An Evidence-Based Study on the Relationship of Software Engineering Practices on Code Smells in Python ML Projects

Giammaria Giordano$^{1[0000-0003-2567-440X]}$, Antonio Della Porta$^{1[0000-0003-1860-8404]}$, Filomena Ferrucci$^{1[0000-0002-0975-8972]}$, and Fabio Palomba$^{1[0000-0001-9337-5116]}$

Software Engineering (SeSa) Lab — University of Salerno, Fisciano, Italy {giagiordano, adellaporta, fferrucci, fpalomba}@unisa.it

Abstract. The rapid adoption of Machine Learning (ML) technologies has introduced new challenges for code quality. Code smells, i.e., suboptimal design and implementation choices applied when developing source code, represent a particularly prevalent problem. While software engineering (SE) practices are often recommended to improve maintainability, their actual impact on code smells in ML projects remains unclear. In this paper, we present an evidence-based empirical study of 566 real-world Python ML projects from the NICHE dataset, labeled according to adherence to eight established SE practices. Using static analysis and statistical testing, we assess the relationship between these practices and the presence of ten Python-specific code smells. Our results show that projects adopting SE practices exhibit significantly fewer code smells. In particular, Continuous Integration is negatively correlated with the *Complex Container Comprehension* smell. These findings highlight the importance of engineering discipline in managing code quality in ML development.

Keywords: Quality Metrics · Software Maintenance Effort · Empirical Software Engineering.

1 Introduction

The adoption of Machine Learning (ML) technologies has grown rapidly across industries, enabling data-driven automation and decision-making. While this growth has led to major technological advancements, it has also introduced new challenges in software quality assurance. In particular, ML projects often exhibit maintainability and evolvability issues, due to a combination of fast-paced prototyping, experimentation-driven development, and limited use of mature software engineering (SE) practices. Among these issues, *code smells*, i.e., suboptimal design and implementation choices that complicate maintenance and evolution, are notably prevalent [9]. Sculley *et al.* [21] highlighted that ML systems are susceptible to technical debt, including code smells, due to their inherent complexity and lack of traditional software safeguards. This observation has since been echoed in studies emphasizing the scarcity of quality assurance tools tailored to the peculiarities of ML pipelines [14].

To help assess and guide the engineering rigor of software repositories, Munaiah *et al.* [17] introduced a framework consisting of eight SE practices, ranging from the use of Continuous Integration and Unit Testing to the presence of documentation and licensing information. This framework, designed to be agnostic of specific programming languages or application domains, serves as a proxy to characterize a project's engineering quality. Although originally proposed for general software, a natural yet underexplored question arises: *to what extent are these practices related to the emergence (or mitigation) of code smells in ML projects?*

In this paper, we apply *evidence-based* research to investigate the impact that these software engineering practices may have on the presence and distribution of code smells in real-world ML projects. Drawing from 566 real-world ML repositories in the NICHE dataset [28], we quantitatively analyze the relationship between the presence of code smells and adherence to the SE practices identified by Munaiah *et al.* We use the PySmell tool [4] to detect ten Python-specific code smells and investigate their distribution across three project size strata (*Small*, *Medium*, *Large*) defined by Lines of Code percentiles. Our analysis combines descriptive statistics, non-parametric tests, and correlation analysis to uncover statistically significant associations.

Our findings report that projects adhering to the SE practices in Munaiah's framework exhibit significantly fewer code smells. Among the practices, the adoption of Continuous Integration tools is negatively correlated with the presence of the *Complex Container Comprehension* smell. These results suggest actionable implications for both researchers and practitioners. For practitioners, especially ML developers, the evidence supports the adoption of SE practices as a way to mitigate code quality issues. For researchers, our study provides a validated methodology for operationalizing engineering quality in ML projects and opens new directions for quality-focused tooling in data-centric development workflows.

2 Related Work

This section summarizes the most relevant literature regarding code smell in traditional and ML Projects.

Over the years, numerous studies have examined code smells, primarily focusing on Java from various perspectives [25,13,8,30,2]. Tufano et al. [24] investigated the introduction and removal of code smells in Java projects, discovering that code smells are usually introduced during the initial stages of development and are often removed when files are deleted. Giordano *et al.* [12] explored the relationship between reusability mechanisms and code smells over time, finding a statistical relation between adopting reusability mechanisms and the reduction of code smell severity.

A study that bridges the gap between code smells in Java and other programming languages was conducted by Vavrová and Zaytsev [27], who statistically compared smells between Java and Python projects. Their findings revealed Python methods are generally longer than Java methods, and code smells related to sub-optimal use of classes are rarely detected. Giordano *et al.* [10] corroborated these results by examining ML projects from the NICHE dataset, noting the infrequent detection of code smells related to object-oriented practices.

