complete in well under ten seconds (median = 3.78s; 90^{th} percentile = 6.2s) while only 1% exceed 15s. The mean cost normalized by size is 0.19s per 1,000 LOC, and this rate stays stable across quartiles, suggesting near-linear scalability.

Impact of system size: Figure 3 plots execution time against system size. The modest Pearson coefficient (r = 0.30, $p < 10^{-16}$) confirms that size alone explains little of the variance, as the analysis of many small systems can still be "slow" if they contain deeply nested ASTs or extensive dynamic imports, whereas several hundred-megabyte systems finish in well under ten seconds. Systems that rely heavily on dynamic imports (importlib.import_module,

__import__) may also slow the tool down, because the parser has to perform conservative extra passes to approximate the resulting graph. Taken together, these observations suggest that AST complexity, not system size, is the main driver of runtime variability, which explains the broad vertical spread of points for our validation corpus (see Figure 3).

 RQ_3 : SpecDetect4AI scales efficiently across systems of widely differing sizes. On 826 public AI-based systems (20.5 M LOC) the median analysis time is only 3.78s per system; 95% finish within 8s and 99% within 14.5s. Runtime grows slowly with system size, the Pearson size-time correlation is modest (r=0.30), implying that factors such as AST depth or import dynamism, not raw byte count, dominate cost. With an average of 0.19s per 1,000 LOC, the tool remains fast even on the largest systems in the corpus, confirming its ability to handle diverse AI-based systems without prohibitive overhead.

4.6.4 RQ₄: What is the required effort to extend SpecDetect4AI with new AI-specific code-smell rules?

To evaluate the extensibility of *SpecDetect4AI*, we recruited five participants: three Master's-level students in AI development and two industry experts (ML/MLOps engineer, software engineer). The three Master's students were the same annotators who built our ground truth and had prior exposure to the 22 code smells under study. The two industry participants had 2 and 4 years of professional experience, respectively, in ML/MLOps and software engineering. All five participants used Python as their primary language for ML workflows. This mixed cohort captures both academic and professional practice, strengthening the validity of our extensibility results. None had prior exposure to our DSL or our tool. All participants followed the structured workflow in our replication package [3]. Under two representative scenarios from recent literature [36], each participant authored the same rule twice: first with **SpecDetect4AI's DSL** and then without the DSL, only Python code (no DSL). A conservative ordering that mitigates order/learning bias. **Scenario A** used only provided predicates (no parser changes). **Scenario B** required introducing one new semantic predicate, extending the DSL parser, then defining the rule with this new abstraction. The starting modality (DSL vs no DSL) was counter-balanced when possible.

Selection of Code Smells: Scenario A: Index Column Not Explicitly Set in DataFrame Read [42]. Omitting explicit indexing when loading a pandas Dataframe is a common source of misalignment in downstream operations such as joins or plots. Prior work [42] highlights this oversight as a recurrent cause of subtle bugs in ML pipelines. Encouraging explicit indexing improves robustness in data processing. Scenario B: Early Stopping Not Used in Model.fit [47]. Omitting Early Stopping in model training can lead to overfitting, degrading generalization performance. This rule exemplifies a semantically meaningful practice that is not easily detected using generic AST-based patterns.

Validation Procedure: We adopt a test-driven development (TDD) protocol: for each rule, unit tests codify curated positive/negative cases. Implementation proceeds until tests pass, with runtime and precision checks to prevent regressions. Artifacts are provided in the replication package [3].

Metrics Collected: For each scenario and each modality (DSL vs. no DSL (a Python API without the DSL)), we record five objective effort metrics: (1) **Specification time (minutes)**: time to author the rule; (2) **Rule specification (lines)**: size of the rule in the chosen authoring surface (DSL file or Python code); (3) **Edit-generate-test cycles**: number of iterations until all tests pass; (4) **Parser modifications (lines)**: lines added to introduce and register a new semantic predicate in the DSL; for the no DSL baseline we analogously report predicate implementation lines when applicable; and (5) **Integration time (minutes)**: end-to-end duration from first edit to validated detection. We complement these per-task ratings with a post-task 10-item **System Usability Scale (SUS)** [18] administered at the tool

Table 6. Extension effort by scenario and modality with the DSL and without DSL (only Python code). Values are mean \pm SD over 5 participants.

