Empirical Software Engineering manuscript No.
(will be inserted by the editor)

When Code Smells Meet ML: On the Lifecycle of ML-specific Code Smells in ML-enabled Systems

Gilberto Recupito · Giammaria Giordano · Filomena Ferrucci · Dario Di Nucci · Fabio Palomba

Received: date / Accepted: date

Abstract The adoption of Machine Learning (ML)-enabled systems is growing rapidly, introducing novel challenges in maintaining quality and managing technical debt in these complex systems. Among the key quality threats are ML-specific code smells (ML-CSs), suboptimal implementation practices in ML pipelines that can compromise system performance, reliability, and maintainability. Although these smells have been defined in the literature, detailed insights into their characteristics, evolution, and mitigation strategies are still needed to help developers address these quality issues effectively. In this paper, we investigate the emergence and evolution of ML-CSs through a large-scale empirical study focusing on (i) their prevalence in real ML-enabled systems, (ii) how they are introduced and removed, and (iii) their survivability. We analyze over 400,000 commits from 337 ML-enabled projects, leveraging CodeSmile, a novel ML smell detector that we developed to enable our investigation and identify ML-specific code smells. Our results reveal that: (1) CodeSmile can detect ML-CSs with precision and recall rates of 87.4% and 78.6%, respectively; (2) ML-CSs are frequently introduced during file modifications in new

Gilberto Recupito
Software Engineering (SeSa) Lab — University of Salerno, Fisciano, Italy
E-mail: grecupito@unisa.it

Giammaria Giordano
Software Engineering (SeSa) Lab — University of Salerno, Fisciano, Italy
E-mail: giagiordano@unisa.it

Filomena Ferrucci
Software Engineering (SeSa) Lab — University of Salerno, Fisciano, Italy
E-mail: fferrucci@unisa.it

Dario Di Nucci
Software Engineering (SeSa) Lab — University of Salerno, Fisciano, Italy
E-mail: ddinucci@unisa.it

Fabio Palomba
Software Engineering (SeSa) Lab — University of Salerno, Fisciano, Italy
E-mail: fpalomba@unisa.it

feature tasks; (3) smells are typically removed during tasks related to new features, enhancements, or refactoring; and (4) the majority of ML-CSs are resolved within the first 10% of commits. Based on these findings, we provide actionable conclusions and insights to guide future research and quality assurance practices for ML-enabled systems.

Keywords Software Engineering for Artificial Intelligence · Software Quality for Artificial Intelligence · Technical Debt · Empirical Software Engineering.

1 Introduction

Machine Learning (ML) evolved through the emergence of complex software integrating ML modules, defined as ML-enabled systems [18]. Self-driving cars, voice assistance instruments, or conversational agents like ChatGPT[1] are just some examples of the successful integration of ML within software engineering projects. Despite such a rapidly growing evolution, the strict time-to-market and change requests pressure practitioners to roll out immature software to keep pace with competitors, leading to the possible emergence of technical debt [8] *i.e.*, a technical trade-off that can give benefits in a short period, but that can compromise the software health in the long run.

Code smells represent a form of technical debt: these are *symptoms* of poor design and implementation choices applied by developers during evolutionary activities that, if left unaddressed, can deteriorate the overall quality of the system [10]. In the last decades, researchers have been studying code smells in legacy code from multiple perspectives [1,23], identifying them as one of the main precursors of defects and code instability [12,14,15,20]. More recently, Sculley et al. [30] showed that ML-enabled systems are prone to technical debt and code smells, raising the need for a quality assurance process for ML components. Cardozo et al. [4] and Van Oort et al. [36] argued that while the issues in those systems are emerging, there is a lack of quality assurance tools and practices that ML developers can use. Recupito et al. [26] further examined the impact of nine AI Technical Debt (AITD) issues, revealing their significant influence on multiple quality attributes and underscoring the limited support available to practitioners for enhancing overall system quality. This lack of quality management assets stimulates the proliferation of code smells in ML-enabled systems [17]. Consequently, given the complex nature of those systems, new types of code smells have emerged. Considering the aspects that ML developers face when dealing with ML pipelines, Zhang et al. [41] defined a new form of code smells, *ML-specific code smells* (ML-CSs). Similarly to traditional code smells, an ML-CS is defined as a *sub-optimal implementation solution for ML pipelines that may significantly decrease the quality of ML-enabled systems.*

An exemplary case of these quality issues is represented by the use of a loop operation instead of exploiting the corresponding Pandas function for

[1] `https://chat.openai.com/`

data handling, leading to the so-called *'Unnecessary Iteration'* smell, as shown in Listing 1 [41]. In particular, the green line shows how to replace the unnecessary loop to add the value "1" to a data frame.

```
df = pd.DataFrame([1, 2, 3])
- result = []
- for index, row in df.iterrows():
- result.append(row[0] + 1)
- result = pd.DataFrame(result)
+ result = df.add(1)
```

Listing 1: Example of Unecessary Iteration Smell.

Recent research has noted the need for further investigations into the nature of ML-CSs [7,41]. In particular, our research provides empirical insights on how and why these smells emerge and persist in ML-enabled systems. Specifically, the lifecycle of ML-CSs—spanning their introduction, evolution, and removal—-remains poorly understood. For example, whether ML-CSs are predominantly introduced during initial project development or emerge later due to incremental changes and maintenance is unclear. Similarly, the conditions under which these smells are removed and how they survive over time are yet to be thoroughly explored. This lack of knowledge hampers efforts to proactively design effective tools and practices to mitigate the risks associated with ML-CSs. Addressing this gap is critical for ensuring the long-term maintainability of ML-enabled systems. Adopting an evolutionary perspective allows us to analyze software quality in light of Lehman's Laws of Software Evolution [16], which suggests that system degradation is a gradual process that becomes evident over time. This implies that while certain code smells might be acceptable within a software community, exceeding a critical threshold could lead to noticeable quality issues. In this context, a just-in-time tool could be designed with policies that account for such thresholds. For instance, rather than flagging every minor occurrence of a smell, it could provide warnings only when the accumulation of smells reaches a level that risks degrading maintainability or performance. Additionally, our study shows that certain smells appear more frequently during specific development activities. By studying the evolution of smells, we can determine whether developers actively resolve them or if they persist, accumulating as technical debt. Conversely, if a smell naturally disappears due to routine maintenance activities, it may indicate that the issue is less critical. These insights are valuable for prioritization, helping teams focus on addressing smells that have a lasting negative impact rather than treating all detected issues indiscriminately. Additionally, tracking the history of smells allows us to investigate their impact on maintainability and developer behavior. This understanding can inform better refactoring strategies, ensuring that efforts are directed toward smells that historically persist over time and are not resolved through other maintenance activities. Therefore, given the complexity of these smells, developers may struggle to resolve all issues due to various factors, including difficulty in detection or refactoring, limited domain knowledge, or underestimation of potential consequences. This study aims to provide a deeper understanding of the evolutionary characteristics of code smells, forming the basis for more effective prioritization strategies. Furthermore, the contribution made through CodeSmile supports developers by automatically detecting these issues, facilitating a more efficient and informed approach to software quality improvement.

Based on the considerations above, this paper aims to investigate the lifecycle of ML-CSs by conducting a large-scale mixed *confirmatory* and *exploratory* study. We focus on analyzing (i) the prevalence of ML-CSs, (ii) when and why ML-CSs are introduced and removed, and (iii) how ML-CSs survive over time. By leveraging a dataset of over 400k commits across 337 projects coming from the NICHE dataset [39], and employing a novel ML-specific smell detection tool, coined CodeSmile, we aim to provide actionable insights into the factors influencing the emergence and evolution of ML-CSs.

The findings of the study report that ML-CSs are highly prevalent across ML-enabled systems, with specific smells, such as *'Columns and Data Types Not Explicitly Set'* and *'TensorArray Not Used'*, occurring most frequently. Furthermore, the study reveals that ML-CSs are often introduced during project evolution, particularly when developers modify existing files rather than create new ones. This trend highlights ongoing maintenance and feature expansion as key contributors to the introduction of smells. Additionally, the findings show that the survivability of ML-CSs varies significantly based on the type of smell. Certain smells, such as *'In-Place APIs Misused'*, persist for extended periods, potentially due to their subtle impact, which delays their detection and removal. Conversely, more disruptive smells, like *'Gradients Not Cleared'*, tend to be addressed relatively quickly, as they directly affect system functionality during runtime. Finally, the study identifies differences in the prevalence and management of ML-CSs across projects of varying sizes and those adopting Continuous Integration (CI) practices. Larger projects and those without CI tend to exhibit higher rates of ML-CSs, highlighting the importance of proactive quality assurance measures in such contexts.

To sum up, our study provides the following contributions:

1. A large-scale empirical study that provides insights into the prevalence, introduction, removal, and survivability of ML-specific code smells across 337 real-world ML-enabled systems, offering actionable insights into their lifecycle and evolution;
2. A novel static analysis tool, coined CodeSmile, specifically designed to detect 12 ML-specific code smells. This tool advances automated quality assurance for ML-enabled systems and supports further applications both in research and practice.
3. An online replication package [25], which contains all the data collected, scripts, and additional materials used in our study. It enables researchers and practitioners to replicate our findings, build upon our methodology, and further contribute to studying ML-specific code smells.

Structure of the paper. Section 2 provides background information on ML-CSs and overviews the current state of the art, pointing out how we contribute to advancing the body of knowledge. Section 3 presents the methodology of our study, detailing the research questions, dataset preparation, ML-CS detection, and the analysis pipeline. Section 4 reports the key findings of our investigation, while Section 5 provides an in-depth discussion of the take-away messages for both researchers and practitioners. Section 6 discusses potential threats to the validity of our study and the measures taken to mitigate them. Finally, Section 7 concludes the paper and outlines avenues for future work.

2 Background and Related Work

This section presents an overview of ML-specific code smells and summarizes the state-of-the-art research on code smells, highlighting how our work complements and advances the current body of knowledge.

2.1 Background

ML-CSs represent suboptimal implementation practices within ML pipelines that can significantly impact the quality, maintainability, and performance of ML-enabled systems [41].

This new form of code smells has been recently introduced by Zhang et al. [41], who released a catalog of 22 ML-CSs by empirically analyzing white and grey literature. These smells encompass various aspects of ML pipeline development, including data preprocessing, model training, evaluation, and deployment. In the context of our work, **we specifically focus on the subset of ML-CSs detectable using static analysis**, summarized in Table 1. The table presents their descriptions, pipeline stages, effects, and types. These smells can be accurately identified solely by analyzing the source code without requiring runtime information or external dependencies such as datasets or execution logs. On the one hand, the static nature of these smells ensures their accurate detection—as further explained in Section 3.3. On the other hand, it enables a feasible historical analysis of their evolution, as the source code history is readily available in version control systems: this is crucial for our study, as it allows us to systematically investigate the prevalence, introduction, and removal of ML-CSs over time, shedding light on their lifecycle within ML-enabled systems. Additional details and examples of these ML-CSs are available in our appendix [25].

Table 1: List of ML-CSs detectable through static analysis.

Code Smell	Description	Pipeline Stage	Effect	Type
Chain Indexing (CIDX)	This smell refers to when a developer uses to access a single data of a data frame using "[][]".	Data Cleaning	Performance	API-Specific
Columns and DataType Not Explicitly Set (CDE)	This smell refers to when a developer declares a data frame without declaring the column name and the data type.	Data Cleaning	Defect Proneness	Generic
Dataframe Conversion API Misused (DCA)	This smell refers to when a developer uses the function .values() to transform a data frame object to a Numpy array.	Data Cleaning	Defect Proneness	API-Specific
In-Place APIs Misused (IPA)	This smell refers to when the developer assumes the Pandas function returns an in-place value.	Data Cleaning	Defect Proneness	Generic
Gradients Not Cleared Before Backward Propagation (GNC)	This smell refers to when a developer does not use "optimizer.zero_grad()" before " loss_fn.backward()" to clear gradients.	Model Training	Defect Proneness	API-Specific
Matrix Multiplication API Misused (MMA)	This smell refers to when the developer uses the function "np.dot" to multiply a Numpy matrix.	Data Cleaning	Readability	API-Specific
Memory Not Freed (MNF)	This smell regards when a developer declares a machine learning model in a loop operation without using the library ad-hoc function to free the memory at the end of the loop.	Model Training	Memory Issue	Generic
Merge API Parameter Not Explicitly Set (MAP)	This smell refers to when a developer does not specify the options "How" and "On" during a Pandas merge operation.	Data Cleaning	Readability	Generic
NaN Equivalence Comparison Misused (NAN)	This smell refers to when a developer uses the function " np.nan" to compare a data frame value with a NaN value.	Data Cleaning	Defect Proneness	Generic
Pytorch Call Method Misused (PC)	This smell is when a developer forwards the input to the network for the function. "self.net.forward()"	Model Training	Robustness	API-Specific
TensorArray Not Used (TA)	If a developer initializes an array using the function "tf.constant" and assigns a value in a loop operation, it is necessary to use "tf.TensorArray()" avoiding possible errors.	Model Training	Efficiency & Defect Proneness	API-Specific
Unnecessary Iteration (UI)	This smell is regarded when a developer uses a loop operation rather than the corresponding Pandas function.	Data Cleaning	Efficiency	Generic

```
for step, inputs in enumerate(tqdm(eval_dataloader,
    desc="Iteration", disable=args.local_rank not in [-1, 0])):
    for k, v in inputs.items():
        + optimizer.zero_grad()
        inputs[k] = v.to(args.device)
    outputs = model(**inputs, head_mask=head_mask)
    loss, logits, all_attentions = (
            outputs[0],
            outputs[1],
            outputs[-1],
        )
    loss.backward()  # Back propagate to populate the
    gradients in the head mask
```

Listing 2: Example of Gradients Not Cleared before Backward Propagation in the Transformer project.

Unlike traditional code smells, ML-CSs often emerge from the unique characteristics of ML pipelines, such as their reliance on data transformations, iterative model training, and the frequent use of specialized libraries like TensorFlow and PyTorch. These smells can hinder system performance, cause defects, or reduce the interpretability of models. To provide a tangible example of ML-CS, let us consider an instance of the *'Gradients Not Cleared before Backward Propagation'* smell. It refers to when a developer builds a neural network in a loop operation and does not use the function `optimizer.zero_grad()` to clear the old gradients at the end of each iteration. Without this operation, the gradients will gather from all the preceding backward calls. This situation can lead to a gradient explosion, causing a failure in the training process [38]. The function `optimizer.zero_grad()` should be used before the backpropagation step to mitigate this smell. Listing 2 shows an example of *'Gradients Not Cleared before Backward Propagation'* smell for the project TRANSFORMERS.[2] We added an extra line (in green) to indicate how to refactor the smell as denoted in the taxonomy of Zhang et al. [41].

Beyond isolated defects, ML-CSs have broader implications for system performance and maintainability. Smells like *'Unnecessary Iteration'*, where inefficient loops replace vectorized operations, can significantly slow down data processing, especially in large-scale datasets, and can ripple through the entire ML pipeline, introducing bottlenecks that affect training times and deployment schedules. Similarly, smells like *'Columns and Data Types Not Explicitly Set'* in data preprocessing can lead to inconsistencies that cascade into later stages, causing unexpected failures during model inference. Another key implication of ML-CSs lies in their impact on reproducibility and interpretability. For example, issues such as *'In-Place APIs Misused'* can make it unclear whether data transformations have been applied, leading to inconsistencies in experimental results and undermining the credibility of findings. For ML models deployed in high-stakes environments like healthcare or finance, this lack of clarity poses significant risks for practitioners and end-users.

These considerations highlight the need better to understand the nature and evolutionary properties of ML-CSs. Our study addresses this need by systematically investigating the prevalence, introduction, and removal of ML-CSs over time. By shedding light on their properties and behaviors, our findings aim to provide a foundation for designing quality assurance instruments that are not only tailored to the unique challenges of ML pipelines but also specifically adapted to the peculiarities of ML-CSs.

Table 2 shows the 12 ML-specific code smells detectable using CODESMILE. We will discuss the potential impact and possible best practices for removing or avoiding each smell.

Chain Indexing. When using df["one"]["two"], Pandas interprets this as two separate operations: first, it retrieves df["one"], and then it accesses ["two"] based on the result of the first operation. Additionally, assigning values when

[2] Source code available at: `https://github.com/huggingface/transformers/blob/main/examples/research_projects/bertology/run_bertology.py`

Table 2: Code Smells Potential Negative Impact and Best Practices.