Similarly, Tang *et al.* [23] inspected 26 ML projects written in Python, highlighting that *duplicated code* is one of the most frequent smells. Van Oort *et al.* [18] extended the previous study by examining 74 ML projects, reaching similar conclusions. Cardozo et al. [3] further confirmed these findings by investigating 29 Reinforcement Learning (RL) projects, emphasizing the emergence of code smells for these systems.

Our study complements the existing body of literature. While previous work analyzes the diffusion of code smells in ML projects, we statistically investigated whether the software engineering practices proposed by Munaiah [17] are related to code smells.

3 Study Design

The *goal* of this evidence-based study is to quantitatively assess the extent to which software engineering practices, as defined by the framework of Munaiah *et al.* [17], correlate with the presence and distribution of code smells in real-world Python-based ML projects. The *purpose* is to statistically investigate the correlation between such practices and the presence of code smells. The *quality focus* of this study is on the software engineering practices of Munaiah *et al.* as measurable indicators of project quality. The *perspective* is for both practitioners and researchers: From the practitioners' standpoint, particularly ML developers, the study provides actionable evidence on how adopting certain software engineering practices is related to structural code issues in ML projects. From the researchers, the study contributes to the empirical software engineering community by validating and operationalizing the framework of Munaiah *et al.* [17] in a large-scale setting.

Based on our goal, we formulated the following research questions:

🔍 **RQ$_1$.** *To what extent software engineering practices impact code smells distribution in ML projects?*

RQ$_1$ aims to analyze the relation from a statistical viewpoint in terms of distribution between projects "not-well-engineered" and "well-engineered".

🔍 **RQ$_2$.** *Are software engineering practices correlated to code smell presence in ML projects?*

RQ$_2$ aims to statistically analyze whether and how software engineering practices are related to code smell variations in ML projects.

Our empirical research had statistical connotations *i.e.,* we approached our research questions using statistical tests. The research method follows the guidelines of Wohlin *et al.* [29] and the ACM/SIGSOFT

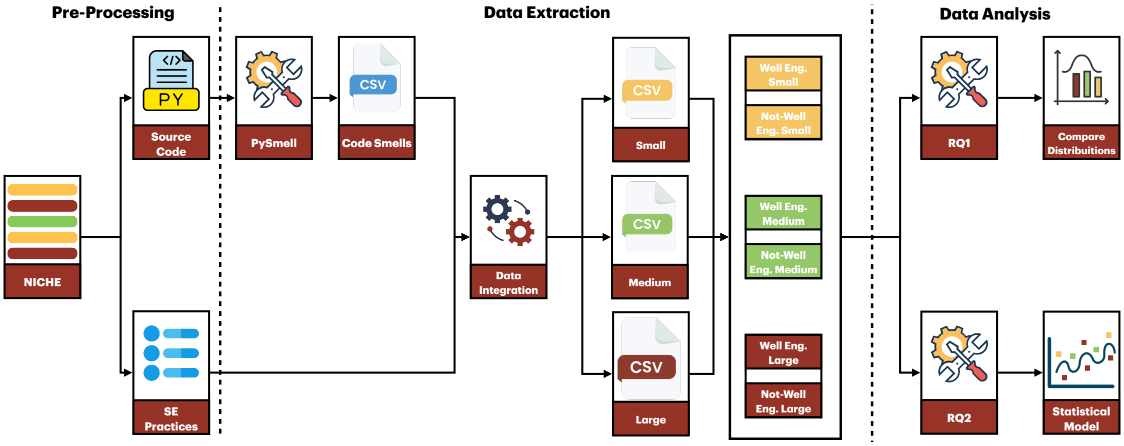

Fig. 1: Overview of the Research Method.

Table 1: Descriptive statistics of the NICHE projects.

	Stars	LOC	Commits
Min	100	234	100
1st Q.	211.2	4,022	218.2
Median	538.5	9,303	419.5
Mean	1991.3	24,672	1,241.3
3rd Qu.	1,641.0	22,308	1,065.8
Max	76,838	699,513	90,927

Empirical Standards.[1] Specifically, we used “General Standard”, “Data Science”, and “Repository Mining” guidelines. First, we cloned projects from NICHE [28], a dataset composed of 572 ML projects labeled as “well-engineered” and “not-well-engineered” according to eight software engineering practices; second, we identified code smells instances by running PySmell [4] *i.e.,* a static smell analyzer; then, we combined smell-related information with data provided from NICHE and split dataset calculating the LOC percentile, and lastly, we divided projects “well-engineered” and “not-well-engineered” and applied statistical tests to respond of our research questions. Figure 1 describes the research method applied. All data, materials, and scripts are publicly available in our online appendix [11].

