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

Gilberto Recupito
Sesa Lab - University of Salerno
Salerno, Italy
grecupito@unisa.it

Giammaria Giordano
Sesa Lab - University of Salerno
Salerno, Italy
giagiordano@unisa.it

Filomena Ferrucci
Sesa Lab - University of Salerno
Salerno, Italy
fferrucci@unisa.it

Dario Di Nucci
Sesa Lab - University of Salerno
Salerno, Italy
ddinucci@unisa.it

Fabio Palomba
Sesa Lab - University of Salerno
Salerno, Italy
fpalomba@unisa.it

## ABSTRACT

**Context.** The adoption of Machine Learning (ML)–enabled systems is steadily increasing. Nevertheless, there is a shortage of ML-specific quality assurance approaches, possibly because of the limited knowledge of how quality-related concerns emerge and evolve in ML-enabled systems. **Objective.** We aim to investigate the emergence and evolution of specific types of quality-related concerns known as ML-specific code smells, *i.e.*, sub-optimal implementation solutions applied on ML pipelines that may significantly decrease both quality and maintainability of ML-enabled systems. More specifically, we present a plan to study ML-specific code smells by empirically analyzing (i) their prevalence in real ML-enabled systems, (ii) how they are introduced and removed, and (iii) their survivability. **Method.** We will conduct an exploratory study, mining a large dataset of ML-enabled systems and analyzing over 400k commits about 337 projects. We will track and inspect the introduction and evolution of ML smells through CodeSmile, a novel ML smell detector that we will build to enable our investigation and to detect ML-specific code smells.

## CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**.

## KEYWORDS

Technical Debt; ML-Specific Code Smells; Software Quality Assurance; Software Engineering for Artificial Intelligence.

**ACM Reference Format:**
Gilberto Recupito, Giammaria Giordano, Filomena Ferrucci, Dario Di Nucci, and Fabio Palomba. 2023. When Code Smells Meet ML: On the Lifecycle of ML-specific Code Smells in ML-enabled Systems. In *MSR 2024: 21st International Conference on Mining Software Repositories, April 15–16, 2024, Lisbon, ES.* ACM, New York, NY, USA, 8 pages. https://doi.org/XXXXXXX.XXXXXXX


*MSR 2024, April 2024, Lisbon, Portugal*

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

## 1 INTRODUCTION

Machine Learning (ML) evolved through the emergence of complex software integrating ML modules, defined as ML-enabled systems [13]. 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.

However, 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 [7] *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* is a manifestation of technical debt. They are *symptoms* of poor design and implementation choices that, if left unaddressed, can deteriorate the overall quality of the system [8].

Sculley et al. [19] showed that ML-enabled systems are incredibly prone to technical debt and code smells, raising the need for a quality assurance process for ML components. Cardozo et al. [3] and Van Oort et al. [24] 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. This lack of quality management assets stimulates the proliferation of code smells in ML-enabled systems [12]. 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. [29] defined a new form of code smells, *AI-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.* A key example of those quality issues is using a loop operation instead of exploiting the corresponding Pandas function for data handling, leading to *Unnecessary Iteration* smell [29].

While some work underlines the need to explore AIML-CSs [6, 29], there is still a lack of knowledge on this type of quality issues. Among the various possible causes, we outline a lack of knowledge on how ML-CSs emerge and evolve and what motivations lead developers to introduce and remove them. This poor knowledge significantly threatens the release of ML-CSs detectors aimed at improving the system's quality. Researchers and practitioners cannot define crucial aspects of smell detection and refactoring, such as (i) the conditions where ML-CSs are more prone to be introduced and

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

removed, (ii) in which stage of the development lifecycle practitioners should pay attention *e.g.*, when they introduce new features or when they fix defects, (iii) what are the practices that developers use to remove code smells, (iv) in which stage a quality assurance monitoring tool should focus on tracking the evolution of ML-CSs *e.g.*, data preparation or model training.