Code Smell	Potential Issue	Best Practice
Chain Indexing	Chain indexing in Pandas is slow and unreliable. It performs multiple operations instead of one and may cause assignment failures due to unpredictable views or copies.	Use loc[] for better performance and stability.
Columns and DataType Not Explicitly Set	If columns and data types are not explicitly defined, the downstream data schema becomes unpredictable, potentially leading to silent errors that surface later.	It is recommended that the columns and the type are explicitly indicated during data preprocessing.
Dataframe Conversion API Misused	df.values() has an inconsistency issue, as it may return different array types, making its behavior unpredictable. While noted in the documentation, it is not deprecated and does not trigger warnings or errors.	It is better to use df.to_numpy() rather than df.values().
In-Place APIs Misused	Some methods return a copy instead of modifying data in-place, leading to unintended behavior.	Developers should ensure results are assigned to a variable or the in-place parameter is set.
Gradients Not Cleared Before Backward Propagation	Failing to call optimizer.zero_grad() before loss_fn.backward() causes gradient accumulation, leading to gradient explosion and training failure.	Use optimizer.zero _grad(), loss_fn.backward(), andoptimizer.step() in order, ensuring optimizer.zero_grad() is called first to prevent gradient accumulation.
Matrix Multiplication API Misused	In mathematics, the dot product should return a scalar, but np.dot() returns a matrix for 2D multiplication, deviating from this expectation.	For 2D matrix multiplication, np.matmul() is preferred over np.dot() due to its clearer semantics and correct mathematical behavior.
Memory Not Freed	If the machine runs out of memory during training, the process will fail.	Deep learning libraries provide APIs to mitigate out-of-memory issues. For example, TensorFlow suggests using clear_session() in loops, while PyTorch recommends .detach() to free tensors from the computation graph.
Merge API Parameter Not Explicitly Set	The validate parameter helps ensure expected merge behavior, preventing silent errors from duplicate keys. Since merging is computationally expensive, it should be done efficiently in a single operation.	Developers should explicitly specify merge parameters (e.g., how, on) to avoid ambiguity and silent errors.
NaN Equivalence Comparison Misused	In NumPy, None == None is True, but np.nan == np.nan is False. Since Pandas treats None as np.nan, comparisons with np.nan always return False, which can cause unintended bugs.	Developers should be cautious with NaN comparisons and rely on functions like pd.isna() or np.isnan().
Pytorch Call Method Misused	In PyTorch, self.net() and self.net.forward() are not identical. Calling self.net.forward() directly bypasses hooks registered in self.net().	It is recommended to use self.net() instead of self.net.forward().
TensorArray Not Used	Use tf.TenTensorArray() is used to grow arrays in a loop, avoid inefficiencies, and excessive intermediate tensor creation.	Using tf.TensorArray() is a better solution for growing arrays in a loop, as it avoids inefficiencies and excess.
Unnecessary Iteration	Both Pandas and TensorFlow discourage looping for iteration, as it is inefficient. Pandas recommends avoiding row-wise iteration, while TensorFlow suggests alternatives to slicing loops.	Vectorized solutions are preferable to loops for efficiency and simplicity. Pandas' built-in methods (e.g., join, groupby) and TensorFlow's tf.reduce_sum() outperform manual iteration.

using chained indexing (df[“one”][“two”] = value) can lead to unpredictable results. This is because Pandas does not guarantee whether df[“one”] returns a view or a copy, which can cause assignments to fail unexpectedly and possible performance issues. To avoid this, developers should consider using df.loc[:, (“one”, “two”)] since this function runs as a single call, making it significantly more efficient.

Columns and DataType Not Explicitly Set. This smell refers to when developers import a data frame using the pd.read_csv() function without specifying the columns and the data type for each column. Without this information, it can be difficult for developers to understand the structure of the downstream data schema, causing possible readability issues. To avoid this smell, developers must specify the column names and types using the dtype parameter.

Dataframe Conversion API Misused. When developers use the .value() function provided by the Pandas's official documentation to convert dataframe in a Numpy provided by the Pandas library, It is unclear whether it returns the actual array, a modified version of it, or a Pandas custom array, causing consistency issues and increasing the error-proneness. Despite this, the .values() API has not been deprecated. While the documentation includes a warning about its use, it does not trigger a warning or error during code execution when .values() is used. According to the Pandas' official documentation, to avoid this smell, it is better to use the function pd.to_numpy().

In-Place APIs Misused. This smell affects Pandas and Numpy's libraries and refers to a consistency issue. Specifically, some of the APIs offered by these two libraries consider the variables passed by reference, while others are for value. This inconsistency can increase error proneness if developers do not verify whether the invoked method works with in-place APIs. When possible, it is better to set TRUE as the in-place API parameter value to avoid this and to ensure the return type of each method before it is used.

Gradients Not Cleared Before Backward Propagation. This smell can arise when developers use the PyTorch library to build a deep-learning model. It appears when developers do not use the optimizer.zero_grad() function before loss_fn.backward() in a loop statement. Without this function, the gradients will be accumulated during all the loss_fn.backward() invocations, potentially causing a gradient explosion and increasing the error-proneness of source code. To mitigate this smell, developers must invoke the optimizer.zero_grad() function before the loss_fn.backward() function.

Matrix Multiplication API Misused. In mathematics, the dot product is expected to yield a scalar rather than a vector. However, np.dot() performs different operations depending on the input dimensions, and it may cause confusion. As a result, developers sometimes misuse np.dot() in scenarios where it is not intended, such as for two-dimensional matrix multiplication. For example, a developer expecting a dot product but mistakenly using np.dot() on matrices might not get a scalar. To avoid such issues, developers should carefully choose the appropriate matrix multiplication API, explicitly indicating the intended operation to prevent misunderstandings and avoid code readability issues.

Memory Not Freed. This smell occurs when developers do not properly manage memory in deep learning libraries, leading to run-out-of-memory issues. In TensorFlow, repeatedly creating models within a loop without calling .clear_session() can cause memory buildup. Similarly, in PyTorch, not using .detach() keeps tensors linked to the computation graph, which wastes memory.

To avoid this smell, developers must call the respective clear session method according to the library used (e.g., .clear_session() in the case of TensorFlow or .detach() in the case of PyTorch).

Merge API Parameter Not Explicitly Set. This smell can arise when developers use the function .merge provided by Pandas library. When developers use this function, it is important to indicate the parameters "on" to specify explicitly which columns should be used for joining the datasets, "how" to define the type of join (e.g., outer, inner), and the "validate" parameter to checks whether the merge follows the expected structure. The absence of these parameters can lead to readability and error-proneness issues. To avoid this code smell, developers must explicitly set these parameters.

NaN Equivalence Comparison Misused. In Python, None == None evaluates to True, but in NumPy, np.nan == np.nan evaluates to False. Since Pandas treats None as equivalent to np.nan for simplicity and performance, any comparison involving np.nan in a DataFrame will always return False. If developers are unaware of this behavior, it can lead to unexpected bugs in their code. To mitigate this smell, developers should avoid or minimize the use of this function.

Pytorch Call Method Misused. In PyTorch, self.net() and self.net.forward() are not the same. Calling self.net() also triggers all registered hooks, whereas self.net.forward() only executes the forward pass without considering the hooks. If developers mistakenly use .forward() directly, it can bypass important preprocessing or custom logic implemented in the hooks, potentially affecting the model's behavior and reducing its robustness. To avoid this potential smell, developers must prefer to use self.net() rather than self.net.forward().

TensorArray Not Used. If a developer initializes an array using tf.constant() and attempts to update it in a loop to grow it, the code will result in an error. This issue can be resolved using the low-level tf.while_loop() API, but this approach is inefficient. It creates numerous intermediate tensors during the process, leading to unnecessary memory usage and reduced performance. To avoid this issue, use tf.TensorArray() is a better solution for growing arrays within a loop.

Unnecessary Iteration. As reported by the Pandas's official documentation, "*iterating through pandas objects is generally slow.*" For this reason, it should be avoided, when possible, due to the possible efficiency issues. Developers should adopt a vectorized solution instead of iterating over data. Using operations optimized for batch processing, such as tf.TensorArray(), is a more efficient approach for handling growing arrays in loops. This avoids excessive memory usage and improves performance compared to iterating and appending values manually.

2.2 Related Work

Several studies have investigated code smells in traditional systems [15,20,21,32], exploring their impact on software quality and evolution [13,37]. Among these, Tufano et al. [34] conducted a large-scale empirical study to analyze when and why code smells are introduced, their survivability, and how developers address them. Their findings revealed that most code smells are introduced during the creation of files, while only a negligible proportion are removed through dedicated refactoring efforts. Inspired by the work of Tufano et al., our research aims to extend this understanding into the domain of ML-specific code smells. By doing so, we seek to enhance the quality assurance processes for ML-enabled systems, which present unique challenges distinct from those in traditional software systems.

In the remainder of this section, we focus on state-of-the-art research related to traditional code smells in ML-enabled systems, *i.e.*, studies that investigated how the code smells originally defined by Fowler [9] manifest in ML-enabled systems. To our knowledge, no study has explicitly focused on ML-CSs within the context of ML-enabled systems.

Tang et al. [33] conducted an empirical analysis of 26 ML projects, showing that traditional code smells are highly diffused, with *'Duplicated Code'* emerging as the most prevalent. Similarly, Van Oort et al. [36] analyzed 74 open-source ML projects using PyLint and confirmed that *'Duplicated Code'* is the most frequent smell. The authors also noted that code smells occur more frequently in ML systems than traditional software. Building on this work, Giordano et al. [11] conducted a longitudinal study of code smell diffusion over time in ML-enabled systems, focusing on the activities leading to their introduction and survivability. The findings suggested that the smell variation does not follow a specific pattern over time; their introductions are mainly due to evolutionary activities, and code smells can persist for several years without remediation. More recently, Cardozo et al. [4] extended these findings by examining 29 reinforcement learning projects. Their results echoed previous studies, confirming that code smells emerge more frequently in reinforcement learning projects than traditional systems, underscoring the distinct challenges ML-enabled systems pose.

While these studies provide insights into traditional code smells in ML-enabled systems, our research focuses specifically on ML-CSs, which are inherently tied to the unique characteristics of ML pipelines. More particularly, by investigating the prevalence, introduction, removal, and survivability of ML-CSs, our study aims to shed light on how these smells arise and persist.

Contribution to the State of the Art.

Our work advances the state of the art by shifting the focus from traditional code smells to ML-specific code smells, which are uniquely tied to the characteristics of ML pipelines. By systematically studying the prevalence, introduction, removal, and survivability of ML-CSs, we provide insights that address a significant gap in the literature. These findings contribute to the body of knowledge by informing the design of customized quality assurance tools and practices, enabling developers and researchers to better manage the quality of ML-enabled systems.

3 Research Method

Our empirical study aims to explore to what extent ML-CSs are prevalent in ML-enabled systems, when and how they are introduced and removed, and for how long they survive. More specifically, let us formulate the specific goal of the study through the application of the GQM approach [2].

◎ **Our Goal.**
Purpose: Explore
Issue: (i) the prevalence, (ii) the introduction, (iii) the removal, and (iv) the survival
Object: of ML-specific code smells in ML-enabled systems
Viewpoint: from the points of view of ML developers.

Figure 1 depicts the process we followed to address our research goal by addressing a preliminary and three main research questions.

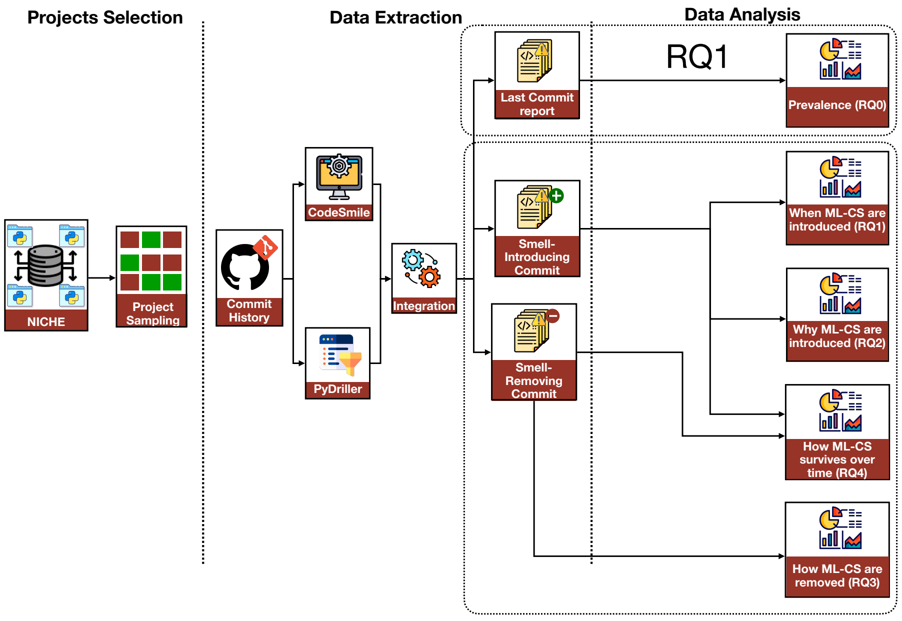


Fig. 1: The process designed for the study.

RQ$_0$. *How are ML-specific code smells prevalent in ML-enabled systems?*

The reason behind this preliminary investigation is twofold. On the one hand, we may assess the relevance of the problem: should we identify a poor prevalence of ML-CSs, this may indicate that the problem is not as relevant as in traditional systems [20,3], possibly not motivating further research on the matter. On the other hand, we may identify the most common ML-specific code smells and the type of ML projects in which they manifest themselves.

RQ$_1$. *When are ML-specific code smells introduced in ML-enabled systems?*

The first research question aims to classify the conditions and contexts under which developers introduce ML-CSs. Additionally, it seeks to determine whether ML-CSs are injected during the initial creation of ML projects or emerge throughout the system's evolution. The results to **RQ$_1$** would inform when ML-CSs should be mitigated.

RQ$_2$. *What tasks were performed when the ML-CSs were introduced?*

After understanding when ML-CSs are injected, it is necessary to extract information about the reasons that led developers to update the system by introducing an ML-CS. So, **RQ$_2$** aims to extract developers' actions that likely introduce ML-CSs.

RQ$_3$. *When and how ML-specific code smells are removed in ML-enabled systems?*

RQ$_3$ focuses on the timing and methods employed to remove ML-CS. This research question is motivated by the need to understand the strategies and circumstances under which developers address ML-CSs. By identifying the set of strategies that developers use to remove ML-CSs, we aim to extract insights to define automatic refactoring strategies for ML-CSs that developers would be inclined to integrate into the development processes.

RQ$_4$. *How long do ML-specific code smells survive in the code?*

Finally, **RQ$_4$** aims to observe the survival time of each ML-CS to identify the ones that persist in the project over time. The outcome of the analysis can be utilized to focus on detecting the ML-CSs that exhibit greater endurance during software maintenance.

In terms of reporting, we followed the guidelines by Wohlin et al. [40] and the ACM/SIGSOFT Empirical Standards[3]. As for the latter, we used the *'General Standard'*, *'Data Science'*, and *'Repository Mining'* guidelines.

3.1 Dataset Description and Projects Selection

We relied on the NICHE dataset [39] for our investigations. This dataset contains 572 ML-enabled systems and was released at MSR '23. We selected it for two reasons. On the one hand, it contains only popular, active ML projects

[3] Available at `https://github.com/acmsigsoft/EmpiricalStandards`

Table 3: Descriptive statistics of the NICHE projects.

	Stars	Commits	LOC
Min	100	100	10
1st Q.	211	219	3,829
Median	529	420	9,235
Mean	1,978	1,307	24,414
3rd Q.	1,641	1,070	21,845
Max	76,838	90,927	699,513

with extensive commit histories *i.e.*, projects with over 100 stars on GitHub, with commits more recent than May 1st, 2020, and with at least 100 commits, allowing us not to select personal or inactive projects. On the other hand, it contains heterogeneous projects with different characteristics.

To verify the feasibility of the analysis on the selected dataset, we preliminary mined it.

This operation was necessary because it is reasonable to suppose that some projects could be no longer available for some reason (*e.g.*, repositories are archived, some communities migrated to other version control systems, or some repositories have restricted access). At the end of this step, we identified 566 projects available out of the 572 and 1,110,689 commits. Table 3 shows the descriptive statistics on the variables "Stars", "Commits", and "Lines of Code" provided in NICHE [39]. As we can notice from the metrics extracted, the project distribution in the NICHE dataset presents a median of about 529 stars, 420 commits, and 9,235 lines of code, suggesting that the projects have a high development activity.

Observing the statistics of the projects, we noticed a high variability between projects in terms of lines of code (LOC). According to Zhou et al. [42], the project size is an impactful confounding variable when analyzing code-related aspects. Therefore, we analyzed the active projects in the dataset through a percentile distribution analysis and divided them into three groups:

Small: Projects with a number of lines of code below the 30th percentile. This set consists of 173 projects, all containing less than 4,765 lines of code.

Medium: Projects with a number of lines of code above the 30th percentile and below the 60th percentile. This set consists of 169 projects, all containing less than 11,836 lines of code.

Large: Projects with a number of lines of code above the 60th percentile. This set consists of 224 projects, all containing more than 11,836 lines of code.

Due to the potential computational issues arising from the large number of projects and commits, we applied statistically significant sampling for each group. We used an online sample size calculator to estimate the required number of projects needed to draw statistically valid conclusions.[4] Specifically, we selected the projects considering each population, a sample with a 95% confidence level and a 5% margin of error. As a result, we considered 117

[4] Qualtrics sample size: `https://www.qualtrics.com/blog/calculating-sample-size/`

Table 4: Descriptive statistics of the active projects divided by size.

		Stars	Commits	LOC
Small	Min	100	100	1,648
	1st Q.	171	148	2,301
	Median	367	244	2,921
	Mean	1,170	552	3,115
	3rd Q.	879	436	3,871
	Max	13,265	13,542	4,763
Medium	Min	100	105	4,783
	1st Q.	159	241	6,268
	Median	290	375	7,817
	Mean	1,138	610	8,039
	3rd Q.	907	722	9,344
	Max	18,087	3,299	11,835
Large	Min	105	103	12,005
	1st Q.	336	405	17,656
	Median	901	855	27,352
	Mean	3,052	2,271	54,342
	3rd Q.	2,652	1,514	46,488
	Max	33,741	47,094	661,808

projects, composed of 64,607 commits, for Small projects, 118 projects, consisting of 71,380 commits, for Medium projects, and 142 projects, composed of 265,671 commits, for the Large projects *i.e.*, we analyzed 337 projects and 401,658 commits. Table 4 shows the descriptive statistics for each size group. In addition to analyzing projects based on their size in terms of LOC, we also considered the most related characteristics that led the authors to define a project as ML-engineered. One such characteristic is the adoption of Continuous Integration (CI). The presence of a CI pipeline may directly affect the quality assurance mechanisms implemented by the projects, possibly affecting the presence of ML-CSs. This distinction between projects with and without CI allows us to explore potential differences in the prevalence and management of ML-CSs between the two groups. Therefore, to incorporate this aspect into our analysis, we thoroughly examined the dataset and found that 319 projects utilize CI tools, whereas 247 projects do not incorporate CI into their development environment.

3.2 Data Extraction

After cloning the projects, we gathered fine-grained structural metrics using PyDriller, a framework helpful to analyze Git repositories [31]. We extracted the commit history of a project P belonging to the selected projects. For each commit, $C_i \in P$, we collected the total number of files, the number of removed and added files, the commit date, and the commit message.

3.3 Building an ML-Specific Code Smell Detection Tool

To address our research questions, we required detecting ML-specific code smells. However, to the best of our knowledge, no existing tools are available, necessitating the development of a custom detection tool that could be used for our purposes. As such, we developed CodeSmile, an ML-CS detection tool that leverages Abstract Syntax Tree (AST) analysis to automate the identification of the 12 statically detectable ML-specific code smells introduced in Section 2. The tool implements rule-based criteria derived directly from the definitions provided by Zhang et al. [41].