	Scenario A	(existing predicates)	Scenario B (new predicate)		
Metric (Average)	DSL	no DSL	DSL	no DSL	
Specification time (minutes)	5.4 ± 1.1	13.0 ± 3.2	9.6 ± 2.2	17.6 ± 3.9	
Rule specification (lines)	5.4 ± 1.3	18.6 ± 4.8	7.6 ± 2.0	28.6 ± 7.2	
Edit-generate-test cycles	2.0 ± 0.5	3.0 ± 0.7	4.4 ± 0.9	5.6 ± 1.2	
Parser modifications (lines)	-	-	13.2 ± 3.3	13.6 ± 3.1	
Integration time (minutes)	8.4 ± 2.0	15.4 ± 3.6	20.8 ± 4.4	29.0 ± 5.6	
Ease of implementation (1-5) [†]	1.2 ± 0.4	3.2 ± 0.7	2.2 ± 0.6	3.4 ± 0.8	
Perceived difficulty (1-5)	1.0 ± 0.0	2.8 ± 0.6	3.2 ± 0.7	3.6 ± 0.7	
Feasibility for non-experts (1–5)	1.8 ± 0.5	2.8 ± 0.7	2.8 ± 0.6	4.2 ± 0.7	
Clarity of DSL syntax (1-5)	1.0 ± 0.0	-	1.0 ± 0.0	-	
Reusability of predicates (1-5)	1.0 ± 0.0	2.0 ± 0.6	2.2 ± 0.5	3.6 ± 0.8	

[†] Lower is better (1 = very easy, 5 = very hard).

level. The SUS questionnaire (*N*=5) yields an overall score of **81.7**/100, indicating excellent perceived usability [6]. All questionnaire items, raw responses, and the scoring script are available in our replication package [3]. Participant's top improvement requests were an interactive predicate explorer and rule debugger (e.g., VS Code extension with syntax highlighting/autocomplete) and clearer, example-driven predicate documentation with more targeted error messages.

Comparative Analysis: Table 6 compares our DSL to a Python-only baseline across two extension scenarios. In Scenario A, the DSL cuts both authoring and end-to-end integration to about half of the baseline, with fewer edit-generate-test cycles and much shorter rule specifications. In Scenario B, the DSL preserves these advantages. Users write far fewer lines (roughly a quarter of the Python-only code) and complete the extension in fewer cycles, while parser edits stay comparable. Subjective ratings track these gaps: participants report easier implementation, clearer semantics, and higher feasibility for non-experts with the DSL. Overall, the results highlight extensibility: we add semantics by implementing a small predicate once and keep the rule declarative and reviewable, whereas the Python-only code spreads logic across larger, less reusable edits.

RQ₄: SpecDetect4AI supports fast, low-overhead rule extension with the DSL. In Scenario A (reusing predicates), the DSL cuts authoring and integration to about half of the Python-only code, with fewer edit-generate-test cycles and much shorter rule specification time. In Scenario B (adding a new predicate), the DSL preserves these advantages. Users write far fewer lines and complete the task in fewer cycles, while parser edits remain comparable. Subjective ratings track these gaps, and a SUS score of 81.7/100 indicates excellent perceived usability.

4.7 Threats to Validity

We organize threats along four classical dimensions.

- i) Internal validity: Our ground truth is a stratified sample of one system (mlflow). We mitigated mono-project bias by stratifying (size, directory, test/non-test) and using blind double-labeling with consensus ($\kappa = 0.85$). To further reduce bias, we leveraged the CodeSmile ground truth (12 smells): due to definition misalignments and label inconsistencies, we re-annotated the CodeSmile ground truth with the same three annotators and then held a joint adjudication with the first author of CodeSmile to reconcile definitions and labels, producing a unified, adjudicated subset.
- ii) External validity: Results may not generalize to all AI-based systems or other languages. We mitigate this by analyzing a heterogeneous corpus of 826 Python AI-based systems stratified by domain and size. To justify using mlflow for deep labeling, we benchmarked its structural complexity against the corpus (Section 4): it exhibits more functions/classes per file ($+1 \sigma$ over the mean) with comparable pipeline-idiom frequency and larger LOC, placing it

above the corpus average and challenging case study of a real-world system. In addition, mlflow is widely adopted by the community, reinforcing its relevance. While our implementation currently targets Python, the DSL is language-agnostic and can be ported to other languages given an AST backend.

iii) Construct validity: Two risks motivate our design and evaluation: potential labeling bias and the intrinsic difficulty of *data-/control-flow* and cross-file smells (e.g., R7, R19). We address the first via blinded, independent annotation with a documented rule-by-rule adjudication (Fleiss' $\kappa = 0.85$). For the second, our tool adopts a deliberate "soundy" approach with intra-file static analysis with conservative, per-file approximations that keep instances observable. This preserves coverage of all 22 smells and favors scalability and maintainability, at the cost of occasional FP/FN on flow-dependent rules, hence slightly lower F_1 -score on those rules compared even if the global F_1 outperforms other approaches.

iv) Conclusion validity. Statistical errors or uncontrolled factors could bias results. To reduce this bias, we used standard metrics and tests (McNemar, Wilcoxon, α =0.05) with Wilson 95% confidence intervals, released all artifacts for replication [3], and documented potential confounders to enable independent verification.