In this registered report, we aim to bridge this knowledge gap by describing our plan to understand the evolution of ML-CSs in ML-enabled systems. Specifically, we will perform a large-scale mixed *confirmatory* and *exploratory* study considering 337 projects coming from the NICHE dataset [27] and will mine over 400k commits to analyze (i) the prevalence of ML-CSs in ML-enabled systems, (ii) when and why ML-CSs are introduced and removed, and (iii) how ML-CSs survive over time. All the collected data, analysis scripts, and additional material will be publicly available online.

On the one hand, we believe that an improved understanding of these evolutionary aspects may provide *researchers and tool vendors* with insights that might be useful to characterize the peculiarities of ML-CSs and the way practitioners currently deal with them, possibly leading to the definition of novel quality assurance mechanisms that better fit the typical lifecycle of ML-CSs, thus better-assisting developers daily. On the other hand, *practitioners* with findings that may be used to improve the quality of ML-enabled systems by removing ML-CSs through the mechanisms employed by other practitioners and that will be described as part of our work.

## 2 BACKGROUND AND RELATED WORK

This section provides an overview of ML-CSs and summarizes the state-of-the-art concerning how code smells have been investigated in traditional and ML-enabled systems.

```
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 1: Example of Gradients Not Cleared before Backward Propagation in the Transformer project.**

### 2.1 Background

Zhang et al. [29] recently released a catalog of 22 ML-CSs by empirically analyzing white and grey literature. In our appendix, information on the list of ML-CSs identified by the authors, their description, the pipeline stage they affect, and the quality aspects they impact are presented [16].

To provide a tangible example of ML-CS, let consider *Gradients Not Cleared before Backward Propagation*. 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 [26]. To mitigate this smell, the function `optimizer.zero_grad()` should be used before the backpropagation step. Listing 1 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. [29].

### 2.2 Related Work

Several studies have been carried out in the context of code smells in traditional systems [11, 14, 15, 21] also investigating their impact during the software evolution [10, 25]. Tufano et al. [23] conducted a large empirical study on when and why code smells are introduced in traditional systems, their survivability, and how developers remove them. They discovered that code smells are mainly introduced when files are created, and only a negligible percentage of them are removed through refactoring operations. Their impactful contribution allowed for improving the management of traditional code smells through the implementation of automatic detection and refactoring tools. Our work, inspired by the contribution of Tufano et al., aims to explore the nature of ML-CSs and to improve the quality assurance process for ML-enabled systems.

In the remaining part of this section, we focus on state-of-the-art traditional code smells in ML-enabled systems because, to our knowledge, no studies explicitly focus on ML-CSs in the context of ML-enabled systems. Tang et al. [22] conducted an empirical study analyzing 26 machine learning (ML) projects. They identified several code anti-patterns ML-Specific, highlighting the high prevalence of *Duplicated Code*. Their findings shed light on unique challenges and debt types specific to ML projects, helping researchers and practitioners understand the nature of technical debt in ML projects. Van Oort et al. [24] conducted an empirical study on code smells by analyzing 74 open-source machine learning projects using PyLint. Also, in this case, they found that *Duplicated Code* is the most frequent smell. Furthermore, they noticed that the emergence of code smells is more frequent in machine-learning systems than in traditional ones. Inspired by the work of Van Oort et al. [24], Giordano et al. [9] performed a large study on the diffusion of code smells over time in ML-enabled systems, focused on the activities that lead developers to introduce code smells and the survival time. The findings suggested that the smell variation does not follow a specific pattern over time; their introductions are principally due to evolutionary activities, and code smells can survive even for several years. These previous findings were confirmed by Cardozo et al. [3], who, in 2023, investigated the presence of code smells by considering 29 reinforcement learning (RL) projects. Still, in this case, the results seem to go in the same direction, pointing out that the emergence of traditional code smells is more frequent in reinforcement learning projects than in traditional ones.