The detection process begins by analyzing the libraries imported in the source code. This step distinguishes general-purpose libraries (e.g., `numpy`, `pandas`) from machine learning frameworks (e.g., `tensorflow`, `torch`). These libraries form the basis for identifying ML-CSs, as their APIs and constructs underpin the rule-based criteria implemented in CodeSmile.

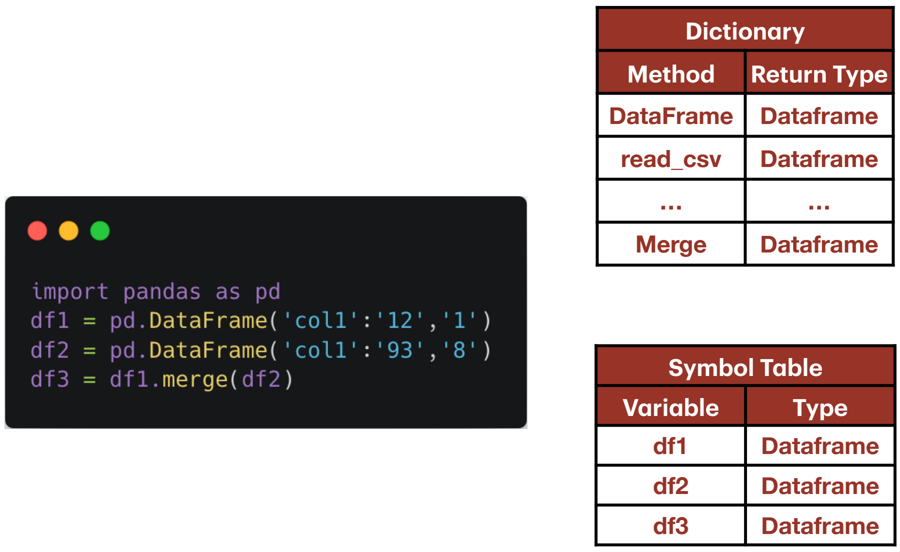
Next, CodeSmile collects relevant keywords from the documentation of the identified libraries. These keywords include method names, attributes, and parameters critical to defining and detecting ML-CSs. For example:

– For Pandas, operations such as `set_index`, `drop`, `merge`, and `fillna` are gathered.
– For TensorFlow and PyTorch, APIs related to the definition of ML models are collected.

To build this dictionary, we referred to the smell descriptions and examples provided by Zhang et al. [41]. We identified all libraries involved in the code smells and manually annotated each method with its corresponding return type in a CSV file. This labeling was done based on the official documentation of each library. In cases where the return type was unclear, we created and executed ad-hoc scripts to determine the return value using the built-in function “type()”.

Figure 2 illustrates how CodeSmile performs static type inference. When analyzing source code, CodeSmile inspects each variable assignment and checks whether the method invoked has a known return type in the internal dictionary. If so, it extracts the variable name and checks the symbol table. If the variable already exists in the symbol table, its type is updated with the newly inferred one. If the variable is not yet declared, a new entry is added to the symbol table with the inferred type. This systematic collection of relevant keywords ensures that CodeSmile may incorporate library-specific behaviors into its detection rules, enhancing its accuracy and robustness. Afterwards, the detection tool applies three main steps, as described below.

Variable Extraction and Library Association. A critical workflow component is determining whether the objects involved in potential smells are associated with specific libraries, such as Pandas or TensorFlow. This process involves tracking variable definitions and operations throughout the code to provide accurate contextual information for smell detection.



```
import pandas as pd
df1 = pd.DataFrame('col1':'12','1')
df2 = pd.DataFrame('col1':'93','8')
df3 = df1.merge(df2)
```

Dictionary	
Method	Return Type
DataFrame	Dataframe
read_csv	Dataframe
...	...
Merge	Dataframe

Symbol Table	
Variable	Type
df1	Dataframe
df2	Dataframe
df3	Dataframe

Fig. 2: Example of How CodeSmile Performs Type Inference.

1. **Variable Initialization Tracking**: CODESMILE identifies variable definitions during AST traversal by analyzing assignment nodes. For each variable, the tool inspects its initialization to determine if it corresponds to a specific object associated with a library. For example, variables initialized using `pd.DataFrame()` or `read_csv()` are identified as Pandas DataFrames, while those created with `tf.Tensor()` or `torch.tensor()` are recognized as objects belonging to TensorFlow or PyTorch, respectively. This classification ensures that CODESMILE accurately associates variables with their corresponding libraries and enables precise smell detection.
2. **Cross-Referencing Library Usage**: The tool cross-references the detected variable with the libraries imported in the source code, which ensures the identified operations are relevant to the associated library and not generic constructs. Therefore, library-related variable usage is tracked throughout the code to handle transformations and reassignments. For example, if `df` (a Pandas DataFrame) is reassigned as `df_copy = df`, this association is preserved, and the subsequent transformations are monitored to collect all the related variables in the context.
3. **Contextual Validation**: CODESMILE confirms that operations flagged as potential smells involve library-specific objects. For instance, it is excluded if a method matches a Pandas API but is applied to a non-DataFrame object to minimize false positives.

AST-Based Detection After establishing foundational information, the tool parses the source code into an AST representation. Using a recursive traversal of AST nodes, the tool inspects constructs such as function calls, attribute accesses, and indexing operations, enabling a detailed analysis of API usage against predefined detection rules.

For example, consider the case of the *'In-Place APIs Misused'* smell, which occurs when a DataFrame operation is performed without explicitly setting the `inplace=True` parameter or assigning the result to a new variable. These practices can lead to ambiguity, as the original DataFrame remains unmodified, discarding the operation's effect. For the sake of clarity, Listing 3 illustrates this smell in a Pandas DataFrame operation.

```
[...]
fields = [data_manager.get_field_name(idx) for idx in]
pd_df = pd.DataFrame(flatten_obj , columns=fields ,[...] )
- pd_df.set_index(data_manager.schema.sample_id_name)
+ pd_df = pd_df.set_index(data_manager.schema.sample_id_name)
[...]
```

Listing 3: Example of In-Place API Misused in the FederatedAI/FATE project.

A running example of execution of the detection of this smell is:

1. The tool detects the initialization of `pd_df` as a Pandas DataFrame through the `pd.DataFrame()` constructor.
2. The prefix `pd` is mapped to the Pandas library based on the import statement (`import pandas as pd`).
3. The method `set_index()` is flagged as a potential in-place API due to the absence of `inplace=True` and lack of result assignment.
4. By verifying that `pd_df` is a Pandas DataFrame, the tool confirms the context and flags the smell.

Rule-Based Detection for 12 ML-CSs CodeSmile implements 12 rule-based conditions, each targeting a specific ML-CS, as shown in Table 5. These rules capture both structural and semantic characteristics of smells:

- **Structural Analysis**: Identifies patterns such as API misuse based on method names.
- **Semantic Context**: Ensures that smells are only detected when the operations involve the appropriate objects, such as DataFrames or tensors.

By combining variable extraction, library association, and AST-based traversal, CodeSmile ensures precise detection of ML-CSs, minimizing false positives due to API-Call of other libraries and enhancing detection accuracy.

3.4 Validation of the ML-Specific Code Smell Detection Tool

After developing our detection tool, we validated its accuracy through a user study involving ML engineers. This approach was chosen to ensure the validation process was grounded in practical, domain-specific expertise, reflecting

Table 5: Rule-Based Conditions for Detecting Code Smells

Name of the Smell	Specific Detection Rule
Chain Indexing (CIDX)	Detect nested `ast.Subscript` nodes where indexing operations are chained (e.g., `df['col']['subcol']`).
Columns and DataType Not Explicitly Set (CDE)	Check for `DataFrame` or `read_csv` calls where the `keywords` attribute is missing or does not contain the `dtype` argument.
Dataframe Conversion API Misused (DCA)	Identify the method calls where the method name is `values()` and the object is a Pandas `DataFrame`.
In-Place APIs Misused (IPA)	Detect `DataFrame` related method calls where the `inplace=True` parameter is not set or the result is not assigned to a variable.
Gradients Not Cleared Before Backward Propagation (GNC)	Search for calls in training cycles to `loss_fn.backward()` without a preceding call to `optimizer.zero_grad()`.
Matrix Multiplication API Misused (MMA)	Detect calls to `np.dot` for matrix multiplication instead of using the `np.matmul`.
Memory Not Freed (MNF)	Check loops where ML models are instantiated without a subsequent call to `clear_session` or an equivalent memory-freeing method.
Merge API Parameter Not Explicitly Set (MAP)	Identify `merge` calls in Pandas where the `keywords` attribute is missing or does not include `how`, `on`, and `validate` arguments.
NaN Equivalence Comparison Misused (NAN)	Detect comparing nodes that involve `np.nan` as one of the two comparator of the condition.
Pytorch Call Method Misused (PC)	Identify direct calls to the method `forward()`.
TensorArray Not Used (TA)	Detect loops where `tf.constant` is used for tensor array assignment.
Unnecessary Iteration (UI)	Identify `for` loops over rows in `DataFrames` (e.g., using `iterrows`) instead of vectorized operations like `groupby` or `apply`.

real-world development practices. To achieve this, we first selected a statistically significant sample from the 3,439 ML-related files identified as containing at least one ML-CS. Using this subset, we designed an experimental kit to guide participants through the analysis and assessment of the tool's detection capabilities. The kit comprised three primary components:

– **Definitions and Instructions for Each ML-CS:** Participants were provided with detailed definitions and instructions for each ML-CS, as outlined by Zhang et al. [41]. For each smell to be analyzed, the participants received information on its definition, the potential problems it could cause, possible refactoring solutions, and practical examples illustrating its occurrence.
– **Validation Register:** Participants used a structured sheet to report their findings. For each file and ML-CS type, participants indicated whether the file was affected (answering "Yes" or "No") and recorded the line number(s) where the smell occurred.
– **Task-Specific Instructions:** A detailed supplementary document was provided to ensure participants clearly understood the task objectives and the steps required to complete the evaluation. This document included an

Table 6: Performance Metrics of CodeSmile to detect each ML-CSs

Smell Name	TP	FP	FN	Precision	Recall
Chain Indexing (CIDX)	29	2	4	0.935	0.879
Columns and Datatype Not Explicitly Set (CDE)	55	1	1	0.982	0.982
Tensor Array Not Used (TA)	33	0	8	1.000	0.805
Dataframe Conversion API Misused (DCA)	19	0	21	1.000	0.475
Gradients Not Cleared Before Backward Propagation (GNC)	12	1	1	0.923	0.923
NaN Equivalence Comparison Misused (NaN)	2	0	0	1.000	1.000
In-Place APIs Misused (IPA)	14	5	8	0.737	0.636
Matrix Multiplication API Misused (MMA)	1	0	0	1.000	1.000
Memory Not Freed (MNF)	17	1	5	0.944	0.773
Merge API Parameter Not Explicitly Set (MAP)	14	0	2	1.000	0.875
PyTorch Call Method Misused (PC)	10	20	0	0.333	1.000
Unnecessary Iteration (UI)	2	0	7	1.000	0.222
Total	209	30	57	0.874	0.786

overview of the experimental materials, instructions on how to interpret the detection results from the tool, and guidance on how to validate or refute the identified ML-specific code smells.

Given the large number of files containing potential smells, we conducted a pilot study to assess the feasibility of the evaluation process. Two Ph.D. students with three years of experience in ML were recruited to inspect a subset of 12 files affected by ML-CSs. This pilot study served two purposes. First, it allowed us to refine the clarity of the task and improve the reporting mechanism in the validation register. Second, we gathered data on the time required for participants to inspect specific smells, providing insights into the ideal workload size to prevent excessive task duration.

Participants in the pilot study took approximately 25 minutes to complete the inspection. While certain ML-CSs, such as *'Columns and Datatype Not Explicitly Set'* were quick to identify, others, such as *'Memory Not Freed'* required up to 8 minutes per detection. Based on the results, we determined that the full set of 346 files affected by ML-CSs would be too extensive for practical evaluation and reduced the sample size to 100 files. To ensure a robust validation, we employed a team of ten ML engineers. Each participant has a minimum of three years of experience in ML development. We stratified the sample to include files representing all types of ML-CSs covered in this study. Each file was inspected by two evaluators to enhance reliability. In cases where only one evaluator identified an instance of a smell, the two discussed their findings to reach a consensus on whether the smell should be flagged.

This stratified approach allowed participants to focus on a selected subset of smells, ensuring greater precision while reducing the required human effort.

Results of the evaluation are summarized in Table 6. The distribution of true positives (TP), false positives (FP), and false negatives (FN) across the analyzed ML-CS types reveals significant variability in their prevalence and detectability. For instance, smells such as *'Columns and Datatype Not Explicitly Set'* and *'Tensor Array Not Used'* exhibit high true positive rates with minimal false positives and negatives, suggesting that these smells are common and relatively straightforward to detect, aligning with their inherent clarity in definition and behavior. On the other hand, more complex smells, such as *'Dataframe Conversion API Misused'* and *'Memory Not Freed'*, pose greater challenges, as evidenced by higher FN counts, indicating their complex nature during detection.

The precision and recall metrics highlight the efficacy of CodeSmile in detecting different ML-CSs. Most smells, including *'Tensor Array Not Used'* and *'Merge API Parameter Not Explicitly Set'* reaches a maximum precision on a higher samples of evaluations. Similarly, *'Columns and Datatype Not Explicitly Set'*, *'Memory Not Freed'*, *'Change Indexing'*, and *'Gradients Not Cleared Before Backward Propagation'* detection exceed the 90% of precision, highlighting therefore excellent detection from our static analysis tool. Other smells, including *'Unnecessary Iteration'* and *'Merge API Parameter Not Explicitly Set'* are evaluated on very small sample of instances of smells but considering the examples found on real projects, these are evaluated by the participants as true positives. However, the recall for some smells, such as *'Unnecessary Iteration'* (22.2%) and *'Dataframe Conversion API Misused'* (47.5%), indicates room for improvement in reducing false negatives. These results suggest that while CodeSmile performs well in identifying instances of these smells, some occurrences are overlooked, likely due to the contextual nature of their detection.

The overall precision (87.4%) and recall (78.6%) achieved by CodeSmile indicate that the tool provides a robust foundation for identifying ML-CSs. These metrics significantly increase our confidence in the conclusions drawn from the analyzed set of code smells to address our research questions. Even in cases where recall is relatively low, the tool demonstrates exceptionally high precision, reaching up to 100%. This ensures that the vast majority of detected instances are correct, enabling reliable detection outcomes.

In the context of our mining study, high precision is particularly critical, as it ensures the analysis of a reliable sample of smell instances. Although lower recall means that not all true positives are included in the sample, the availability of a highly accurate subset of smells still allows us to draw meaningful and trustworthy conclusions. In other words, the high precision of the tool minimizes the impact of false positives on the analysis, enabling robust insights into the characteristics and behaviors of ML-specific code smells. Additionally, these results represent a valuable side contribution of our work. CodeSmile has potential usefulness for practitioners to detect ML-CSs in real-world scenarios and can serve as a foundation for researchers seeking to enhance its functionality by building upon the rule-based approach proposed in this study.

3.5 Commit Data Extraction

After identifying the method for detecting ML-CS, we extracted the commit data for each project to address our research questions. First, we identified the smell-introducing and smell-removing commits for each identified ML-CS instance. Specifically, we tracked each smell s_i identified in a commit c_i, using its file name and line number. We analyzed the project's history from the first commit, comparing c_i and $c_i + 1$ pairwise. For each pair of consecutive commits, we considered the two following cases:

1. If $c_i + 1$ contains a smell s_i not contained in c_i, then $c_i + 1$ is the smell-introducing commit for s_i.
2. If c_i contains a smell s_i not contained in $c_i + 1$, then $c_i + 1$ is the smell-removing commit for s_i.

After collecting smell-introducing and smell-removing commits, we analyzed the commit messages to understand the rationale behind introducing and removing ML-CSs.

3.6 Data Analysis

The following section explains how we analyzed the collected data to respond to our research questions.

RQ$_0$: *How are ML-specific code smells prevalent in ML-enabled systems?* CodeSmile analyzed the last snapshot of the selected ML-enabled systems to observe the prevalence distribution of each ML-CS. Statistical descriptions and plots were employed to understand the characteristics of each distribution. Then, insights on the most prevalent ML-CS across several projects were provided. Such results were further enriched by mapping each identified ML-CS to the related ML-pipeline stage, utilizing the mapping framework established by Zhang et al. [41]. This additional mapping step enhanced our understanding of ML-CSs, revealing the most prone areas within the ML development pipeline. Each analysis was conducted considering the effect that related factors could have. Through the different size groups and the adoption of continuous integration defined in Table 3.1, we applied statistical tests to understand whether these are possible factors influencing the prevalence of ML-CS in ML-enabled systems. At first, we hypothesize that different types of ML-CS may differ in prevalence. Hence, we formulated the following null hypothesis:

H0: *There is no statistically significant difference between the prevalence of the smells i and j.*

with i and j belonging to the set of ML smells S considered in the study. We identified as alternative hypotesis:

Ha0 : There is a statistically significant difference between the prevalence of the smell i and j.

Secondly, we hypothesized that the prevalence of ML-CS may depend on the size of the ML projects. Larger projects may be more complex and involve more contributors, increasing the likelihood of introducing code smells during development. As such, we formulated the following null hypothesis:

H1: *There is no statistically significant difference in the prevalence of the smell i among large, medium, and small projects.*

with i belonging to the set of ML smells S considered in the study, large projects being those having a size (in terms of lines of code) above the 60^{th} percentile of the distribution of the sizes of all projects, medium projects being those having a size between the 30^{th} and 60^{th} percentile of the distribution of the sizes of all projects, and small projects being those having a size lower than the 30^{th} percentile of the distribution of the sizes of all projects. We identified the following alternative hypotheses:

Ha1: There is a statistically significant difference in the prevalence of smell i between small and medium projects.
Ha2: There is a statistically significant difference in the prevalence of smell i between small and large projects.
Ha3: There is a statistically significant difference in the prevalence of smell i between medium and large projects.