5 Related Work

Our empirical evaluation reveals distinct trade-offs among existing approaches to AI-specific code quality. We now revisit prior work in light of these findings, highlighting how *SpecDetect4AI* addresses observed limitations.

Catalogues and Hardcoded Existing Detectors: Zhang et al. [48] deliver the first catalogue of 22 ML-specific code smells, defining each smell's risks and recommended fixes. CodeSmile [35] is a recent Python static detector that implements only 12 of the 22 smells from Zhang et al. [48], reporting 87.40% precision and 78.60% recall on their own dataset, on the adjudicated ground truth explained Section 4.6.2, the precision and recall drop to 82.13% and 75.79%, respectively. Hamfelt's mlpylint [19], developed as a Master's thesis project only (not peer reviewed), also implements a static detector for 20 of the 22 code smells as Pylint plugins to demonstrate CI compatibility. However, its sparse documentation and reliance on handwritten AST-traversal boilerplate for each rule make it difficult to maintain or extend. Mlpylint achieves high precision (91.03%) but very low recall (36.57%). In contrast, SpecDetect4AI contributes a declarative DSL with reusable semantic predicates, enabling practitioner-driven rule specification and easy extension while covering all 22 smells consolidated in the literature with a precision of 88.66% and a recall of 88.89%. Moreover, we added two novel smell rules (absent from Zhang's catalogue and prior tools) with minimal effort (Section 4.6.4), illustrating both accessibility to non-experts and scalability over time.

Pattern Mining and LLM-Based Analyses: Approaches like MLRefScanner [28] apply machine learning to learn code transformation patterns from refactoring commits, while PyEvolve [11] mines frequent edit patterns from GitHub histories of Python projects to recommend code evolutions. These approaches inspire our use of declarative logic for expressing smell conditions, offering more flexibility and maintainability than rigid pattern templates. Previously, LLM-powered detectors (e.g., deepseek-r1, llama3) have been explored for code smell detection in AI pipelines. While these models improve recall over mlpylint, they suffer from severe precision drops and incur significant latency overheads, with median analysis times more than ten times higher than the worse static analyzer. Our RQ1 and RQ2 results show that SpecDetect4AI's analysis achieves both high recall (87.60%) and precision (88.74%) with 3.78s median per-project runtimes.

API-Aware and Dynamic Analyses: Ariadne [12] applies static analysis to ML code specifically using TensorFlow, by extending the WALA framework [14] with a tensor-aware type system and data flow analysis to detect API misuse. While effective for TensorFlow-specific errors, its scope is limited to API correctness within this single framework, lacking support for broader ML code smells or for user-defined extensibility, which SpecDetect4AI addresses through its

general-purpose. Wei et al.'s LLMAPIDet [42, 43] manually curates API misuse patterns and leverages ChatGPT for patches, demonstrating semantic abstraction benefits yet relying on external LLM calls. DynaPyt [13] uses dynamic instrumentation to catch runtime issues but at the cost of test-execution overhead. In contrast, SpecDetect4AI's DSL-based pipeline combines semantic richness with CI-friendly performance, as RQ1 confirms stable analysis times regardless of project size (r = 0.30). Complementary to these API-centric detectors, Pinconschi et al. [31] propose a pipeline that chains ML models for software-vulnerability detection across languages. SpecDetect4AI adopts a similar "configuration vs. execution" split, but targets AI-specific smells and relies on a declarative DSL rather than YAML orchestration, which simplifies rule extensibility.

Code Generation Error Taxonomies: Wang et al. [41] analyze HumanEval outputs from six LLMs to construct a taxonomy of semantic and syntactic code generation errors. They show that LLMs frequently produce complex, multiline faults correlated with task difficulty. While their work focuses on post-generation repair, our findings underline the complementary need for pre-merge, static detection of smell patterns indicative of latent faults before execution.