Compared to previous work, our study will not focus on traditional code smells but on ML-specific code smells. Our results could

[2] https://github.com/huggingface/transformers/blob/main/examples/research_projects/bertology/run_bertology.py

shed light on their prevalence, introduction, removal, and survivability for eliciting ways to prevent developers from introducing such smells and help remove them when already present.

## 3 RESEARCH METHOD

The following section presents the design of the study, highlighting the main goal and the relative research questions. We will follow the guidelines by Wohlin et al. [28] and the ACM/SIGSOFT Empirical Standards[3]; in particular, we will use the "General Standard", "Data Science", and "Repository Mining" guidelines.

### 3.1 Goal and Research Questions

The 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. To address our goal, we formulated the specific goal through the GQM approach [1].

**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 will follow to address our research goal by addressing a preliminary and three main research questions.

**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 [2, 14], possibly not motivating further research on the matter. On the other hand, we may identify the most common ML-specific code smells and in which stage of an ML pipeline they manifest themselves.

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

This research question allows the classification of the conditions and contexts that lead developers introduce ML-CSs, other than the understanding if ML-CSs are injected, *e.g.*, when the ML projects are created or during the evolution of the system. 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 the actions performed by developer which likely introduce ML-CSs.

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

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

**Table 1: 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 |

**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.

### 3.2 Dataset Description and Projects Selection

We will rely on the NICHE dataset [27] 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 with extensive commit histories *i.e.*, projects with over 100 stars on GitHub, with commits more recent than May $1^{st}$, 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 1 shows the descriptive statistics on the variables "Stars", "Commits", and "Lines of Code" provided in NICHE [27]. 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. [30], 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 $30^{th}$ percentile. This set consists of 167 projects, all containing less than 4,765 lines of code.

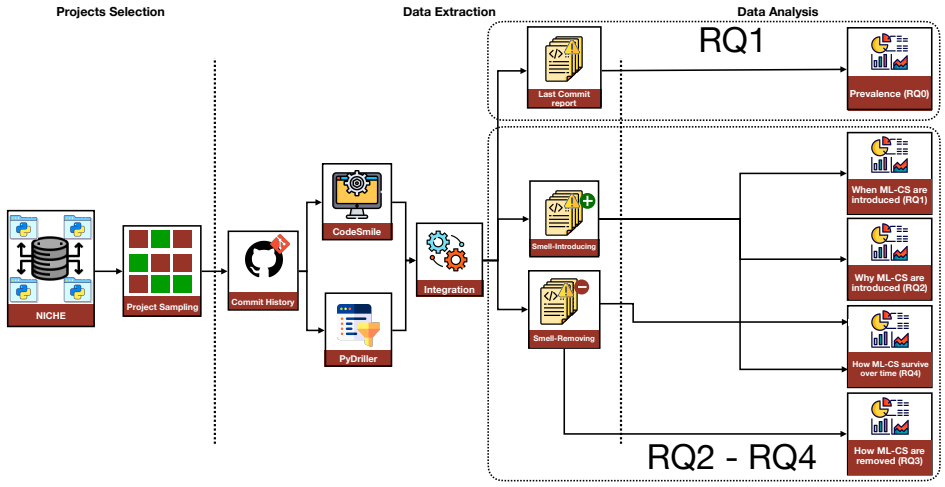


**Figure 1: The process designed for the study.**

**Table 2: 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 |

**Medium:** Projects with a number of lines of code above the $30^{th}$ percentile and below the $60^{th}$ 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 $60^{th}$ 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 will apply a statistically significant sampling for each group. Specifically, we will select the projects considering for each population, a sample with 95% confidence level and 5% margin of error. As a result, we will consider 117 projects, composed of 64,607 commits, for Small projects, 118 projects, composed of 71,380 commits, for Medium projects, and 142 projects, composed of 265,671 commits, for the Large projects *i.e.*, we want to analyze 337 projects and 401,658 commits. Table 2 shows the descriptive statistics for each size group.

In addition to analyzing projects based on their size in terms of LOC, we will also consider 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.

## 3.3 Data Extraction

After cloning the projects, we will gather fine-grained structural metrics using PyDriller, a framework helpful to analyze Git repositories [20]. We will extract the commit history of a project P belonging to the selected projects. For each commit, $C_i \in P$, we will collect the total number of files, the number of removed and added files, the commit date, and the commit message.

## 3.4 ML-Specific Code Smell Detection

To achieve the study's objectives, we will develop an ML-CS detection tool, CodeSmile. This tool will analyze the Abstract Syntax Tree (AST) of code to gather information about statements and methods. Notably, despite the clarity and specificity of ML-CS definitions provided by Zhang et al. [29], there is a lack of automated solutions for detecting these code smells. CodeSmile aims to fill this gap and provide an automatic solution to detect a set of 14 ML-CS; for the sake of space limitations, the definitions of the targeted ML-CS are reported as part of our online appendix [16]. The tool will use the rule-based conditions defined by Zhang et al. [29] to identify ML-CSs effectively, and we will manually validate it by selecting a statistically significant sampling of smells. The feasibility of static analysis to detect these smells is discussed in the technical report available in our online appendix [16]. The tool