We hypothesized that projects relying on a CI pipeline may have a lower prevalence of code smells than those not relying on that. Indeed, the presence of a CI pipeline may directly affect the quality assurance mechanisms implemented by the projects, possibly affecting the presence of ML code smells. Hence, we formulated our last null hypothesis:

H2: *There is no statistically significant difference in the prevalence of the smell i between projects relying and not on a Continuous Integration pipeline.*

with i belonging to the set of ML smells S considered in the study. We formulated as an alternative hypothesis:

Ha2 There is a statistically significant difference in the prevalence of the smell i between projects relying on and not on a Continuous Integration pipeline.

Regarding statistical verification, we used different tests for the three hypotheses we formulated. For *H0* and *H2*, we used the non-parametric Wilcoxon test [6], which investigates significant differences between two populations. Then, for the analysis for *H1* and given the goal of exploring differences between three groups, we used a test that allows us to study differences across more than two populations: the non-parametric Friedman test.

The results were statistically significant at α=0.05. We normalized the data distribution by the project LOC to avoid possible biases and quantify the effect size using the Cliff's Delta (δ) [5].

RQ$_1$: *When are ML-Specific code smells introduced in ML-enabled systems?* After the commit data extraction phase described in Section 3.5, we collected all the smell-introducing commits to understand when each ML-CS is introduced. The outline of the smell-introducing commits allowed us to understand which ML-CSs are introduced during file creation and which occur during the evolution and maintenance of ML projects. To gain insights into the lifecycle of ML-specific code smells, we also implemented a segmentation approach based on three key factors: *development time*, *activity levels*, and *distance from the release*. The first two segments were used to examine the moment ML-CSs are introduced. Each commit is categorized based on its duration since the project started and its position in the commit history. Subsequently, we analyzed within each segment to determine the presence of smell-introducing commits. In addition to these segments, the third segment investigates the relationship between the introduction of ML-CSs and project releases. We identified commits labeled as "Release" using PyDriller and categorize all other commits based on their proximity to the subsequent project release (*e.g.*, one day before the next release). By examining the temporal proximity of code smell occurrences to release events, we aim to ascertain whether the timing of releases influences developers' proneness to introduce ML-CSs. Table 7 provides the segments and the value used for the segmentation.

Table 7: Segmentation of commits for analyzing the introduction of ML-CSs.

Tag	Description	Values
Development Time	Based on the duration since the project's starting date	[one week, one month, one year, more than one year]
Activity Level	Based on its sequence in the project's commit history, identifying the number of previous commits.	[first 10% of commits, first 20% of commits, first 50% of commits, after the first 50% of commits]
Distance from a Release:	Based on the time elapsed before the next release.	[one day, one week, one month, more than one month]

RQ$_2$: *What tasks were performed when the ML-Specific code smells were introduced?* After collecting the list of smell-introducing commits for all ML-CS instances, we analyzed their messages to explain the rationale behind the changes. Specifically, we leveraged pattern matching, as previously done by Tufano et al. [34] to analyze why traditional code smells are introduced. In detail, we extracted the rationale, starting from the label set indicating the change operations described in Table 8. Finally, we analyzed to what extent commit rationales and introduced ML-CSs co-occur.

RQ$_3$: *When and how are ML-specific code smells removed in ML-enabled systems?* After analyzing the conditions and reasons for introducing ML-CSs, we

Table 8: Change operation tags for the rationale analysis.

Tag	Description
Bug Fixing	The commit aimed at fixing a bug.
Enhancement	The commit aimed at implementing an enhancement in the system.
New Feature	The commit aimed at implementing a new feature in the system.
Refactoring	The commit aimed at performing refactoring operations.

performed a similar analysis from the smell-removing commits. As for **RQ**$_1$, we first verified whether ML-CSs are mitigated in a smell-removing commit and identify which are unmitigated, employing the same segmentation adopted and represented in Table 7. Afterward, we focused on the smell-removing commits. We collected the messages of all smell-removing commits using the pattern matching approach adopted in **RQ**$_2$, relying on the tags in Table 8 to extract the rationale behind the removal. This analysis allowed us to extract the refactoring operations addressing ML-CSs. From the set of the smell-removing commits analyzed, we considered apart the commits that did not perform changes to ML-CSs but removed them by deleting the files.

RQ$_4$: *How long do ML-Specific code smells survive in the code?* To understand the lifetime of each ML-CS instance, we computed the number of commits from the smell-introducing commit to the smell-removing commit and the period in days. Given such values, we computed the mean lifespan of each ML-CS type to understand which smells survive for a longer lifespan.

3.7 Public Data Availability

To ensure the replicability of this work and enable researchers to build upon our study, we released all materials, including scripts and datasets, in an online appendix hosted in permanent storage [25]. In addition, we included the GitHub repository link for CodeSmile: `https://github.com/giammariagiordano/smell_ai/tree/main`.

4 Analysis of the Results

This section presents the results of the study. For the sake of clarity, we split the discussion by research question.

4.1 **RQ**$_0$: How are ML-specific code smells prevalent in ML-enabled systems?

Using our static analysis tool, we identified 8,542 ML-CSs across the analyzed projects, highlighting the prevalence of technical debt in ML systems. However, as illustrated in Figure 4, the distribution of these smells is highly uneven.

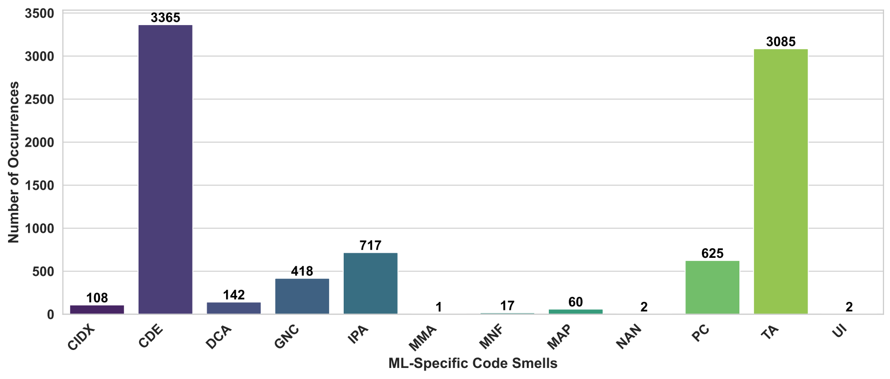


Fig. 3: Number of occurrences of ML-CSs in the analyzed projects. The code smells considered are: *'Chain Indexing'* (CIDX), *'Columns and DataType Not Explicitly Set'* (CDE), *'Gradients Not Cleared Before Backward Propagation'* (GNC), *'In-Place APIs Misused'*(IPA), *'Matrix Multiplication API Misused'* (MMA), textsl'Memory Not Freed' (MNF), *'Merge API Parameter Not Explicitly Set'* (MAP), *'NaN Equivalence Comparison Misused'* (NAN), *'PyTorch Call Method Misused'* (PC), *'TensorArray Not Used'* (TA), and *'Unnecessary Iteration'* (UI).

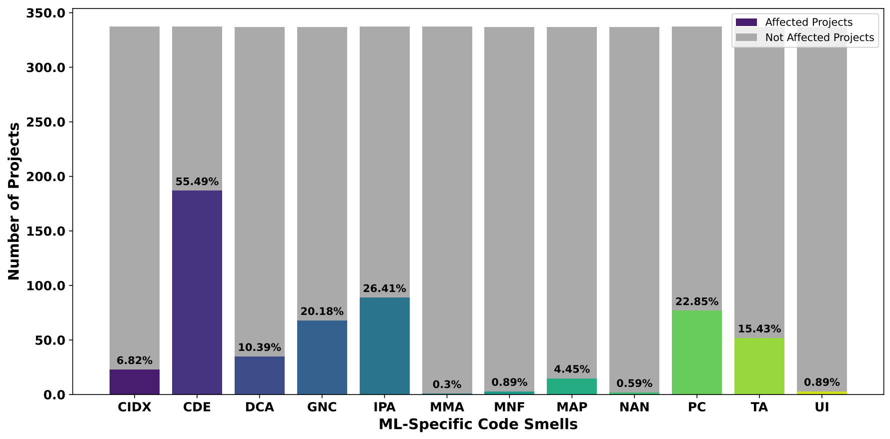


Fig. 4: Number of projects affected by ML-CSs. The code smells considered are: *'Chain Indexing'* (CIDX), *'Columns and DataType Not Explicitly Set'* (CDE), *'Gradients Not Cleared Before Backward Propagation'* (GNC), *'In-Place APIs Misused'*(IPA), *'Matrix Multiplication API Misused'* (MMA), textsl'Memory Not Freed' (MNF), *'Merge API Parameter Not Explicitly Set'* (MAP), *'NaN Equivalence Comparison Misused'* (NAN), *'PyTorch Call Method Misused'* (PC), *'TensorArray Not Used'* (TA), and *'Unnecessary Iteration'* (UI).

The most recurrent smell *'Column and Datatype Not Explicitly Set'* accounts for 3,365 instances, representing a significant portion of the total occurrences. Similarly, a substantial number of TA smells were detected, with 3,085 instances across all projects, indicating widespread inefficiencies in managing tensor data structures. Moderately frequent smells include 717 instances of

'In-Place APIs Misused' and 418 instances of *'Gradients Not Cleaned Before Backpropagation'*, both of which have the potential to introduce subtle bugs or degrade performance in ML pipelines. Furthermore, while 625 instances of *'Pytorch Call Method Misused'* were detected, the relatively lower detection performance for this specific smell introduces uncertainty about the actual prevalence. In addition to these more frequent smells, many other smells were identified, including 108 instances of *'Chain Indexing'* and 142 instances of *'Dataframe Conversion API Misused'*. Similarly, *'Merge API Parameter Not Explicitly Set'* was observed 60 times. In contrast, very low occurrences were found for several other smells: 17 instances of *'Memory Not Freed'*, two of *'Unnecessary Iteration'*, two of *'NaN Equivalence Comparison Misused'*, and only one instance of *'Matrix Multiplication API Misused'*. While less common, these low-frequency smells could still have significant implications in specific scenarios and warrant further attention in targeted contexts. To complement the analysis of absolute occurrences, we examined the prevalence of each ML-CS across projects. Figure 4 shows the percentage of projects affected by each ML-CS, providing a broader view of how widely these issues are distributed. The results highlight notable differences in how smells propagate across repositories. For instance, *'Columns and DataType Not Explicitly Set'* affects over half of the projects (55.49%), suggesting it is a generalizable and widespread issue across ML projects. In contrast, while similar in total occurrence count, *'TensorArray Not Used'* appears in only 15.43% of projects—suggesting that it is more domain-specific and concentrated in fewer repositories. Similarly, smells like *'Gradients Not Cleared Before Backward Propagation'* and *'In-Place APIs Misused'* affect 20.18% and 26.41% of projects, respectively, while others such as *'Merge API Parameter Not Explicitly Set'* or *'Matrix Multiplication API Misused'* are found in less than 5% of the projects. This distribution-based perspective highlights that while the occurrences of some ML-CSs may be high, differences are notable in terms of distribution across ML projects, as some smells—despite high absolute counts—may be localized, while others are more pervasive and thus require more generalized mitigation strategies.

It is important to interpret the observed prevalence of smells in the context of the detection performance of our tool, particularly its recall. As shown in Table 6, recall values vary significantly across smells, ranging from 0.222 for *'Unnecessary Iteration'* to 1.000 for smells such as *'Columns and Datatype Not Explicitly Set'* and *'NaN Equivalence Comparison Misused'*. This variation suggests that our findings primarily reflect the subset of ML-CSs that fall within the current detection capabilities of CodeSmile. For instance, the high number of *'Columns and Datatype Not Explicitly Set'* instances is consistent with the tool's perfect recall for this smell, which increases our confidence in the reported values. In contrast, smells like *'DataFrame Conversion API Misused'* and *'In-Place APIs Misused'*, with recall values of 0.475 and 0.636, respectively, may be underrepresented in the current results. Their actual prevalence could be significantly higher than what our tool was able to detect. Therefore, while our approach already uncovers a substantial number of ML-CS instances, it

Table 9: Wilcoxon Test Results. The code smells are: *'In-Place APIs Misused'* (IPA), *'Columns and DataType Not Explicitly Set'* (CDE), *'TensorArray Not Used'* (TA),*'DataFrame Conversion API Misused'* (DCA), *'Chain Indexing'* (CIDX), and *'DataFrame Conversion API Misused'* (MAP).

Smell A	Smell B	Statistic	P-Value	Cliff's Delta (δ)	P-Value (Bonferroni)
GNC	PC	44818.0	0.3906	-0.0301	1.0
GNC	IPA	43158.5	0.0672	-0.0660	1.0
GNC	CDE	26967.0	**<0.0001**	-0.4164***	**<0.0001**
GNC	TA	48557.0	0.1186	0.0508	1.0
GNC	DCA	51454.5	**0.0002**	0.1135	**0.0059**
GNC	CIDX	53285.0	**<0.0001**	0.1532*	**<0.0001**
GNC	MAP	54611.5	**<0.0001**	0.1819*	**<0.0001**
PC	IPA	44505.0	0.3164	-0.0369	1.0
PC	CDE	28381.0	**<0.0001**	-0.3858***	**<0.0001**
PC	TA	49945.0	**0.0158**	0.0809	0.4423
PC	DCA	52857.5	**<0.0001**	0.1439	**0.0002**
PC	CIDX	54663.5	**<0.0001**	0.1830*	**<0.0001**
PC	MAP	56011.5	**<0.0001**	0.2122*	**<0.0001**
CDE	TA	67352.5	**<0.0001**	0.4576**	**<0.0001**
CDE	DCA	70469.0	**<0.0001**	0.5250***	**<0.0001**
CDE	CIDX	72116.0	**<0.0001**	0.5607***	**<0.0001**
CDE	MAP	73422.5	**<0.0001**	0.5890***	**<0.0001**
TA	DCA	49008.5	**0.0337**	0.0606	0.9444
TA	CIDX	50809.0	**0.0002**	0.0996	**0.0056**
TA	MAP	52103.5	**<0.0001**	0.1276	**<0.0001**
DCA	CIDX	48078.5	0.0902	0.0405	1.0
DCA	MAP	49393.5	**0.0020**	0.0689	0.0566
CIDX	MAP	47505.5	0.1535	0.0281	1.0

is likely that the true number of occurrences is considerably higher, especially for smells with lower recall.

In the following, details of the difference in the prevalence of smells are presented based on the type of smell, the size of the project, and the adoption of continuous integration. To ensure a reliable and significant statistical analysis, we checked the minimum requirements needed to ensure the effectiveness of each test. Therefore, *'Matrix Multiplication API Misused'*, *'NaN Equivalence Comparison Misused'*, and *'Unnecessary Iteration'* are not considered in our hypothesis testing analysis, with a sample size of less than five [29].

H0: Differences among the prevalence of smells. The results of the Wilcoxon tests, presented in Table 9, provide a comparative analysis of the statistical differences between various pairs of code smells. Each comparison is evaluated based on the Wilcoxon test statistic, p-value, and Cliff's Delta (δ), which measures the magnitude of the effect. The effect sizes are classified as negligible (<0.15), small ($\geq$0.15, <0.33), medium ($\geq$0.33, <0.47), and large ($\geq$0.47) [5], as indicated by asterisks in the table. In the following, the findings are described relying on these levels.

Comparisons involving *'Columns and DataType Not Explicitly Set'* consistently reveal significant differences across all pairings, with very low p-

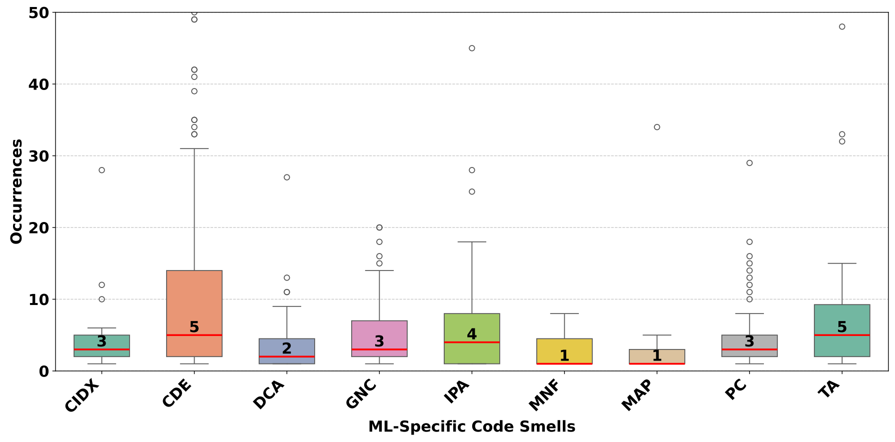


Fig. 5: Distribution of the occurrences of ML-CSs in affected ML projects in every project analyzed. The median values of the distributions are reported in bold. The smells are: *'Chain Indexing'* (CIDX), *'Columns and DataType Not Explicitly Set'* (CDE), *'DataFrame Conversion API Misused'* (DCA), *'Gradients Not Cleared Before Backward Propagation'* (GNC), *'In-Place APIs Misused'* (IPA), *'Memory Not Freed'* (MNF), *'Merge API Parameter Not Explicitly Set'* (MAP), *'PyTorch Call Method Misused'* (PC), and *'TensorArray Not Used'* (TA).

values. These comparisons exhibit medium to large effect sizes, underscoring the distinctiveness of this smell. For instance, the pairing of *'Gradients Not Cleared Before Backward Propagation'* and *'Columns and DataType Not Explicitly Set'* shows a medium effect size (δ = -0.4164***), while *'Columns and DataType Not Explicitly Set'*, compared to *'DataFrame Conversion API Misused'*, results in a large effect size (δ = 0.5250***). These findings highlight the strong difference between *'Columns and DataType Not Explicitly Set'* occurrences and other smells. Moreover, the comparisons involving *'Columns and DataType Not Explicitly Set'* consistently result in large effect sizes, such as *'Columns and DataType Not Explicitly Set'* vs. *'Chain Indexing'* (δ = 0.5607***) and *'Columns and DataType Not Explicitly Set'* vs. *'Merge API Parameter Not Explicitly Set'* (δ = 0.5890***). Similarly, *'Merge API Parameter Not Explicitly Set'* shows significant differences in most comparisons, with small to moderate effect sizes. For example, *'Merge API Parameter Not Explicitly Set'* compared to *'Gradients Not Cleared Before Backward Propagation'* yields a small effect size (δ = 0.1819*), while *'Merge API Parameter Not Explicitly Set'* compared to *'PyTorch Call Method Misused'* shows a small effect size (δ = 0.2122*). These values indicate a notable but less pronounced prevalence compared to *'Columns and DataType Not Explicitly Set'*.