Human-Centered and DSL Paradigms: Operational studies highlight developer trade-offs between performance, maintainability, and reproducibility in AI-based systems [37]. Existing detection tools like Semgrep and CodeQL offer extensibility but lack AI-specific abstractions. Foundational DSL research [25] demonstrates that declarative languages reduce boilerplate and improve maintainability. SpecDetect4AI synthesizes these insights, combining DSL expressiveness, AST-level precision, and semantic predicates, to deliver a solution that our evaluation shows to be effective and extensible.

Across descriptive catalogs, hardcoded analyzers, pattern mining, LLM-based approaches, and dynamic instrumentation, no prior method simultaneously achieves the precision, recall, performance, and extensibility validated by our results for the detection of AI-specific code smells.

6 Conclusion

In this work, we proposed *SpecDetect4AI*, an approach based on a declarative DSL for specifying and detecting AI-specific code smells at scale. By specifying 22 high-level smell definitions into an efficient, rule-driven analysis pipeline, *SpecDetect4AI* enables practitioners to express, extend, and apply custom detection rules without altering the underlying framework. Our evaluation on 826 AI-based systems achieved a precision of 88.66% and a recall of 88.89%, uncovering 86,910 smell instances with a median analysis time of just 3.78s per project. These findings demonstrate *SpecDetect4AI*'s efficiency and extensibility, and suggest its potential to aid efforts toward more correct and maintainable AI-based systems. Looking ahead, we will (i) modularize the predicate library as loadable plugins to simplify rule extension, (ii) explore lightweight dynamic analysis (e.g., runtime thresholds) to complement static reasoning, (iii) conduct a prospective remediation study by preparing and submitting patches/PRs that fix detected code smells in Github repositories and observing their reception (review comments, change requests, time-to-merge, acceptance).

References

- [1] Meta AI. 2024. LLaMA 3: Open Foundation and Instruction-Tuned Language Models by Meta. https://github.com/meta-llama/llama3. Accessed: 2025-03-10.
- [2] Eduardo Almeida and Elvys Soares. 2025. Agentic SLMs: Hunting Down Test Smells. arXiv preprint arXiv:2504.07277 (2025). https://arxiv.org/abs/ 2504.07277
- [3] Anonymous. 2025. SpecDetect4AI Replication Package. https://anonymous.4open.science/r/SpecDetect4AI-B903. Study artefacts.
- [4] Sebastian Baltes and Paul Ralph. 2022. Sampling in Software Engineering Research: A Critical Review and Guidelines. *Empirical Software Engineering* 27, 3 (2022), 1–38. doi:10.1007/s10664-021-10072-8