will use the rule-based conditions defined by Zhang et al. [29] to identify ML-CSs effectively, and we will manually validate it by selecting a statistically significant sampling of smells.

The validation set will be defined as follows. Starting from the whole set of files included in the software projects considered in the study, we will first identify the files containing ML modules - in this way, we will filter out those files that, by definition, cannot contain any ML-CS. This first step will be performed by statically analyzing the content of each file: if it contains a reference to libraries such as Pandas,[4] TensorFlow,[5] Theano,[6] or PyTorch,[7] then the file will be marked as ML-related. The choice of the libraries to verify is mainly based on previous work [27]: the authors of the NICHE dataset have indeed defined ML projects as those relying on TensorFlow, Theano, or PyTorch. We also consider Pandas because several of the code smells targeted by our study refer to the suboptimal use of this library. We will then pick a statistically significant sample (confidence level = 95%, margin of error = 5%), which will finally form our validation set.

Upon completing the step above, we will then recruit external ML engineers, asking them to inspect the files in the validation sample (or a part thereof, in case the number of files to validate would be excessively large) and annotate the ML-CS instances they contain. In particular, for each ML engineer, we will prepare a validation package containing (1) the set of files to analyze, (2) a readme file reporting definitions and examples of the specific code smells we are interested in, and (3) a spreadsheet containing a number of rows equals to the files to validate and a number of columns equals to the smells to evaluate. Given this validation package, the ML engineers will be asked to fill each entry of the spreadsheet with a "Yes" if the *i-th* file is affected by the *j-th* code smell, with a "No" otherwise. We will give ML engineers up to 21 days to send us back the annotated spreadsheet. The number of ML engineers involved will depend on the validation sample size. We will start recruiting ML engineers from our contact network. Whenever needed, we will attempt to involve further practitioners through public calls on social media, practitioner's blogs, and specialized ML platforms. Among all the candidates, we will retain only those having at least three years of experience in the development of ML systems.

Once we receive the annotated spreadsheets, we will analyze them as follows. On the one hand, we will compute Cohen's k inter-rater agreement to measure the level of agreement among the ML engineers that assessed the smelliness of the same files - this will be useful to understand the extent to which ML code smells are perceived as such. On the other hand, we will define a ground truth by means of majority voting: for each file of the validation set, we will finally consider it as affected by a given smell if the majority of the ML engineers marked it as smelly. Such a ground truth will finally be used to assess the capabilities of CodeSmile in terms of precision and recall, hence assessing the amount of false positive and negative output by the tool.

As a result of the detection, the tool will report all the identified ML-CS with the relative position. CodeSmile will successively analyze each commit of ML projects to report all the information helpful in analyzing introduction, survival time, and removal of ML-CSs. To conduct our analyses, we will combine this data with those previously extracted using PyDriller [20].

[4]https://pandas.pydata.org/
[5]https://www.tensorflow.org/?hl=it
[6]https://pypi.org/project/Theano/
[7]https://pytorch.org/

## 3.5 Commit Data Extraction

After identifying the method for detecting ML-CS, we will extract the commit data for each project to address our research questions. First, we will identify the smell-introducing and smell-removing commits for each identified AI-CS instance. Specifically, we will track each smell $s_i$ identified in a commit $c_i$, using its file name and line number. We will analyze the project's history from the first commit, comparing $c_i$ and $c_i + 1$ pairwise. For each pair of consecutive commits, we will consider 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 will analyze the commit messages to understand the rationale behind introducing and removing ML-CSs.

## 3.6 Data Analysis

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