Certain comparisons, such as those involving *'TensorArray Not Used'*, reveal smaller, yet statistically significant, differences.

For instance, the pairing of *'TensorArray Not Used'* with *'Chain Indexing'* has a p-value of 0.0056 and a negligible effect size (δ = 0.0996), and *'TensorArray Not Used'*, compared to *'Merge API Parameter Not Explicitly Set'*, results

Table 10: Friedman and Nemenyi Test Results Highlighting Differences Between Size-related Groups. The smells considered are: *'Gradients Not Cleared Before Backward Propagation'* (GNC), *'In-Place APIs Misused'* (IPA), *'Columns and DataType Not Explicitly Set'* (CDE),*'TensorArray Not Used'* (TA), *'DataFrame Conversion API Misused'* (DCA),*'Chain Indexing'* (CIDX), *'Merge API Parameter Not Explicitly Set'* (MAP), *'PyTorch Call Method Misused'* (PC).

Friedman Test							
GNC	**IPA**	**CDE**	**TA**	**DCA**	**CIDX**	**MAP**	**PC**
0.971	0.200	<**0.001**	**0.031**	0.270	**0.027**	**0.011**	0.211
Nemenyi Test							
Smell	**S-M**	**S-M (δ)**	**S-L**	**S-L (δ)**	**M-L**	**M-L (δ)**	
CDE	**0.0256**	-0.2245*	**0.0003**	-0.3580**	0.4195	-0.1292	
TA	0.9919	-0.0126	0.3739	-0.1453	0.4433	-0.1312	
CI	0.8156	-0.0668	0.5172	-0.1196	0.8776	-0.0523	
MAP	0.9292	-0.0395	0.5938	-0.1053	0.8156	-0.0654	

in a negligible effect size ($\delta = 0.1276$). These findings suggest that *'TensorArray Not Used'* has a more limited prevalence than other smells. Finally, some comparisons did not reveal statistically significant differences or meaningful effect sizes. For example, the pairing of *'Chain Indexing'* compared to *'Merge API Parameter Not Explicitly Set'* results in a non-significant p-value of 1 and a negligible effect size ($\delta = 0.0281$), underscoring the minimal distinction between these smells.

Observing the distribution of the projects affected by each ML-CSs, represented in Figure 5, additional insights emerge. *'Columns and DataType Not Explicitly Set'* and *'TensorArray Not Used'* have the highest median values of 5, indicating that they are the most frequently occurring code smells in the dataset. However, the distribution of *'TensorArray Not Used'* extends in the upper quartile to less than 10, and the *'Columns and DataType Not Explicitly Set'* distribution shows an extension of the upper quartile to 14 occurrences per project and a maximum data value of occurrences at 31, highlighting the probability of having a lot of occurrences of this ML-Cs inside a single ML-project. *'Merge API Parameter Not Explicitly Set'* and *'DataFrame Conversion API Misused'* show moderate median occurrences of 3, reflecting consistent but less dominant frequencies. PC presents a lower median of 2. In contrast, *'Gradients Not Cleared Before Backward Propagation'*, *'In-Place APIs Misused'*, and *'Chain Indexing'* have the lowest medians of 1, suggesting that these are the least frequent code smells.

H1: Differences of smell prevalence among the project size. To analyze the difference in size among the projects, stratified as small, medium, and large, we first applied the Friedman Test to each smell. It is important to note here that the distributions of smell occurrences are normalized by the number of LOC of each project. Therefore, this test is not limited to understanding the difference between the project size but also whether this difference is disproportionate to the project size.

Table 11: Mann-Whitney U Test and Cliff's Delta Results for Analyzing Statistically Significant Differences of Smells Between CI and Non-CI Projects (Normalized by LOC). The table shows the following smells: *'Gradients Not Cleared Before Backward Propagation'* (GNC), *'PyTorch Call Method Misused'* (PC), *'In-Place APIs Misused'*(IPA),*'Columns and DataType Not Explicitly Set'* (DCE), *'TensorArray Not Used'*(TA), *'DataFrame Conversion API Misused'*(DCA), *'Chain Indexing'* (CIDX), and *'Merge API Parameter Not Explicitly Set'* (MAP).

Smell	Mann-Whitney U	P-Value	Cliff's Delta	P-Value (Bonferroni)
GNC	9629.0	**0.0040**	-0.1413	**0.036**
PC	10997.0	0.7074	-0.0194	1.0
IPA	12469.0	**0.0387**	0.1119	0.34
CDE	12777.0	**0.0331**	0.1394	0.29
TA	10150.5	**0.0319**	-0.0948	0.28
DCA	11853.0	0.1270	0.0570	1.0
CIDX	11888.0	0.0517	0.0601	0.46
MAP	11252.0	0.8947	0.0034	1.0

The results of the Friedman Test are reported in Table 10. Among the analyses of all the smells, significant overall differences were observed for four cases: *'Columns and DataType Not Explicitly Set'* ($p < 0.001$), TA (p =0.031), *'Chain Indexing'* (p =0.027), and *'Merge API Parameter Not Explicitly Set'* (p =0.011). These findings suggest that these smells are influenced by size-related group distinctions (Small, Medium, and Large). Conversely, the other smells, such as *'Gradients Not Cleared Before Backward Propagation'*, *'In-Place APIs Misused'*, *'DataFrame Conversion API Misused'* , and *'PyTorch Call Method Misused'* , did not exhibit statistically significant overall differences, indicating that size variations may not be critical in their occurrence.

H2: Differences of smell prevalence among the CI adoption projects. Table 11 reports the results of the Mann-Whitney U Test and Cliff's Delta for analyzing differences in smell occurrences between projects adopting CI and those not. To ensure the robustness of our findings, we also applied the Bonferroni correction for multiple comparisons. Findings show that only one ML-specific code smell (ML-CS) exhibited a statistically significant difference. Specifically, *'Gradients Not Cleared Before Backward Propagation'* showed a significant reduction in CI projects ($p = 0.0036$), with Cliff's Delta ($\delta = -0.1413$) indicating a small effect size. This suggests a slight tendency for CI projects to better manage gradient clearing compared to non-CI projects. For the other smells analyzed, some initially appeared significant under the uncorrected test but did not remain so after applying the Bonferroni correction. For instance, *TensorArray Not Used'* showed a decrease in CI projects ($p = 0.0319$, $\delta = -0.0948$), with a small effect size reflecting potentially better tensor management. Conversely, two smells—*'In-Place APIs Misused'* and *'Columns and DataType Not Explicitly Set'*—were slightly more prevalent in CI projects. The former showed a higher occurrence in CI projects ($p = 0.0387$, $\delta = 0.1119$), as did the latter ($p = 0.0331$, $\delta = 0.1394$), both with small effect sizes. These trends may hint at minor increases, possibly influenced by the fast-paced and iterative na-

Table 12: Overview of the number of projects, introducing commits, and initial set of commits.

Category	Projects (Commits)	Smell Projects (Commits)	Smell-Introducing Commits
Small	117 (64,607)	79 (77,101)	1,226
Medium	118 (71,380)	105 (72,118)	4,482
Large	142 (265,671)	125 (212,599)	11,667
Total	337 (401,658)	309 (361,818)	17,775

ture of CI workflows. No statistically significant differences were observed for the remaining smells—*'PyTorch Call Method Misused'*, *'DataFrame Conversion API Misused'*, *Chain Indexing'*, and *'Merge API Parameter Not Explicitly Set'*—as all had p-values greater than 0.05. In summary, the statistically significant results observed are characterized by small effect sizes. While CI adoption appears to reduce the occurrence of *'Gradients Not Cleared Before Backward Propagation'*, overall, the relationship between CI practices and ML-CS occurrences remains subtle and smell-dependent.

☞ **Answer to RQ$_0$.** The static analysis tool identified 8,542 ML-CSs across projects, with some smells being significantly more prevalent, highlighting widespread inefficiencies in ML systems. Smell prevalence varied notably with project characteristics, such as size, with larger projects exhibiting disproportionately higher occurrences of certain smells. The occurrence of *'Gradients Not Cleared Before Backward Propagation'* shows differences associated with CI adoption, reflecting variations in development practices.

4.2 **RQ$_1$**:When ML-Specific Code Smells are introduced

Using CodeSmile, we analyzed 401,658 commits across 337 ML projects, identifying 17,775 smell-introducing commits from an initial set of 361,818 smelly commits as shown in Table 12. Smells were detected in 91.7% of projects, with the proportion of smell-introducing commits increasing with project size: 4.38% for small projects, 17.88% for medium projects, and 77.73% for large projects. The following analyses are based on these data, considering all the smell-introducing commits found in the three categories of projects.

Analysis on the modification type of the commit. Firstly, when collecting the file affected by ML-CSs, we also collected information on its change type in the specific commit to understand when the ML-CSs are introduced. As shown in Table 13, the distribution of change types—whether a file was newly created or modified—provides insight into how ML-CSs are introduced. The table presents the percentage of ML-CSs introduced in new files compared to those introduced in modifications to existing files, offering a deeper understanding of the contexts in which smells emerge. However, we noted that some removals may be incidental distinct patterns across different ML-CS types. For certain

Table 13: Percentage distribution of commit modification type of smell-introducing commits for every ML-CSs. The smells are: *'Chain Indexing'*(CIDX), *'Columns and DataType Not Explicitly Set'* (DCE), *'DataFrame Conversion API Misused'*(DCA), *'Gradients Not Cleared Before Backward Propagation'* (GNC),*'In-Place APIs Misused'* (IPA), *'Memory Not Freed'* (MNF), *'Merge API Parameter Not Explicitly Set'* (MAP), *'PyTorch Call Method Misused'* (PC), and *'TensorArray Not Used'* (TA).

Smell Name	Added	Modified	Total
CIDX	**71 (53.79%)**	61 (46.21%)	132
CDE	3044 (41.15%)	**4354 (58.85%)**	7398
DCA	**147 (56.76%)**	112(43.24%)	259
GNC	**568 (54.56%)**	473(45.44%)	1041
IPA	522(33.38%)	**1042 (66.62%)**	1564
MNF	1 (4.35) %	**22 (95.65%)**	23
MAP	**67 (77.91%)**	19 (22.09%)	86
PC	**658 (53.93%)**	562 (46.07%)	1220
TA	2943 (49.65%)	**2985(50.35%)**	5928

smells, a higher proportion of introductions occurs during file modifications. For example, *'DataFrame Conversion API Misused'* is introduced in 58.85% of cases through modifications, and *'In-Place APIs Misused'* exhibits an even higher proportion at 66.62%. Similarly, *'Memory Not Freed'* is mainly introduced during file modifications, with 95.65% of occurrences arising from this change type.

Conversely, some smells are more frequently introduced when creating new files. For instance, *'Merge API Parameter Not Explicitly Set'* has the highest proportion of introductions in newly created files at 77.91%, indicating that this smell often arises when new components or modules are added to a project. Similarly, *'DataFrame Conversion API Misused'* and *'Gradients Not Cleared Before Backward Propagation'* are introduced in 56.76% and 54.56% of the cases, respectively, during file creation. Certain smells, such as *'TensorArray Not Used'* and *'Pytorch Call Method Misused'*, show a near-equal distribution between new file creation and modifications, with 49.65% and 53.93% introduced during file creation, respectively. This balanced distribution indicates that these smells are not strongly associated with a specific type of change and can emerge in both contexts, reflecting their pervasive nature. Therefore, while ML-CSs are balanced through the introduction during the addition or modification of a specific file, the data suggests that ML-CSs are slightly predominantly introduced during the modification of existing files rather than creating new ones. This trend among ML-CSs reflects the ongoing development and maintenance activities within projects, where incremental changes to existing functionalities may inadvertently lead to the introduction of smells.

Analysis on activity and time-related metrics. Table 14 presents the percentage distribution of three key metrics: activity level (AL), development time (DT), and distance from release (DR) for each type of ML-CS. These metrics provide valuable insights into the contexts and timelines under which ML-CSs are introduced across projects. The AL metric, which represents the percentage of commits since the start of the project at which a smell was introduced,

Table 14: Percentage distribution of Activity Level, Development Time, and Distance from Release for smell-introducing commits. The smells are: *'Chain Indexing'* (CIDX), *'Columns and DataType Not Explicitly Set'* (CDE), *'DataFrame Conversion API Misused'* (DCA), *'Gradients Not Cleared Before Backward Propagation'* (GNC), *'In-Place APIs Misused'* (IPA), *'Merge API Parameter Not Explicitly Set'* (MAP), *'PyTorch Call Method Misused'* (PC), and *'TensorArray Not Used'* (TA).

	Class	CIDX	CDE	DCA	GNC	IPA	MAP	PC	TA
AL	10%	5.3%	14.5%	8.3%	7.2%	11.5%	0.0%	3.8%	6.2%
	20%	25.0%	12.4%	13.5%	8.5%	4.8%	35.7%	5.4%	9.3%
	50%	9.2%	18.7%	14.6%	17.8%	13.3%	0.00%	14.9%	**66.7%**
	>50%	**60.5%**	**54.4%**	**63.6%**	**66.6%**	**70.4%**	**64.3%**	**75.9%**	17.8%
DT	<7d	2.6%	2.1%	5.4%	3.7%	3.4%	0.0%	1.8%	3.8%
	<1m	5.3%	10.0%	5.4%	4.2%	4.5%	0.0%	2.8%	1.9%
	<1y	**57.9%**	37.4%	17.2%	34.6%	26.5%	63.6%	35.6%	19.9%
	>1y	34.2%	**50.5%**	**72.0%**	**57.5%**	**65.6%**	36.4%	**59.8%**	**74.4%**
DR	1d	0.0%	1.1%	0.0%	0.4%	1.3%	0.0%	0.2%	0.5%
	7d	2.6%	3.4%	0.0%	0.4%	4.4%	0.0%	1.1%	0.7%
	<1m	3.9%	7.7%	6.2%	6.4%	6.9%	21.4%	9.8%	3.7%
	>1m	**93.4%**	**87.8%**	**93.7%**	**92.7%**	**87.4%**	**78.6%**	**88.9%**	**95.1%**

shows that most ML-CSs are introduced after substantial project activity. For most smells, more than 50% of their introductions occur when the activity level exceeds 50%. Notably, smells such as *'PyTorch Call Method Misused'* and *'In-Place APIs Misused'* show the highest proportions in this range, with 75.9% and 70.4% of their occurrences introduced late in a project activity timeline. Conversely, *'TensorArray Not Used'* demonstrates a different pattern, with 66.7% of occurrences introduced earlier when activity levels are at 50% or below, indicating that this smell may emerge more frequently in earlier project stages.

The DT metric, which measures the time since the project started when a smell was introduced, highlights clear differences between smell types. Most smells are introduced more than a year into project development, with particularly high proportions for *'DataFrame Conversion API Misused'* at 72.0%, *'TensorArray Not Used'* at 74.4%, and *'In-Place APIs Misused'* at 65.6%. On the other hand, smells such as *'Chain Indexing'* are more likely to appear earlier, with 57.9% of occurrences within the first year.

The DR metric, which captures the timing of smell introduction relative to the next release, shows a strong tendency for ML-CSs to be introduced long before a release. Across all smell types, over 78.6% of occurrences are introduced more than one month before a release, with particularly high proportions for *'TensorArray Not Used'* (95.1%), *'DataFrame Conversion API Misused'* (93.7%), and *'Chain Indexing'* (93.4%).

In summary, combined with the previous analysis based on commit change type, the data reveals that ML-CSs are predominantly introduced during later stages of project activity and development, with most smells appearing after a year since the project started and well before upcoming releases.

Additional Analysis on the distance from the release. A key observation from our initial analysis was that the majority of ML-CSs tend to be introduced well before a release, with over 78.6% appearing more than one month in advance.

However, this finding does not account for the varying number of commits made at different times, which could influence the perceived distribution of ML-CSs. To better understand this relationship, we conducted a more granular analysis by normalizing the number of smell-introducing commits with respect to the total number of commits in each time frame. We computed the ratio of smell-introducing commits to the total number of commits to the four different time classes related to the distance from the next release. Table 15 presents the resulting percentage distributions, allowing us to compare the relative likelihood of smell introduction across different stages of development. The results indicate that the proportion of smell-introducing commits remains relatively stable over time, with no substantial variations across different time frames. This suggests that ML-CSs do not necessarily accumulate due to the pressures of a release but are instead introduced consistently throughout the development process.

While the overall trend appears stable, some ML-CSs exhibit slight variations. For example, the smell *'TensorArray Not Used'* shows the highest percentage in the day-before-release category (5.0%), suggesting that tensor-related inefficiencies may emerge more frequently during last-minute code modifications or final optimizations before deployment. On the other hand, *'Chain Indexing'* is most frequently introduced more than one month before release (4.0%), implying that inefficient indexing practices are often established early in the development lifecycle. Similarly, smells such as *Gradients Not Cleared Before Backward Propagation'* and *'In-Place APIs Misused'* also show their highest percentages in the more-than-one-month category, at 7.0% and 3.9% respectively. This suggests that these type of issues tend to be present from stable development. Interestingly, *'PyTorch Call Method Misused'* displays a relatively uniform distribution across all time windows, with the highest value in less than one month before the release (5.2%), but also with a relatively high ratio during the day before the release (5.1%), suggesting that this issue can occur during stable development activities, but also when a release is approaching.

Overall, this additional analysis reinforces the idea that the introduction of ML-CSs is not strongly correlated with the timing of software releases. Instead, these smells appear to be an inherent byproduct of ML development, influenced by iterative experimentation, evolving model architectures, and continuous code adjustments. However, some insights leads to understand that the distance from the release when considering the smell ratio could be related to part of the smells analyzed. Future research on this aspects is needed to extract the factors that lead the developers to introduce ML-CSs.