- [5] Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. 2009. Pearson correlation coefficient. Springer. 1–4 pages. doi:10.1007/978-3-642-00296-0 1
- [6] John Brooke. 1996. SUS: a "quick and dirty" usability scale. Usability evaluation in industry 189, 194 (1996), 4-7.
- [7] William G. Cochran. 1977. Sampling Techniques (3 ed.). John Wiley & Sons, New York.
- [8] Databricks. 2024. mlflow: An Open Source Platform for the Machine Learning Lifecycle. https://github.com/mlflow/mlflow. Accessed: 2025-04-01.
- [9] DeepSeek-AI. 2024. DeepSeek-R1: A Strong Open LLM Series with 7B/67B Models. https://github.com/deepseek-ai/DeepSeek-R1. Accessed: 2025-03-21.
- [10] DeepSeek-AI et al. 2025. DeepSeek-RI: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. arXiv preprint arXiv:2501.12948 (2025). https://arxiv.org/abs/2501.12948
- [11] Malinda Dilhara, Danny Dig, and Ameya Ketkar. 2023. PyEvolve: Automating Frequent Code Changes in Python ML Systems. In Proceedings of the 45th International Conference on Software Engineering (Melbourne, Victoria, Australia) (ICSE '23). IEEE Press, 995–1007. doi:10.1109/ICSE48619.2023.00091
- [12] Julian Dolby, Avraham Shinnar, Allison Allain, and Jenna Reinen. 2018. Ariadne: analysis for machine learning programs. In Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (Philadelphia, PA, USA) (MAPL 2018). Association for Computing Machinery, New York, NY, USA, 1–10. doi:10.1145/3211346.3211349
- [13] Aryaz Eghbali and Michael Pradel. 2022. DynaPyt: A Dynamic Analysis Framework for Python. In Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22). 760–771. doi:10.1145/3540250.3549126
- [14] John Field, Manu Sridharan, and Gaurav S. Kc. 2005. WALA: A Scalable Infrastructure for Program Analysis. In Workshop on Middleware for Software Development Environments (at ACM/IFIP/USENIX Middleware). https://wala.github.io/ IBM T.J. Watson Research Center.
- [15] Martin Fowler. 2002. Refactoring: Improving the Design of Existing Code. In Extreme Programming and Agile Methods XP/Agile Universe 2002, Don Wells and Laurie Williams (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 256–256.
- [16] Corrado Gini. 1921. Measurement of inequality of incomes. The economic journal 31, 121 (1921), 124-125.
- [17] GitHub Security Lab. 2024. CodeQL: Semantic Code Analysis Engine. https://codeql.github.com. Accessed May 2025.
- [18] Rebecca A Grier, Aaron Bangor, Philip Kortum, and S Camille Peres. 2013. The system usability scale: Beyond standard usability testing. In Proceedings of the human factors and ergonomics society annual meeting, Vol. 57. SAGE Publications Sage CA: Los Angeles, CA, 187–191.
- [19] Pär Hamfelt. 2023. MLpylint: Automating the Identification of Machine Learning-Specific Code Smells. Dissertation. Blekinge Institute of Technology. https://urn.kb.se/resolve?urn=urn:nbn:se:bth-25490
- [20] Weigang He, Peng Di, Mengli Ming, Chengyu Zhang, Ting Su, Shijie Li, and Yulei Sui. 2024. Finding and Understanding Defects in Static Analyzers by Constructing Automated Oracles. Proc. ACM Softw. Eng. 1, FSE, Article 74 (July 2024), 23 pages. doi:10.1145/3660781
- [21] Heli Järvenpää, Patricia Lago, Justus Bogner, Grace Lewis, Henry Muccini, and Ipek Ozkaya. 2024. A Synthesis of Green Architectural Tactics for ML-Enabled Systems. In Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Society (Lisbon, Portugal) (ICSE-SEIS'24). Association for Computing Machinery, New York, NY, USA, 130–141. doi:10.1145/3639475.3640111
- [22] Klaus Krippendorff. 2013. Content Analysis: An Introduction to Its Methodology (3rd ed.). SAGE Publications.
- [23] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. arXiv preprint arXiv:2309.06180 (2023). arXiv:2309.06180 [cs.LG] doi:10.48550/arXiv.2309.06180
- [24] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In defense of soundiness: a manifesto. Commun. ACM 58, 2 (Jan. 2015), 44–46. doi:10.1145/2644805
- [25] Marjan Mernik, Jan Heering, and Anthony M. Sloane. 2005. When and how to develop domain-specific languages. ACM Comput. Surv. 37, 4 (Dec. 2005), 316–344. doi:10.1145/1118890.1118892
- [26] Nadia Nahar, Haoran Zhang, Grace Lewis, Shurui Zhou, and Christian Kästner. 2023. A meta-summary of challenges in building products with ml components—collecting experiences from 4758+ practitioners. In 2023 IEEE/ACM 2nd International Conference on AI Engineering—Software Engineering for AI (CAIN). IEEE, 171–183.
- [27] Himesh Nandani, Mootez Saad, and Tushar Sharma. 2023. DACOS: A Manually Annotated Dataset of Code Smells. In Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE). 1–12. doi:MSR59073.2023.00067
- [28] Shayan Noei, Heng Li, and Ying Zou. 2018. Detecting Refactoring Commits in Machine Learning Python Projects: A Machine Learning-Based Approach. arXiv preprint arXiv:2404.06572 (2018).
- $[29] \quad Open AI.\ 2025.\ GPT-4.1\ mini\ (Model\ documentation).\ https://platform.openai.com/docs/models/gpt-4.1-mini\ Accessed:\ 2025-09-01.$
- [30] Samir Passi and Steven J. Jackson. 2018. Trust in Data Science: Collaboration, Translation, and Accountability in Corporate Data Science Projects. Proc. ACM Hum.-Comput. Interact. 2, CSCW, Article 136 (Nov. 2018), 28 pages. doi:10.1145/3274405
- [31] Eduard Pinconschi, Sofia Reis, Chi Zhang, Rui Abreu, Hakan Erdogmus, Corina S. Păsăreanu, and Limin Jia. 2023. Tenet: A Flexible Framework for Machine-Learning-based Vulnerability Detection. In Proceedings of the 2023 IEEE/ACM 2nd International Conference on AI Engineering – Software Engineering for AI (CAIN). IEEE. doi:10.1109/CAIN58948.2023.00026 Paper presented at CAIN 2023.
- [32] PyCQA_flake8. 2024. flake8. https://pypi.org/project/flake8/. Accessed May 2025.
- [33] PyCQA_pylint. 2024. Pylint. https://pypi.org/project/pylint/. Accessed May 2025.
- [34] r2c, Inc. 2024. Semgrep: Lightweight Static Analysis for Many Languages. https://semgrep.dev. Accessed May 2025.