**RQ$_0$***: How are ML-specific code smells prevalent in ML-enabled systems?* CodeSmile will analyze last snapshot of the selected ML-enabled systems to observe the prevalence distribution of each ML-CS. Statistical descriptions and plots will be employed to understand the characteristics of each distribution. Then, insights on the most prevalent ML-CS across several projects will be provided. Such results will be further enriched by mapping each identified ML-CS to the related ML-pipeline stage, utilizing the mapping framework established by Zhang et al. [29]. This additional mapping step will enhance our understanding of ML-CSs, revealing the most prone areas within the ML development pipeline. Each analysis will be conducted considering the effect that related factors could have. Through the different size groups and the adoption of continuous integration defined in Section 3.2, we will apply 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 code smells may differ in terms of 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. Secondly, we hypothesized that the prevalence of ML code smells may depend on the size of the ML projects. Larger projects may indeed 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 hypothesized that projects relying on a Continuous Integration (CI) pipeline may have a lower prevalence of code smells than those not relying on that. Indeed, the presence of a CI pipeline may have a direct effect on 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.

For each null hypothesis, we also defined an alternative hypothesis. Regarding statistical verification, we plan to use different tests for the three hypotheses we formulated. For *H0* and *H2*, we will use the non-parametric Wilcoxon test [5], which investigates significant differences between two populations. Then, for the analysis for *H1* and given the goal of exploring differences between three groups, we will use a test that allows us to study differences across more than two populations: the non-parametric Friedman test.

The results will be statistically significant at $\alpha$=0.05. We will normalize the data distribution by the project LOC to avoid possible biases and quantify the effect size using the Cliff's Delta ($\delta$) [4].

**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 will collect all the smell-introducing commits to understand when each ML-CS is introduced. The outline of the smell-introducing commits will allow 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 will also implement a segmentation approach based on three key factors: development time, activity levels, and distance from the release. The first two segments will be used to examine the moment at which 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 will conduct an analysis 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 identify 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 3 provides the segments and the value that will be used for the segmentation.

**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 will analyze their messages to explain the rationale behind the changes. Specifically, we will leverage pattern matching, as previously done by Tufano et al. [23] to analyze why traditional code smells are introduced. In detail, we will extract the rationale, starting from the label set indicating the change operations described in Table 4. Finally, we will analyze to what extent commit rationales and introduced ML-CSs co-occur. We will analyze whether the smell-introducing commits acknowledge the presence of the ML-CS, resulting in self-admitted ML-CSs.

**Table 3: Segmentation of commits for the analysis of 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] |

**Table 4: 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. |

**RQ**$_3$: *When and how ML-specific code smells are removed in ML-enabled systems?* After analyzing the conditions and reasons for introducing ML-CSs, we will perform a similar analysis from the smell-removing commits. As for **RQ**$_1$, we will first verify whether ML-CSs are mitigated in a smell-removing commit and identify which are unmitigated, employing the same segmentation adopted and represented in Table 3. Afterward, we will focus on the smell-removing commits. We will collect the messages of all smell-removing commits using the pattern matching approach adopted in **RQ**$_2$, relying on the tags in Table 4 to extract the rationale behind the removal. This analysis will allow us to extract the refactoring operations addressing ML-CSs. From the set of the smell-removing commits analyzed, we will consider apart the commits that do not perform changes to ML-CSs but remove 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 will compute the number of commits from the smell-introducing commit to the smell-removing commit and the time span in days. Given such values, we will compute 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 will release all materials, including scripts and datasets, in an online appendix hosted in permanent storage.

## 4 THREATS TO VALIDITY

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