Table 15: Percentage Distribution of the ratio of smell-introducing commits over the total number of commits among ML-CSs (SR). The smells are: *'Chain Indexing'* (CIDX), *'Columns and DataType Not Explicitly Set'* (CDE), *'DataFrame Conversion API Misused'* (DCA), *'Gradients Not Cleared Before Backward Propagation'* (GNC), *'In-Place APIs Misused'* (IPA), *'Merge API Parameter Not Explicitly Set'* (MAP), *'PyTorch Call Method Misused'* (PC), and *'TensorArray Not Used'* (TA).

Total Number of Commits				
Commits	**<1d**	**<7d**	**<1m**	**<1m**
401,658	**15,812**	**54,819**	**124,283**	**206,744**

Ratio of Smell-Introducing Commits (SR)								
Class	**CIDX**	**CDE**	**DCA**	**GNC**	**IPA**	**MAP**	**PC**	**TA**
<1d	0.0%	3.9%	0.0%	2.7%	2.2%	0.0%	5.1%	**5.0%**
<7d	3.8%	**4.8**%	0.0%	0.9%	3.2%	0.0%	2.7%	3.8%
<1m	1.8%	4%	**2.2%**	3.9%	2.5%	0.4%	**5.2%**	3.9%
>1m	**4.0%**	4.3%	1.9%	**7%**	**3.9%**	**1.2%**	4.9%	4.2%

☞ **Answer to RQ$_1$.** ML-CSs are often introduced during the modification of existing files, as seen with smells like *'Columns and Datatype Not Explicitly Set'* (58.85%) and *'In-Place API Misused'* (66.62%), though some, such as *'Merge API Parameter Not Explicitly Set'* (77.91%), are more frequently introduced during new file creation. Activity and time-related metrics show that most ML-CSs are introduced in later stages of project development, often after significant activity and long before releases, reflecting their association with maintenance activities and incremental changes to existing functionalities.

4.3 **RQ$_2$**: What tasks were performed when the ML-Specific code smells were introduced?

Using the sets of commits extracted for the previous analysis, we applied a pattern-matching approach to detect the task operations performed during the introduction of ML-CSs. Applying the same approach used by Tufano et al. [34], we analyzed each commit message for the presence of specific keywords that indicate the nature of the task (*e.g.*, the presence of the keyword "refactor" to identify refactoring commits). However, it is important to note that a single commit can contain details related to multiple task types, as different operations can occur within a single commit. Therefore, a single smell-introducing commit can belong to one or more categories defined.

Table 16 summarizes the distribution of the number of commit operations types among ML-CSs. The results show that ML-CSs are most frequently introduced during new feature tasks, with 9187 commit operations, followed by Enhancements (8068), bug fixing (4302), and refactoring (5826). Across individual smells, *'TensorArray Not Used'* and *'Columns and DataType Not Explicitly Set'* are the most frequently introduced, with 3813 and 3357 commits during new feature tasks, respectively. These smells also dominate other categories, with *'TensorArray Not Used'* leading in Enhancements (3492) and *'Columns and DataType Not Explicitly Set'* prevalent in bug fixing (1834) and refactoring (2243). Smells like *'In-Place APIs Misused'* and *'Gradients Not Cleared Before Backward Propagation'* also show high frequencies in new feature tasks, with 766 and 496 commits, respectively, and appear across other task types. In contrast, less frequent smells like *'Memory Not Freed'* and *'Merge API Parameter Not Explicitly Set'* are primarily introduced during new feature tasks, with 18 and 31 commits, respectively.

Interestingly, refactoring tasks account for a significant number of smell introductions, especially for *'Columns and DataType Not Explicitly Set'* (2243) and *'TensorArray Not Used'* (2234), suggesting that while refactoring aims to improve quality on one side, developers can add quality issues that can counterbalance the situation. Going deeper into the analysis, we found some projects that introduced quality issues while performing changes to refactor a specific Python module. For instance, in the `BrikerMan/Kashgari` project, a commit is delivered to perform refactoring operations in the tokenizer modules.[5] A particular refactoring operation added a *'Columns and DataType Not Explicitly Set'* smell in the related *corpus.py* module, resulting in a smell-introducing commit. A similar situation was found in five other medium and two large projects while the data results were being inspected. This additional insight suggests that the primary objective of these refactoring efforts may have been to improve general code structure, readability, or modularity across files without explicitly considering ML-specific quality concerns.

However, these commits can inadvertently introduce ML-CSs. Therefore, this suggests that while the intention behind refactoring activities is generally positive—aimed at improving code maintainability, readability, and modularity—these efforts can inadvertently degrade the quality of the system from an ML-specific perspective. This can occur because ML-CSs often arise from the unique characteristics and complexities of machine learning codebases, which are not always accounted for in standard refactoring practices.

☞ **Answer to RQ$_2$.** ML-CSs are introduced during new feature tasks, which account for the highest number of smell-introducing commits. *'TensorArray Not Used'* and *'Columns and Datatype Not Explicitly Set'* are the most frequently introduced among individual smells across all task types. New feature tasks significantly contribute to their occurrences. Still, refactoring tasks also account for a substantial number of ML-CS introductions, highlighting the unintended consequences of quality improvement efforts that do not explicitly consider ML-specific concerns.

[5] Case of a *'Columns and DataType Not Explicitly Set'* smell-introducing commit when performing refactoring in BrikerMan/Kashgari project: `https://github.com/BrikerMan/Kashgari/commit/f7fb43d2f3651fbba92eb6e5cee8bfd279b0317a`

Table 16: Occurrences of commit operations performed when introducing ML-CSs. The table shows the following smells: *'Chain Indexing'* (CIDX), *'Columns and DataType Not Explicitly Set'* (CDE), *'DataFrame Conversion API Misused'* (DCA), *'Gradients Not Cleared Before Backward Propagation'* (GNC), *'In-Place APIs Misused'* (IPA), *'Memory Not Freed'* (MNF), *'Merge API Parameter Not Explicitly Set'* (MAP), *'PyTorch Call Method Misused'* (PC), and *'TensorArray Not Used'* (TA). Percentages are computed per row.

Smell Name	New Feature	Bug Fixing	Enhancement	Refactoring
CIDX	**72 (32.4%)**	32 (14.4%)	70 (31.5%)	48 (21.6%)
CDE	**3357 (32.6%)**	1834 (17.8%)	2864 (27.8%)	2243 (21.8%)
DCA	116 (26.7%)	88 (20.3%)	**122 (28.1%)**	108 (24.9%)
GNC	**496 (33.4%)**	230 (15.5%)	435 (29.3%)	325 (21.9%)
IPA	**766 (33.7%)**	385 (16.9%)	616 (27.1%)	507 (22.3%)
MNF	**18 (60.0%)**	3 (10%)	5 (16.6%)	4 (13.3%)
MAP	**31 (38.7%)**	17 (21.5%)	17 (21.5%)	15 (18.75%)
PC	**518 (31.8%)**	319 (19.6%)	447 (27.5%)	342 (21%)
TA	**3813 (34.9%)**	1394 (12.7%)	3492 (31.9%)	2234 (20.4%)
Total	**9187 (33.5%)**	**4302 (15.7%)**	**8068(29.5%)**	**5826 (21.3%)**

4.4 **RQ$_3$**: When and how are ML-specific code smells removed in ML-enabled systems?

Similar to the analysis performed in **RQ$_1$** and **RQ$_2$**, we collected information on the smell-removing commits. In detail, we started collecting the last commit for which CodeSmile can detect the presence of the instance of the ML-CSs, detecting, therefore, the last commit that signals the presence of the ML-CS. Table 17 provides an overview of the distribution of projects and commits containing ML-CSs, focusing on the number of commits where a smell is last observed. For small projects, 223 out of 64,607 commits represent the last occurrence of an ML-CS, indicating a relatively low frequency of lingering smells in smaller codebases. Medium projects show a higher proportion, with 639 out of 71,380 commits marking the last instance of a smell. Large projects exhibit the highest absolute number of last-smell commits, with 1,518 out of 265,671 commits. Overall, across all 337 projects (401,658 commits), 2,380 commits were identified as the last-smell commits.

However, while the set of commits found represents the last moment the smell is identified, different cases can be observed. First the last commit of the project history still denotes the presence of the smell, indicating that the smell was not removed until the date of this study. Moreover, the smell can be removed because the file is deleted. Finally, a rename operation of the file can occur, resulting in the presence of the smell in a new file. To narrow our analysis on the specific smell-removing commit, we collected this information using Pydriller to understand how the commit operates in the file of interest for ML-CSs. Table 18 reflects the percentage of the type of operation the smell last commit performs. For the goal of this analysis, we consider smell-removing commits the commit performing a modification or a deletion of the file. Out of 2,380 file operations analyzed, 1,352 (56.8%) involve file modifications, indicating that most ML-CSs were resolved through direct edits. Smells such as *'DataFrame Conversion API Misused'* (637 modifications, 54.4%) and

Table 17: Overview of the number of projects, removing commits, and initial set of commits.

Category	Projects (Commits)	Smell Projects (Commits)	Smell last Commit
Small	117 (64,607)	79 (77,101)	223
Medium	118 (71,380)	105 (72,118)	639
Large	142 (265,671)	125 (212,599)	1518
Total	337 (401,658)	309 (361,818)	2380

Table 18: Counts of file operations performed when removing ML-CSs (rename, delete, modify, not modify) by smell name across all project sizes. The smells are: *'Chain Indexing'* (CIDX), *'Columns and DataType Not Explicitly Set'* (CDE), *'DataFrame Conversion API Misused'* (DCA), *'Gradients Not Cleared Before Backward Propagation'* (GNC), *'In-Place APIs Misused'* (IPA), *'Memory Not Freed'* (MNF), *'Merge API Parameter Not Explicitly Set'* (MAP), *'PyTorch Call Method Misused'* (PC), and *'TensorArray Not Used'* (TA).

Smell Name	Delete	Modify	Rename	not modified	Total
CIDX	0	6	12	0	18
CDE	4	637	431	99	1171
DCA	2	14	4	1	21
GNC	0	77	119	8	204
IPA	2	218	58	16	294
MNF	0	6	1	0	7
MAP	0	4	2	0	6
PC	0	137	107	10	254
TA	9	253	132	9	403
Total	**17**	**1352**	**868**	**143**	**2380**

'TensorArray Not Used' (253 modifications, 62.8%) show focused maintenance efforts, while *'In-Place APIs Misused'* exhibits the highest proportion of modifications (218, 74.1%). In contrast, smells like *'Chain Indexing'* show fewer modifications (6 out of 18, 33.3%) and a higher reliance on renaming (66.7%), suggesting that certain smells are more often addressed by file restructuring than direct resolution. Based on this analysis, we will consider the 1,352 file modifications as smell-removing commits in the subsequent evaluation.

Analysis on activity and time-related metrics. Similar to the smell-introducing commits, we collected information about activity level (AL), development time (DT), and the distance from the next release (DR) to understand when ML-CSs are removed by developers, summarized in Table 19. Looking at the AL metric, similar to smell-introducing commits, most ML-CSs are removed late in the project activity timeline, with most removals occurring after 50% of project commits. Notably, *'PyTorch Call Method Misused'* has the highest proportion of removals in this range (91.9%), followed by *'In-Place APIs Misused'* (80.6%) and *'Merge API Not Explicitly Set'* (74.3%). However, the distinct pattern observed for TA, where most removals (64.6%) occur earlier in the activity timeline, indicates that certain smells may be considered for resolution in the earlier phases of a project. Focusing on the development time, for most ML-CSs, smell removal predominantly occurs over a year after the project starts. Smells like *'DataFrame Conversion API Misused'*,*'In-Place*

Table 19: Percentage distribution of Activity Level, Development Time, and Distance from Release for smell-removing commits. The smells considered are: *'Chain Indexing'* (CIDX), *'Columns and DataType Not Explicitly Set'* (CDE), *'DataFrame Conversion API Misused'* (DCA), *'Gradients Not Cleared Before Backward Propagation'* (GNC), *'In-Place APIs Misused'* (IPA), *'Merge API Parameter Not Explicitly Set'* (MAP), *'PyTorch Call Method Misused'* (PC), and *'TensorArray Not Used'* (TA).

	Class	CIDX	CDE	DCA	GNC	IPA	MAP	PC	TA
AL	10%	4%	14.9%	7.2%	7.2%	1.4%	0.0%	3.4%	4%
	20%	11.5%	9.8%	13.5%	9%	6.7%	23.5%	1.41%	6.3%
	50%	28%	17.3%	14.6%	18.2%	11.3%	2.2%	5%	**64.6%**
	>50%	**61.6%**	**56.4%**	**63.5%**	**61.8%**	**80.6%**	**74.3%**	**91.9%**	19.9%
DT	<7d	0%	0%	0%	0%	3.4%	0.0%	1.9%	3.8%
	<1m	0%	5.7%	0%	0%	4.5%	0.0%	2.8%	1.9%
	<1y	**50%**	29.5%	20%	50%	13.5%	**60.6%**	6.6%	37.5%
	>1y	**50%**	**64.7%**	**80%**	**50%**	**86.4%**	39.4%	**80%**	**62.5%**
DR	1d	0%	0%	0%	0%	0%	0%	0%	12.5%
	7d	2.4%	5.6%	0%	33.3%	3.3%	0%	0%	25%
	<1m	6%	7.5%	6.2%	0%	8.4%	**100%**	13.3%	12.5%
	>1m	**91.6%**	**86.8%**	**93.8%**	**66.7%**	**88.1%**	0%	**86.6%**	**50%**

API Misused' and *'PyTorch Call Method Misused'*, with over 70% of removals happening after the first year, reflect this trend. In contrast, smells such as *'Chain Indexing'*, removed within the first year and after that (50%), suggest that some issues are addressed sooner, likely due to their ease of resolution. Finally, a strong trend is observed across all ML-CSs when analyzing the DR metric, with over 78.57% of removals occurring more than one month before a release. Smells such as *'PyTorch Call Method Misused'* (86.8%), *'DataFrame Conversion API Misused'* (93.8%), and *'Chain Indexing'* (91.6%) are particularly likely to be resolved well in advance of releases, suggesting that these smells start to be resolved after a long time during periods of active maintenance rather than close to release deadlines. Overall, as for smell-introducing commits, these results suggest that ML-CSs are typically addressed during active and stable maintenance periods rather than in rushed, release-critical phases. While most smells are resolved later in the project lifecycle, certain smells, such as *'TensorArray Not Used'* and *'Gradients not Cleared Before Backward Propagation'*, show distinct patterns of early removal, emphasizing variability in resolution timelines based on the type of smell.

Tasks performed during the removal of ML-CSs. Table 20 summarizes the number of change-type operations (New Feature, Bug Fixing, Enhancement, and Refactoring) associated with smell-removing commits grouped by smell name. These results provide insights into the context and purpose of the changes to address ML-CSs. New feature changes are most frequently observed for *'Columns and DataType Not Explicitly Set'* (232 changes), *'In-Place APIs Misused'* (91 changes), and *'TensorArray Not Used'* (98 changes), indicating that smell removals often accompany the addition of new functionality in these cases. Bug Fixing changes account for 459 instances, with the highest contributions from *'Columns and DataType Not Explicitly Set'* (179 changes) and *'TensorArray Not Used'* (90 changes), suggesting a strong association between smell removal and resolving defects, particularly for these smells. Enhance-

Table 20: Number of change type operations grouped by smell name. The smells are: *'Chain Indexing'* (CIDX), *'Columns and DataType Not Explicitly Set'* (CDE), *'DataFrame Conversion API Misused'* (DCA), *'Gradients Not Cleared Before Backward Propagation'* (GNC), *'In-Place APIs Misused'* (IPA), *'Memory Not Freed'* (MNF), *'Merge API Parameter Not Explicitly Set'* (MAP), *'PyTorch Call Method Misused'* (PC), and *'TensorArray Not Used'* (TA).

Smell Name	New Feature	Bug Fixing	Enhancement	Refactoring
CIDX	3	3	3	2
CDE	232	179	299	278
DCA	4	5	7	6
GNC	37	27	39	37
IPA	91	78	99	99
MNF	2	2	2	3
MAP	1	1	1	1
PC	75	74	81	77
TA	98	90	153	123
Total	**543**	**459**	**684**	**626**

ment changes are the most common type overall, with 684 instances. Smells such as *'Columns and DataType Not Explicitly Set'* (299 changes), *'TensorArray Not Used'* (153 changes), and *'In-Place APIs Misused'* (99 changes) show significant numbers of enhancements committed during smell removal, reflecting efforts to improve the quality or functionality of the code. Refactoring changes, totaling 626 instances, are prevalent for smells like *'Columns and DataType Not Explicitly Set'* (278 changes), *'TensorArray Not Used'* (123 changes), and *'In-Place APIs Misused'* (99 changes), highlighting a focus on improving code structure and maintainability as part of the removal process. The smell-removing commits span various change types, with enhancements and refactoring commits representing the majority, indicating that the resolution of ML-CSs often aligns with broader efforts to improve code functionality and structure rather than being isolated changes. Smells such as *'Columns and DataType Not Explicitly Set'*, *'TensorArray Not Used'*, and *'In-Place APIs Misused'* consistently show the highest number of change operations across all categories, emphasizing their prominence in the introduction and resolution of ML-CSs.

However, these operations may not always be explicitly aimed at resolving the ML-CS but may involve other changes where the ML-CS transforms into a different form. For instance, in the project `RTIInternational/gobbli`, a refactoring commit described as "improving example prediction formatting in evaluate app" inadvertently removed a *'Columns and DataType Not Explicitly Set'* smell.[6] This removal involved replacing a DataFrame used to display performance metrics with a built-in function, `show_metrics()`. While this change eliminated the *'Columns and DataType Not Explicitly Set'* smell, highlighting readability concerns, it did not necessarily improve code readability, as the core issue remained unaddressed. Similar cases were observed in 14 projects: eight small projects, three medium projects, and three large projects.