[35] Gilberto Recupito, Giammaria Giordano, Filomena Ferrucci, Dario Di Nucci, and Fabio Palomba. 2025. When code smells meet ML: on the lifecycle of ML-specific code smells in ML-enabled systems. Empirical Software Engineering 30, Article 139 (2025). doi:10.1007/s10664-025-10676-4

- [36] Eduardo Royuela and Yania Crespo. 2025. Comparing Different Techniques for Automatic Detection and Correction of React Code Smells. SSRN Electronic Journal (2025), doi:10.2139/ssrn.5251648
- [37] Shreya Shankar, Reyna Garcia, Joseph M. Hellerstein, and Aditya G. Parameswaran. 2022. Operationalizing Machine Learning: An Interview Study. (2022). arXiv:2209.09125 [cs.SE] arXiv preprint.
- [38] Mary Shaw and Liming Zhu. 2022. Can Software Engineering Harness the Benefits of Advanced AI? IEEE Softw. 39, 6 (Nov. 2022), 99-104. doi:10.1109/MS.2022.3203200
- [39] Erez Shinan. 2017. Lark: A Modern Parsing Library for Python. https://github.com/lark-parser/lark. Used for DSL grammar parsing in this work. Accessed April 7, 2025.
- [40] Xiaoxi Wang et al. 2023. Code Llama: Open Foundation Models for Code. In ArXiv Preprint. https://arxiv.org/abs/2308.12950
- [41] Zhijie Wang, Zijie Zhou, Da Song, Yuheng Huang, Shengmai Chen, Lei Ma, and Tianyi Zhang. 2025. Towards Understanding the Characteristics of Code Generation Errors Made by Large Language Models. In Proceedings of the 2025 International Conference on Software Engineering (Track ICSE Research Track) (São Francisco, CA).
- [42] Moshi Wei, Nima Shiri Harzevili, Yuekai Huang, Jinqiu Yang, Junjie Wang, and Song Wang. 2024. Demystifying and Detecting Misuses of Deep Learning APIs. In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (Lisbon, Portugal) (ICSE '24). Association for Computing Machinery, New York, NY, USA, Article 201, 12 pages. doi:10.1145/3597503.3639177
- [43] Moshi Wei, Yuchao Huang, Junjie Wang, Jiho Shin, Nima Shiri Harzevili, and Song Wang. 2022. API recommendation for machine learning libraries: how far are we?. In Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (Singapore, Singapore) (ESEC/FSE 2022). Association for Computing Machinery, New York, NY, USA, 370–381. doi:10.1145/3540250. 3549124
- [44] Frank Wilcoxon. 1945. Individual comparisons by ranking methods. Biometrics Bulletin 1, 6 (1945), 80-83. doi:10.2307/3001968
- [45] Niklaus Wirth. 1977. What Can We Do About the Unnecessary Diversity of Notation for Syntactic Definitions? Commun. ACM 20, 11 (1977), 822–823. doi:10.1145/359863.359909
- [46] Di Wu, Fangwen Mu, Lin Shi, Zhaoqiang Guo, Kui Liu, Weiguang Zhuang, Yuqi Zhong, and Li Zhang. 2024. iSMELL: Assembling LLMs with Expert Toolsets for Code Smell Detection and Refactoring. In 2024 39th IEEE/ACM International Conference on Automated Software Engineering (ASE). 1345–1357.
- [47] Yao Yao, Lei Zhang, and Abhimanyu Chaudhuri. 2008. Early Stopping and Its Applications to Boosting. Journal of Machine Learning Research 9 (2008), 947–970.
- [48] Haiyin Zhang, Luís Cruz, and Arie van Deursen. 2022. Code smells for machine learning applications. In Proceedings of the 1st International Conference on AI Engineering: Software Engineering for AI (Pittsburgh, Pennsylvania) (CAIN '22). Association for Computing Machinery, New York, NY, USA, 217–228. doi:10.1145/3522664.3528620