*Threats to Construct Validity.* A possible threat concerns the detection of the ML-CS instances. Indeed, a rule-based detection tool could lead to false positives and negatives. To limit this threat, we will implement a pattern-matching strategy using an AST and reflecting the definitions provided by Zhang et al. [29]. We will manually validate the tool's accuracy by selecting a statistically significant sampling of ML-CSs. While this solution limits the detection of the set of smells defined in the literature, this is a starting point for creating a quality assessment tool for ML-enabled systems.

Another threat regards data collection, particularly the mismatch between the data collected and the properties of the projects. To mitigate this threat, we will use an established tool to mine repositories *i.e.*, PyDriller, as already done in previous work [9, 17, 18].

*Threats to Internal Validity.* In the context of survivability analysis (**RQ**$_4$), we will exclude the smells developers could not fix because of lack of time. In other words, we will remove from our analysis smell-introducing commits 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 will consider a smell removed at commit $c_i$ when the instance is detected at commit $c_{i-1}$ but no longer detectable at commit $c_i$. This approach could lead to some imprecision due to refactoring, not removing the ML-CS in the commit $c_i$ but modifying the source code until it no longer matches the established detection rules.

*Threats to External Validity.* The main threat to the generalizability of the results regards the dataset we will use. We are conscious that the project selection is a critical experiment component. Therefore, we will rely on the NICHE dataset [27], *i.e.*, a large dataset that contains only real ML-enabled systems. We will analyze 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 validity is related to the statistical test that we want to apply to address the $RQ_0$ *i.e.*, the Friedman or the Wilcoxon test and Cliff's Delta, since the characteristics of the distribution of the data can violate the assumptions that need to be presented to conduct the tests. Before applying these tests, we will verify the distribution of the projects to verify the normality of the data, and only after this will we select the most appropriate test.

As the last conclusion validity threat, to calculate the lifespan in $RQ_4$, we want to use the number of commits and days as a time indicator. These two indicators could not be precise enough. Due to their internal policies, some communities may not commit even over long periods, thus making comparative analyses inaccurate.

## 5 CONCLUSION

This paper describes a plan to investigate AI-specific code smells. Through the analysis of 337 projects, we want to understand (i) the prevalence of AI-CSs, (ii) when and why they are introduced and removed, and (iii) their survival time. The implications of this study could be significant for the AI engineering community. On the one hand, we will provide CodeSmile a tool to detect AI-CS instances. On the other hand, the valuable insights from the large-scale empirical study we will perform will allow for eliciting ways to prevent developers from introducing such smells and help remove them when already present. Using the findings of this study, we aim to enhance the state of knowledge in the domain of AI quality assurance, directing future research endeavors towards the identification and resolution of quality issues specific for AI-enabled systems, moving towards the improving of the overall AI quality.

## ACKNOWLEDGMENT

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.

## REFERENCES

[1] Victor R Basili, Gianluigi Caldiera, and H Dieter Rombach. 1994. The goal question metric approach. *Encyclopedia of software engineering* (1994), 528–532.
[2] Narjes Bessghaier, Ali Ouni, and Mohamed Wiem Mkaouer. 2020. On the Diffusion and Impact of Code Smells in Web Applications. In *Services Computing – SCC 2020*, Qingyang Wang, Yunni Xia, Sangeetha Seshadri, and Liang-Jie Zhang (Eds.). Springer International Publishing, Cham, 67–84.
[3] Nicolás Cardozo, Ivana Dusparic, and Christian Cabrera. 2023. Prevalence of Code Smells in Reinforcement Learning Projects. arXiv:2303.10236 [cs.SE]
[4] Norman Cliff. 1993. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological Bulletin* 114 (1993), 494–509. https://api.semanticscholar.org/CorpusID:120113824
[5] William Jay Conover. 1999. *Practical nonparametric statistics*. Vol. 350. john wiley & sons.