[6] Case of a *'Columns and DataType Not Explicitly Set'* smell-removing commit when performing enhancement in RTIInternational/gobbli project:`https://github.com/RTIInternational/gobbli/commit/b93d184c610c3ae779607679501b4b1dafd30b28`

The findings suggest that addressing ML-CSs is often integrated into broader development activities, such as enhancements, refactoring, and bug fixing, rather than explicitly targeted. However, the frequent unintentional nature of smell removal highlights a potential gap in awareness or prioritization of ML-CSs during routine development, underscoring the need for tools and practices that not only detect ML-CSs but also guide developers in addressing them intentionally and effectively. Moreover, the late-stage introduction and resolution of smells like *'Columns and DataType Not Explicitly Set'*, *'TensorArray Not Used'*, and *'In-Place APIs Misused'* in both introduction and resolution emphasize their role in shaping the maintainability and functionality of ML-enabled systems, making them key targets for future research and tool development. However, we noted that some removals may be incidental—resulting from structural changes, such as file renaming or refactoring—rather than deliberate efforts to eliminate the smells. Therefore, these findings should be interpreted with caution with respect to the intention of the developer.

☞ **Answer to RQ$_3$.** The removal of ML-CSs is often integrated into routine development activities, with 56.8% of smell removals occurring through file modifications, particularly for smells like *'Columns and DataType Not Explicitly Set' (54.4%)*, *'TensorArray Not Used' (62.8%)*, and *'In-Place APIs Misused' (74.1%).* Most ML-CSs are resolved during stable maintenance periods, highlighting a preference for addressing these issues outside time-critical phases. Most ML-CS removal activities include enhancements (684 instances) and refactoring (626 instances). Smells like *'Columns and DataType Not Explicitly Set,' 'TensorArray Not Used,'* and *'In-Place APIs Misused'* dominate these categories, suggesting these smells are closely associated with maintainability-related activities, such as enhancements and refactoring.

4.5 ***RQ$_4$**: How long do ML-Specific code smells survive in the code?*

The survival time of ML-CSs was analyzed using two metrics: *commit activity (CA)*, expressed as a percentage of the project lifetime in commits, and project survivability time (ST), measured in days. Results are summarized in Table 21. These metrics provide a detailed understanding of how long ML-CS instances persist in code before their removal, offering insights into the persistence of smells from both activity and time perspectives.

The results reveal that most ML-CSs are resolved within the first 10% of project commit activity. For example, *'Columns and Datatype Not Explicitly set'* (90.6%), *'TensorArray Not Used'* (98.0%), and *'In-Place API Misused'* (96.6%) disappear relatively early, suggesting they often disappear during the early phases of code evolution. However, for some ML-CSs, the proportion of smells surviving for a longer duration increases with additional commit activity. For instance, *'Merge API Not Explicitly Set'* (66.7%) and *'Pytorch Call Method Misused'* (57.0%) demonstrate slower resolution rates, with a substantial percentage persisting beyond the early phases of project activity

'Merge API Parameter Not Explicitly Set', in particular, stands out for its longevity, with 33.3% of instances persisting beyond 50% of the project commit activity, suggesting the inherent complexity of the project.

In terms of time, most ML-CSs tend to disappear within one year of their introduction. Smells such as *'Merge API Parameter Not Explicitly Set'* (75.0%), IPA (63.2%), and *'PyTorch Call Method Misused'* (55.4%) tend to have shorter lifespans. However, a subset of smells exhibits extended survival times, with *'TensorArray Not Used'* (36.4%) and *'PyTorch Call Method Misused'* (18.5%) remain detectable in the codebase for over a year in a significant portion of cases. These longer lifespans may indicate delayed refactoring or more complex technical debt. In contrast, simpler smells such as *'DataFrame Conversion API Misused'* (25.0%) and *'Chain Indexing'* (16.7%) often disappear within the first seven days. These results indicate that most ML-CSs tend to have a short lifespan in projects, often disappearing within the first 10% of commit activity or within one year of being introduced. However, some types of smells persist significantly longer, indicating differences in how long these smells remain in code depending on their complexity. These results indicate that while most ML-CSs follow a common pattern of short lifespan, lasting less than a year and less than 10% of project commit activity, certain deviate from this trend. The observed variations in persistence across different smells underscore their unique characteristics and the challenges associated with their resolution under specific circumstances.

To assess whether developers actively remove ML-CSs over the lifetime of projects, we conducted a cumulative trend analysis of introduced, removed, and net active smells for each smell type (Figure 6). The results show that while some removals occur, the general trend across all smells is a steady net increase in active smells over time. For instance, *'Columns and DataType Not Explicitly Set'* and *'TensorArray Not Used'* exhibit rapid accumulation of introduced instances (green line), with removals (red line) trailing far behind. This leads to a continually rising net active count (orange line). *'Gradients Not Cleared Before Backward Propagation'* shows more evidence of removal activity compared to other smells, yet the number of introduced instances still outweighs removals. Similarly, *'In-Place APIs Misused'* and *'PyTorch Call Method Misused'* follow consistent upward trends in net active smells, with limited evidence of reduction. *'DataFrame Conversion API Misused'* and *'Merge API Parameter Not Explicitly Set'* show a slight increase in the removal of ML-CSs over time, implying that these smells tend to persist once introduced. Less frequently occurring smells such as *'Memory Not Freed'* and *'Chain Indexing'* also exhibit low levels of removal and accumulation, although their limited prevalence makes broader conclusions difficult.

In general, this cumulative analysis highlights that ML-specific smells, once introduced, are rarely addressed or removed. The persistently growing net active counts suggest that developers do not consistently engage in targeted refactoring to manage ML-CSs, leading to their accumulation over time.

Table 21: Percentage distribution of the survival time of ML-CSs from their to introduction to their removal in terms commit activity(CA) and Survival Time (ST). The table shows the following smells: *'Chain Indexing'* (CIDX),*'Columns and DataType Not Explicitly Set'* (CDE), *'DataFrame Conversion API Misused'* (DCA), *'Gradients Not Cleared Before Backward Propagation'* (GNC), *'In-Place APIs Misused'* (IPA), *'Merge API Parameter Not Explicitly Set'* (MAP), *'PyTorch Call Method Misused'* (PC), and *'TensorArray Not Used'* (TA).

	Class	CIDX	CDE	DCA	GNC	IPA	MAP	PC	TA
CA	<10%	**100.0%**	**90.6%**	**88.9%**	**84.2%**	**96.6%**	**66.7%**	**57.0%**	**98.0%**
	<20%	0.0%	4.1%	11.1%	5.0%	2.3%	0.0%	11.8%	1.6%
	<50%	0.0%	4.2%	0.0%	7.9%	1.0%	0.0%	30.9%	0.4%
	>50%	0.0%	1.1%	0.0%	3.0%	0.1%	33.3%	0.4%	0.0%
ST	<7d	16.7%	12.8%	25.0%	8.1%	11.0%	0.0%	7.2%	4.7%
	<1m	33.3%	14.2%	18.8%	15.1%	13.7%	25.0%	18.9%	11.5%
	<1y	**50.0%**	**52.6%**	**31.3%**	**53.5%**	**63.2%**	**75.0%**	**55.4%**	**47.4%**
	>1y	0.0%	20.4%	25.0%	23.3%	12.1%	0.0%	18.5%	36.4%

Answer to RQ$_4$. The majority of ML-CSs are resolved within the first 10% of project commit activity, with high-resolution rates observed for *'Columns and Datatype Not Explicitly set'* (90.6%), *'TensorArray Not Used'*(98.0%), and *'In-Place API Misused'* (96.6%). While most ML-CSs exhibit short lifespans, their persistence varies significantly depending on their complexity and context. Proactive management strategies are crucial to address longer-living smells and prevent the accumulation of technical debt.

5 Discussion and Take-Away Messages

The findings of this study have several implications for both researchers and practitioners, highlighting how ML-CSs emerge, evolve, and can be addressed effectively in ML-enabled systems.

On the Smell Introduction. Our findings regarding **RQ$_1$** reveal that code smells are frequently introduced during file modifications, underscoring the importance of prioritizing quality assurance efforts that focus on monitoring and managing code changes during the maintenance phases of software development. Integrating ML-CS detectors into Continuous Integration and Continuous Delivery (CI/CD) pipelines could be a proactive solution to mitigate these risks effectively. Such integration would help identify and address code smells early, preventing their inadvertent introduction. The presence of code smells not only degrades the maintainability of the codebase—such as diminished readability in the case of *'Merge API Parameter Not Explicitly Set'* smells—but can also have far-reaching implications on software performance and reliability. For instance, certain smells like *'Chain Indexing'* can adversely affect system performance, while others can significantly increase the likelihood of defects. These defects, in turn, may lead to extended development cycles, delayed time-to-market, and escalating costs, all of which pose significant risks to the project's success.

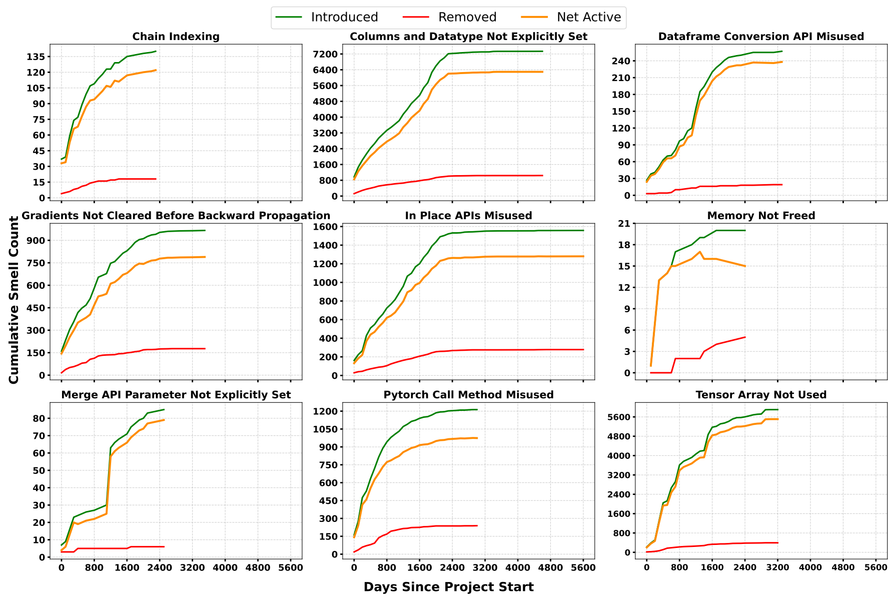


Fig. 6: Cumulative trends of ML-CSs over the project lifetime. The figure presents the cumulative count of introduced (green), removed (red), and net active (orange) ML-specific code smells across projects over time. The code smells considered are: *'Chain Indexing'* (CIDX), *'Columns and DataType Not Explicitly Set'* (CDE), *'Gradients Not Cleared Before Backward Propagation'* (GNC), *'In-Place APIs Misused'*(IPA), *'Matrix Multiplication API Misused'* (MMA), textsl'Memory Not Freed' (MNF), *'Merge API Parameter Not Explicitly Set'* (MAP), *'NaN Equivalence Comparison Misused'* (NAN), *'PyTorch Call Method Misused'* (PC), *'TensorArray Not Used'* (TA), and *'Unnecessary Iteration'* (UI).

Take-Away Message

Frequent introduction of code smells during maintenance highlights the need for robust quality assurance practices. By integrating ML-specific smell detectors into CI/CD pipelines, the community can enhance code maintainability, boost system performance, and mitigate risks, ultimately supporting more reliable and efficient software development.

Moreover, the compounded effect of poor code quality can result in competitive disadvantages. Lengthened development timelines and increased costs could lead to being outperformed by competitors, potentially endangering the sustainability of the software development community or even risking its dissolution. Proactively addressing ML-specific code smells is not merely a technical necessity but a strategic imperative for sustaining software quality, meeting market demands, and ensuring long-term project viability. Our findings from **RQ**$_2$ indicate that smell emergence is closely tied to highly frequent activities such as new feature development, system enhancements, and maintenance. Such a frequency highlights the need for tailored interventions beyond reactive detection and mitigation of smells. Integrating ML-specific smell detection tools, e.g., CodeSmile, into CI/CD pipelines may enable fast identification of code smells during development. At the same time, integrating these tools into IDEs can offer developers immediate, real-time feedback during implementation.

Moreover, organizations should foster a culture of preventive quality assurance by equipping developers with targeted training on smell-prone tasks. This training should explain the nature of ML-CSs and provide actionable strategies for avoiding them during common ML pipeline activities. This is also potentially interesting for researchers working on source code quality assessment, who may contribute by devising and recommending guidelines and instruments that may help avoid introducing code smells in the first place.

Besides education, integrating ML-specific considerations into regular code reviews can effectively complement automated detection. These considerations should prioritize critical parts of the pipeline, such as data preprocessing and model evaluation scripts, where the introduction of smells is more likely to have cascading effects on system performance and reliability. Unlike standard code reviews, these sessions should be designed to specifically address the unique demands of ML-enabled systems, making them more effective in mitigating smell-related risks. This represents a challenge for researchers in the field of code review, who might be interested in devising novel reading techniques and manual analysis procedures that may better support developers when assessing the quality of ML-enabled systems.

Furthermore, organizations can plan smell-specific quality checkpoints alongside major project milestones to sustain these efforts over time. For instance, as part of post-feature integration or pre-release evaluations, focused reviews of components with a history of smell introduction could prevent issues from persisting in the codebase. These checkpoints and regular refactoring sessions can ensure that smells introduced during earlier phases do not accumulate technical debt or compromise system maintainability.

Take-Away Message

Proactively addressing code smells is not just a technical necessity but a strategic imperative to ensure long-term viability and market competitiveness. With smells frequently emerging during tasks like new features, enhancements, and maintenance, organizations must promote best practices for feature implementation and conduct regular code reviews integrating ML-specific considerations to mitigate these risks effectively.

On the Smell Removal. Our findings indicate that ML-CSs are more frequently observed to disappear during stable maintenance periods, which may reflect a natural tendency for code modifications—such as refactoring or clean-up activities—to occur outside of time-critical phases like intensive feature development or bug-fixing sprints. This behavior indicates an opportunity to optimize the design of smell detection tools to account for the temporal dynamics of software development. For instance, CI pipelines could be configured to adaptively skip smell detection checks during periods of high commit activity, reducing overhead and allowing developers to focus on more urgent tasks. Our findings suggest that a *time-aware* smell detection may minimize the risk of disrupting productivity while ensuring that code quality is still prioritized during less critical phases.

Take-Away Message

Most ML-specific code smells are resolved during stable maintenance periods, suggesting the need for time-aware detection tools that adapt to development intensity. These may optimize resource allocation, ensuring smells are addressed without hindering progress during high-intensity phases.

Our results suggest that the removal of ML-specific code smells (ML-CSs) often occurs opportunistically during routine development tasks, reflecting the concept of *"floss refactoring"*—incremental code improvements made alongside other primary activities such as feature implementation or bug fixing [19]. This behavior highlights the importance of lightweight, seamlessly integrated quality assurance tools that align with developers' natural workflows.

To support this, future tools should embed actionable, context-aware feedback directly into development environments (e.g., IDEs), minimizing disruption. Tailoring recommendations to the specific libraries or stages of the ML pipeline can further enhance relevance. Additionally, tools could be designed to recognize when developers are engaged in adjacent refactoring or cleanup tasks, offering timely suggestions for addressing ML-CSs. Such human-centric design can make quality assurance more intuitive and effective in practice.

The integration of smell removal into routine development activities also reflects the pragmatic nature of developer behavior. Addressing ML-CSs alongside other tasks reduces the overhead of separate refactoring efforts and ensures that smells are addressed as part of ongoing development. This highlights the need for tools that can efficiently identify and assist in resolving smells in a way that complements the developer's natural workflow. This observation recalls the need for further *human-centric* adjustments in quality assurance tools. Such adjustments could enable these tools to provide recommendations not only based on the time of development activities, but also when individual developers feel the need to improve code quality. For example, incorporating mechanisms that detect when developers are refactoring adjacent code or engaging in cleanup tasks could trigger contextually relevant suggestions for addressing ML-CSs. This may represent a way to make quality assurance more intuitive and less intrusive within the development process.

Take-Away Message

The removal of ML-specific code smells is frequently embedded in routine development activities, confirming the need for lightweight and seamless tools that facilitate incremental quality improvements. This supports the broader trend of floss-refactoring, where developers opportunistically address quality issues without disrupting their primary tasks.

On the Smell Survivability. The findings of **RQ**$_4$ suggest that certain ML-specific code smells, such as *'In-Place APIs Misused'* and *'TensorArray Not Used'*, are highly persistent in ML-enabled systems. These smells likely significantly negatively impact defect proneness and model efficiency, highlighting the critical need to prioritize code smells based on their potential impact while considering the project scope and goals. For example, in systems where ML components play a critical role, e.g., healthcare systems, addressing smells related to defect proneness and efficiency can not only enhance the reliability of the software but also improve its ability to make rapid and accurate decisions. Additionally, our findings emphasize the importance of the Software Quality Assurance for Artificial Intelligence (SQA4AI) community developing automated refactoring solutions to address ML-specific code smells. Such tools should be seamlessly integrated into development workflows, such as IDE plugins capable of automatically refactoring source code to mitigate ML-CSs. These solutions could leverage trained Large Language Models (LLMs), incorporate tailored pre-existing smell removal strategies, or utilize a catalog of example-based smell mitigation techniques. Furthermore, the cumulative trend analysis supports the key insight that while some ML-CSs are removed, the overall number of active smells continues to increase across projects. This suggests that current maintenance activities do not proportionally offset smell introductions, leading to sustained smell survivability over time. These findings indicate that existing quality assurance practices may not be sufficient to systematically manage ML-CSs, reinforcing the need for proactive detection and mitigation strategies. Addressing this issue could help prevent long-term technical debt accumulation in ML-based systems, ensuring maintainability and performance over extended project lifetimes.

Take-Away Message

Prioritizing the mitigation of high-impact smells is essential for enhancing system reliability and decision-making capabilities. The SQA4AI community must focus on developing automated refactoring tools that can be integrated into workflows.

When Code Smells Meet ML. Broadening the scope of the discussion, we observe that our findings differ from those reported in studies of traditional code smells. Our analysis, based on tasks involving ML-CSs, revealed that a significant proportion of refactoring activities are associated with the introduction of ML-CSs. This finding aligns with observations made by Tufano et al. [34], who reported that up to 11% of the instances of traditional code smells are introduced during refactoring activities. This overlap suggests that the introduction of ML-CSs is not solely the result of suboptimal coding practices but can also emerge during efforts to improve or modify code.

Additionally, the shorter lifespan of ML-CSs, as highlighted in the analysis for **RQ**$_4$, raises interesting questions about their role in the overall quality of ML systems. Unlike traditional code smells, which often persist until explicitly addressed, ML-CSs may be resolved incidentally through unrelated changes.

However, this transient nature does not diminish their impact; unresolved ML-CSs, even if short-lived, can contribute to technical debt, particularly in rapidly evolving ML projects where iterative experimentation is key. In conclusion, while ML-CSs share some characteristics with traditional code smells, their domain-specific origins, dynamic nature, and unique patterns of persistence and resolution highlight the need for specialized approaches to address them.

Take-Away Message

ML-CSs represent a distinct class of quality issues that require tailored detection, management, and resolution approaches. By understanding their unique characteristics and incorporating adaptive practices, developers can mitigate their impact and maintain the quality and sustainability of ML systems in the face of evolving requirements and iterative workflows.

6 Threats to Validity

This section discusses possible threats to validity that could impact our results and the strategies we adopted to mitigate them.

Threats to Construct Validity A potential threat to validity concerns the detection of ML-CSs. Specifically, a rule-based detection approach may lead to false positives and negatives. To mitigate this threat, we adopted a pattern-matching strategy leveraging an Abstract Syntax Tree (AST) designed to reflect the definitions provided by Zhang et al. [41]. We opted for deterministic static analysis as a preliminary approach to detecting ML-CSs. While more advanced, state-of-the-art technologies (*e.g.*, LLM-based detection methods) might offer more optimal results in some cases, their inherent indeterminism can lead to a lack of control over the detection process. By contrast, our static analysis approach, grounded in deterministic and rule-based conditions, provides a simple, explainable, and robust method for identifying ML-CSs. To enhance the reliability of our approach, we manually validated the tool's accuracy using a statistically significant sample of detected ML-CS instances. While this method limits detection to the set of smells explicitly defined in the literature, it represents an essential starting point for creating a comprehensive quality assessment tool for ML-enabled systems.

Another threat to construct validity concerns potential refactoring operations that move a smell from one file to another. As currently implemented, CodeSmile does not track the evolution of individual smell instances across commits. Consequently, when a code smell is relocated—e.g., due to refactoring—the detector interprets it as two separate events: a smell removal in the source file and a smell introduction in the target file. This design choice may lead to overestimating the frequency of smell introductions and removals, potentially skewing some of the derived metrics. It is important to note that this limitation is not unique to our approach. Accurately tracking code artifacts over time remains a known and non-trivial challenge in the field of software repository mining. Prior studies—for example, Tufano et al. [35], Palomba et al. [22], and Ratzinger et al. [24]—have similarly acknowledged how code movements, renamings, and refactorings can complicate the analysis of long-term code quality patterns, including the persistence and evolution of code smells or defects.

Another potential threat relates to data collection, specifically the risk of mismatches between the collected data and the properties of the projects under analysis, including challenges such as incomplete or inconsistent project histories, variations in repository structures, and the potential presence of noise in the extracted data. To address this concern, we employed PyDriller, a widely-used tool for mining repositories, to ensure a systematic and reliable data collection process. Its application in several prior studies [11,28,27] validates its reliability and accuracy, further reinforcing our choice. By relying on a tool with a proven track record, we reduced the likelihood of errors introduced during data extraction, improving the data collection procedures.

Threats to Internal Validity In the context of survivability analysis ($\mathbf{RQ}_4$), we excluded the smells developers could not fix because of lack of time. In other words, we removed smell-introducing commits from our analysis that were too close to the last commit of the project by excluding those instances whose smell-introducing date summed to the median removal time is beyond the end of the commit history. Another threat regards smell-removing commits. We considered a smell removed at commit c_i when the instance is detected at commit c_{i-1} but no longer detectable at commit c_i. While this approach provides a practical means of tracking smell removal, it may introduce some imprecision. Specifically, refactoring activities might not directly remove the ML-CS in commit c_i, but instead modify the source code over multiple commits until it no longer matches the detection rules. This potential imprecision could also stem from limitations in the detection tool itself. Although our evaluation indicates that the tool is sufficiently sound, there remains a possibility that some removals were the result of false positives detected by CodeSmile.

Threats to External Validity The main threat to the generalizability of the results regards the dataset we used. We are conscious that the project selection is a critical experiment component. Therefore, we relied on the NICHE dataset [39], *i.e.*, a large dataset that contains only real ML-enabled systems. We analyzed 337 projects and over 400k commits, proposing a large empirical study. The projects are provided from different contexts and have different characteristics (*e.g.*, size, number of files). We know the results could not directly apply to industrial projects; however, we invite researchers to replicate our study on closed-source projects to identify differences and common points.

Another generalizability threat is related to the programming language used to write the systems under analysis *i.e.*, Python. We are aware that due to the specific characteristics of this programming language, the generalizability of our results needs to be confirmed by future studies using other programming languages. We intend to conduct similar investigations for other programming languages as part of our future agenda to confirm the findings.

Threats to Conclusion Validity The main threat to conclusion validity is related to the statistical tests applied to address $\mathbf{RQ}_0$, specifically the Friedman, the Wilcoxon test, and Cliff's Delta, as violations of the assumptions underlying these tests could affect the validity of the results. To mitigate this threat, we assessed the distribution of the data to verify normality before selecting the most appropriate test for each analysis. Nevertheless, there remains a possibility that subtle deviations in data characteristics could influence the outcomes of the tests. Another threat involves the computation of lifespan in $\mathbf{RQ}_4$, where we used the number of commits and days as time indicators. While these measures provide a practical way to estimate lifespan, they may not be entirely precise. Variations in project practices, such as irregular commit activity due to internal policies or external factors (e.g., holidays or sprints), could skew the analysis and introduce inconsistencies. For instance, some projects may commit less frequently over extended periods, leading to an underestimation of lifespan, while others may commit in bursts, potentially inflating the indicator. The large-scale nature of the study helped mitigate this potential threat; however, future replications in different contexts would further enhance confidence in our conclusions.

7 Conclusion

This study explores ML-specific code smells (ML-CSs), detailing their lifecycle, prevalence, and impact in ML-enabled systems. By analyzing over 400,000 commits across 337 projects, we provide insights into how ML-CSs are introduced, evolve, and are resolved, ultimately improving software quality. We developed CodeSmile, a static analysis tool with 87% precision and 78% recall, specifically for detecting ML-CSs. Our findings reveal that ML-CSs often arise during maintenance and new feature development, sometimes inadvertently during refactoring. Most are resolved during stable maintenance periods, highlighting the need for proactive quality assurance throughout the software development lifecycle. Future research will expand the capabilities of CodeSmile to detect more ML-CSs and understand developers' perceptions of their severity and impact on maintainability, performance, and reliability.

Moreover, another promising direction for future research is understanding developer intent and awareness when addressing ML-CSs. We found a few instances where developers referred to removing ML-CSs. For example, in the project `geyang/ml_logger` [7], one commit message mentions correcting the misuse of the `sort_values` API, corresponding to the *'In-Place APIs Misused'* smell. This suggests that developers are, in some cases, aware of and intentionally address such issues. Moreover, while simple semantic inspection

[7] Example of a smell-removing commit aware of the ML-CSs: `https://github.com/geyang/ml-logger/commit/25aff14cf101ac4db06be02e65d845c54fc38c84`

helped link certain commit messages to specific smells, many other cases would require deeper analysis. Future work will explore the use of semantic analysis and Large Language Models (LLMs) to infer developer intent and provide richer insights into how and why ML-CSs are addressed during development.

We also aim to utilize advanced technologies, including LLMs, to create automated strategies for refactoring ML-CS-affected code, enhancing the robustness of ML-enabled systems.

Acknowledgements

This work has been partially supported by the European Union - NextGenerationEU through the Italian Ministry of University and Research, Projects PRIN 2022 "QualAI: Continuous Quality Improvement of AI-based Systems" (grant n. 2022B3BP5S , CUP: H53D23003510006) and by the EMELIOT national project funded by the MUR under the PRIN 2020 program (Contract 2020W3A5FY).

Declaration of Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data Availability Statement

The manuscript includes data as electronic supplementary material. In particular, datasets generated and analyzed during the current study, detailed results, scripts, and additional resources useful for reproducing the study are available as part of our online appendix on Figshare [25]. In addition, we included the GitHub repository link for CodeSmile: `https://github.com/giammariagiordano/smell_ai/tree/main`.

Credits

Gilberto Recupito: Formal analysis, Investigation, Data Curation, Validation, Writing - Original Draft, Visualization. **Giammaria Giordano**: Formal analysis, Investigation, Data Curation, Validation, Writing - Original Draft, Visualization. **Filomena Ferrucci**: Writing - Review & Editing. **Dario Di Nucci**: Supervision, Writing - Review & Editing. **Fabio Palomba**: Supervision, Resources, Writing - Review & Editing.

References

1. Azeem, M.I., Palomba, F., Shi, L., Wang, Q.: Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. Information and Software Technology **108**, 115–138 (2019)
2. Basili, V.R., Caldiera, G., Rombach, H.D.: The goal question metric approach. Encyclopedia of software engineering pp. 528–532 (1994)
3. Bessghaier, N., Ouni, A., Mkaouer, M.W.: On the diffusion and impact of code smells in web applications. In: Q. Wang, Y. Xia, S. Seshadri, L.J. Zhang (eds.) Services Computing – SCC 2020, pp. 67–84. Springer International Publishing, Cham (2020)
4. Cardozo, N., Dusparic, I., Cabrera, C.: Prevalence of code smells in reinforcement learning projects (2023)
5. Cliff, N.: Dominance statistics: Ordinal analyses to answer ordinal questions. Psychological Bulletin **114**, 494–509 (1993). URL `https://api.semanticscholar.org/CorpusID:120113824`
6. Conover, W.J.: Practical nonparametric statistics, vol. 350. john wiley & sons (1999)
7. Costal, D., Gómez, C., Martínez-Fernández, S.: Metrics for code smells of ml pipelines. In: R. Kadgien, A. Jedlitschka, A. Janes, V. Lenarduzzi, X. Li (eds.) Product-Focused Software Process Improvement, pp. 3–9. Springer Nature Switzerland, Cham (2024)
8. Cunningham, W.: The wycash portfolio management system. ACM Sigplan Oops Messenger **4**(2), 29–30 (1992)
9. Fowler, M.: Refactoring. Addison-Wesley Professional (2018)
10. Fowler, M., Beck, K.: Refactoring: Improving the design of existing code. In: 11th European Conference. Jyväskylä, Finland (1997)
11. Giordano, G., Annunziata, G., De Lucia, A., Palomba, F., et al.: Understanding developer practices and code smells diffusion in ai-enabled software: A preliminary study. In: IWSM-Mensura (2023)
12. Giordano, G., Fasulo, A., Catolino, G., Palomba, F., Ferrucci, F., Gravino, C.: On the evolution of inheritance and delegation mechanisms and their impact on code quality. In: 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 947–958. IEEE (2022)
13. Giordano, G., Fasulo, A., Catolino, G., Palomba, F., Ferrucci, F., Gravino, C.: On the evolution of inheritance and delegation mechanisms and their impact on code quality. In: 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 947–958 (2022)
14. Giordano, G., Sellitto, G., Sepe, A., Palomba, F., Ferrucci, F.: The yin and yang of software quality: On the relationship between design patterns and code smells. In: 2023 49th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pp. 227–234. IEEE (2023)
15. Khomh, F., Penta, M.D., Guéhéneuc, Y.G., Antoniol, G.: An exploratory study of the impact of antipatterns on class change-and fault-proneness. Empirical Software Engineering **17**, 243–275 (2012)
16. Lehman, M.M., Ramil, J.F., Wernick, P.D., Perry, D.E., Turski, W.M.: Metrics and laws of software evolution-the nineties view. In: Proceedings Fourth International Software Metrics Symposium, pp. 20–32. IEEE (1997)
17. Lenarduzzi, V., Lomio, F., Moreschini, S., Taibi, D., Tamburri, D.A.: Software quality for ai: Where we are now? In: Software Quality: Future Perspectives on Software Engineering Quality: 13th International Conference, SWQD 2021, Vienna, Austria, January 19–21, 2021, Proceedings 13, pp. 43–53. Springer (2021)
18. Martínez-Fernández, S., Bogner, J., Franch, X., Oriol, M., Siebert, J., Trendowicz, A., Vollmer, A.M., Wagner, S.: Software engineering for ai-based systems: a survey. ACM Transactions on Software Engineering and Methodology (TOSEM) **31**(2), 1–59 (2022)
19. Murphy-Hill, E., Black, A.P.: Why don't people use refactoring tools? In: Proceedings of the 1st Workshop on Refactoring Tools, pp. 61–62 (2007)
20. Palomba, F., Bavota, G., Di Penta, M., Fasano, F., Oliveto, R., De Lucia, A.: On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. In: Proceedings of the 40th International Conference on Software Engineering, pp. 482–482 (2018)

21. Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., De Lucia, A.: Do they really smell bad? a study on developers' perception of bad code smells. In: 2014 IEEE International Conference on Software Maintenance and Evolution, pp. 101–110. IEEE (2014)
22. Palomba, F., Panichella, A., Zaidman, A., Oliveto, R., De Lucia, A.: The scent of a smell: An extensive comparison between textual and structural smells. IEEE Transactions on Software Engineering **44**(10), 977–1000 (2018). DOI 10.1109/TSE.2017.2752171
23. de Paulo Sobrinho, E.V., De Lucia, A., de Almeida Maia, M.: A systematic literature review on bad smells–5 w's: which, when, what, who, where. IEEE Transactions on Software Engineering **47**(1), 17–66 (2018)
24. Ratzinger, J., Sigmund, T., Gall, H.C.: On the relation of refactorings and software defect prediction. In: Proceedings of the 2008 international working conference on Mining software repositories, pp. 35–38 (2008)
25. Recupito, G., Giordano, G., Ferrucci, F., Di Nucci, D., Palomba, F.: When code smells meet ml: On the lifecycle of ml-specific code smells in ml-enabled systems - appendix (2024). DOI 10.6084/m9.figshare.28167065. URL `https://doi.org/10.6084/m9.figshare.28167065`
26. Recupito, G., Pecorelli, F., Catolino, G., Lenarduzzi, V., Taibi, D., Di Nucci, D., Palomba, F.: Technical debt in ai-enabled systems: On the prevalence, severity, impact, and management strategies for code and architecture. Journal of Systems and Software **216**, 112151 (2024)
27. Rhmann, W.: Quantitative software change prediction in open source web projects using time series forecasting. International Journal of Open Source Software and Processes (IJOSSP) **12**(2), 36–51 (2021)
28. Riquet, N., Devroey, X., Vanderose, B.: Gitdelver enterprise dataset (gded) an industrial closed-source dataset for socio-technical research. In: Proceedings of the 19th International Conference on Mining Software Repositories, pp. 403–407 (2022)
29. Rosner, B., Glynn, R.J.: Power and sample size estimation for the wilcoxon rank sum test with application to comparisons of c statistics from alternative prediction models. Biometrics **65**(1), 188–197 (2009). DOI 10.1111/j.1541-0420.2008.01062.x. URL `https://doi.org/10.1111/j.1541-0420.2008.01062.x`
30. Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M., Crespo, J.F., Dennison, D.: Hidden technical debt in machine learning systems. Advances in neural information processing systems **28** (2015)
31. Spadini, D., Aniche, M., Bacchelli, A.: Pydriller: Python framework for mining software repositories. In: Proceedings of the 2018 26th ACM Joint meeting on european software engineering conference and symposium on the foundations of software engineering, pp. 908–911 (2018)
32. Taibi, D., Janes, A., Lenarduzzi, V.: How developers perceive smells in source code: A replicated study. Information and Software Technology **92**, 223–235 (2017)
33. Tang, Y., Khatchadourian, R., Bagherzadeh, M., Singh, R., Stewart, A., Raja, A.: An empirical study of refactorings and technical debt in machine learning systems. In: 2021 IEEE/ACM 43rd international conference on software engineering (ICSE), pp. 238–250. IEEE (2021)
34. Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Di Penta, M., De Lucia, A., Poshyvanyk, D.: When and why your code starts to smell bad (and whether the smells go away). IEEE Transactions on Software Engineering **43**(11), 1063–1088 (2017)
35. Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Di Penta, M., De Lucia, A., Poshyvanyk, D.: When and why your code starts to smell bad (and whether the smells go away). IEEE Transactions on Software Engineering **43**(11), 1063–1088 (2017)
36. Van Oort, B., Cruz, L., Aniche, M., Van Deursen, A.: The prevalence of code smells in machine learning projects. In: 2021 IEEE/ACM 1st Workshop on AI Engineering-Software Engineering for AI (WAIN), pp. 1–8. IEEE (2021)
37. Walter, B., Alkhaeir, T.: The relationship between design patterns and code smells: An exploratory study. Information and Software Technology **74**, 127–142 (2016)
38. Wang, G., Wang, Z., Chen, J., Chen, X., Yan, M.: An empirical study on numerical bugs in deep learning programs. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, pp. 1–5 (2022)

39. Widyasari, R., Yang, Z., Thung, F., Qin Sim, S., Wee, F., Lok, C., Phan, J., Qi, H., Tan, C., Tay, Q., Lo, D.: Niche: A curated dataset of engineered machine learning projects in python. In: 2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR), pp. 62–66 (2023)
40. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A.: Experimentation in software engineering. Springer Science & Business Media (2012)
41. Zhang, H., Cruz, L., Van Deursen, A.: Code smells for machine learning applications. In: Proceedings of the 1st International Conference on AI Engineering: Software Engineering for AI, pp. 217–228 (2022)
42. Zhou, Y., Leung, H., Xu, B.: Examining the potentially confounding effect of class size on the associations between object-oriented metrics and change-proneness. IEEE Transactions on Software Engineering **35**(5), 607–623 (2009)