[6] Dolors Costal, Cristina Gómez, and Silverio Martínez-Fernández. 2024. Metrics for Code Smells of ML Pipelines. In *Product-Focused Software Process Improvement*, Regine Kadgien, Andreas Jedlitschka, Andrea Janes, Valentina Lenarduzzi, and Xiaozhou Li (Eds.). Springer Nature Switzerland, Cham, 3–9.
[7] Ward Cunningham. 1992. The WyCash portfolio management system. *ACM Sigplan Oops Messenger* 4, 2 (1992), 29–30.
[8] Martin Fowler and Kent Beck. 1997. Refactoring: Improving the design of existing code. In *11th European Conference. Jyväskylä, Finland.*
[9] Giammaria Giordano, Giusy Annunziata, Andrea De Lucia, and Fabio Palomba. 2021. Understanding Developer Practices and Code Smells Diffusion in AI-Enabled Software: A Preliminary Study. (2021).
[10] Giammaria Giordano, Antonio Fasulo, Gemma Catolino, Fabio Palomba, Filomena Ferrucci, and Carmine Gravino. 2022. 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)*. 947–958.
[11] Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. 2012. An exploratory study of the impact of antipatterns on class change-and fault-proneness. *Empirical Software Engineering* 17 (2012), 243–275.
[12] Valentina Lenarduzzi, Francesco Lomio, Sergio Moreschini, Davide Taibi, and Damian Andrew Tamburri. 2021. 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*. Springer, 43–53.

[13] Silverio Martínez-Fernández, Justus Bogner, Xavier Franch, Marc Oriol, Julien Siebert, Adam Trendowicz, Anna Maria Vollmer, and Stefan Wagner. 2022. Software engineering for AI-based systems: a survey. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 2 (2022), 1–59.

[14] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. 2018. 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*. 482–482.

[15] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrea De Lucia. 2014. Do they really smell bad? a study on developers' perception of bad code smells. In *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 101–110.

[16] Gilberto Recupito, Giammaria Giordano, Filomena Ferrucci, Dario Di Nucci, and Fabio Palomba. 2024. When Code Smells Meet ML: On the Lifecycle of ML-specific Code Smells in ML-enabled Systems - Appendix. (2 2024). https://doi.org/10.6084/m9.figshare.25231817

[17] Wasiur Rhmann. 2021. Quantitative Software Change Prediction in Open Source Web Projects Using Time Series Forecasting. *International Journal of Open Source Software and Processes (IJOSSP)* 12, 2 (2021), 36–51.

[18] Nicolas Riquet, Xavier Devroey, and Benoît Vanderose. 2022. GitDelver enterprise dataset (GDED) an industrial closed-source dataset for socio-technical research. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 403–407.

[19] David Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. 2015. Hidden technical debt in machine learning systems. *Advances in neural information processing systems* 28 (2015).

[20] Davide Spadini, Maur'ıcio Aniche, and Alberto Bacchelli. 2018. 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*. 908–911.

[21] Davide Taibi, Andrea Janes, and Valentina Lenarduzzi. 2017. How developers perceive smells in source code: A replicated study. *Information and Software Technology* 92 (2017), 223–235.

[22] Yiming Tang, Raffi Khatchadourian, Mehdi Bagherzadeh, Rhia Singh, Ajani Stewart, and Anita Raja. 2021. An empirical study of refactorings and technical debt in machine learning systems. In *2021 IEEE/ACM 43rd international conference on software engineering (ICSE)*. IEEE, 238–250.

[23] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. 2017. When and why your code starts to smell bad (and whether the smells go away). *IEEE Transactions on Software Engineering* 43, 11 (2017), 1063–1088.

[24] Bart Van Oort, Luís Cruz, Maurício Aniche, and Arie Van Deursen. 2021. The prevalence of code smells in machine learning projects. In *2021 IEEE/ACM 1st Workshop on AI Engineering-Software Engineering for AI (WAIN)*. IEEE, 1–8.

[25] Bartosz Walter and Tarek Alkhaeir. 2016. The relationship between design patterns and code smells: An exploratory study. *Information and Software Technology* 74 (2016), 127–142.

[26] Gan Wang, Zan Wang, Junjie Chen, Xiang Chen, and Ming Yan. 2022. An Empirical Study on Numerical Bugs in Deep Learning Programs. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–5.

[27] Ratnadira Widyasari, Zhou Yang, Ferdian Thung, Sheng Qin Sim, Fiona Wee, Camellia Lok, Jack Phan, Haodi Qi, Constance Tan, Qijin Tay, and David Lo. 2023. NICHE: A Curated Dataset of Engineered Machine Learning Projects in Python. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. 62–66.

[28] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering*. Springer Science & Business Media.

[29] Haiyin Zhang, Luís Cruz, and Arie Van Deursen. 2022. Code smells for machine learning applications. In *Proceedings of the 1st International Conference on AI Engineering: Software Engineering for AI*. 217–228.

[30] Yuming Zhou, Hareton Leung, and Baowen Xu. 2009. 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 (2009), 607–623.