3.1 Dataset Description and Pre-Processing

The careful selection of a representative and reliable dataset is essential to ensure the validity and generalizability of empirical findings. In this study, we rely on the *NICHE* dataset [28], which offers a curated collection of machine learning projects. This dataset was selected based on two primary considerations.

First, *NICHE* comprises only active and widely-used repositories, thereby minimizing the inclusion of personal or abandoned projects. Specifically, it includes 572 open-source projects, collectively accounting for 13,964,565 commits. Each project satisfies a set of inclusion criteria: a minimum of 100 GitHub stars, at least 100 commits, and a last commit date later than May 1st, 2020. To further refine the dataset and eliminate potential toy projects, we excluded six repositories containing fewer than 100 lines of code.

Second, all projects in *NICHE* have been manually classified as either “well-engineered” or “not-well-engineered”, based on the framework defined by Munaiah *et al.* [17] *i.e., Unit Testing*, *Architecture*, *Documentation*, *Issues*, *Continuous Integration*, *History*, *Community*, and *License*. A project is considered “well-engineered” when the majority of these practices are observed. This manual labeling process enables a principled comparison of engineering quality across projects, grounded in clearly defined dimensions.

Table 2 presents the eight-dimensional metrics defined by the Munaiah framework *et al.* [17]. As shown, the framework encompasses attributes that pertain to both the product and the development process. Within

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

Table 2: Description of the Software Engineering Practices Proposed by Munaiah *et al.*

Attribute	Description
Architecture	Defines the internal structure of the project by outlining its components and how they interact with other parts of the system.
Community	Indicates the presence of a broad and active group of contributors responsible for maintaining and evolving the repository.
Continuous Integration	Refers to the adoption of CI mechanisms to ensure a stable and reliable codebase throughout development and release.
Documentation	Includes technical documentation and supplementary resources that support understanding and maintenance activities.
History	Reflects the continuity of maintenance over time, highlighting frequent developer contributions and long-term viability.
Issues	Describes how requirement tracking and project management are handled directly through GitHub Issues, improving traceability.
License	Specifies the terms and conditions for reuse by explicitly declaring a software license in the repository.
Unit Testing	Indicates the presence of unit tests aimed at verifying the correctness of individual software components.

Table 3: Statistics of Projects Well Engineered.

		Stars	LOC	Commits
Small	Min	100	234	102
	1st Q.	166.5	1,704	171
	Median	336	2,724	276
	Mean	978.2	2,679	635
	3rd Q.	875	3,583	517
	Max	12,388	4,683	13,542
Medium	Min	100	4,689	102
	1st Q.	175	6,560	252.5
	Median	352	8,192	424
	Mean	1,203	8,230	701.9
	3rd Q.	911	9,726	861.5
	Max	18,087	11,685	3,938
Large	Min	105	11,711	105
	1st Q.	373.5	17,509	439.8
	Median	1,133	25,618	959
	Mean	3,204.5	51,952	2,442.6
	3rd Q.	3,702	45,550	1,905.8
	Max	76,838	699,513	90,927

the scope of the NICHE dataset [28], the authors manually assessed the presence of these attributes. However, in most instances, they did not provide quantitative values. Instead, the attributes were often represented using descriptive string values. For example, for the attribute “Issues”, rather than reporting a specific number of issues, the authors used qualitative descriptors such as “*Do reply to issues last time but not recently*”.

Considering this, as will be discussed in the following sections, it was not feasible to incorporate all metrics proposed by Munaiah *et al.* in our analysis.

Table 1 provides the statistical description of the attributes “Stars”, “LOC”, and “Commits” of the remaining 566 projects. As it is possible to see from the table, the distribution of the dataset shows a median of 538 Stars, 9,303 LOC, and 419 Commits. The statistical analysis reveals significant variability in project metrics, suggesting a high level of development activity. Furthermore, we observed substantial variation in LOCs. According to Zhou *et al.* [15], this aspect is a confounding factor in analyzing code-related metrics. To enhance our understanding of the NICHE dataset, we first segmented it into three groups —small, medium, and large— based on percentile calculations. Subsequently, we divided these groups further by sorting each according to the values in the “Engineered” column.

Table 4: Statistics of Projects Not Well Engineered.

		Stars	LOC	Commits
Small	Min	111	238	100
	1st Q.	210	1,259	118.5
	Median	443	1,652	162
	Mean	1,348	2,080	247.2
	3rd Q.	897	2,913	276
	Max	16,987	4,647	1,681
Medium	Min	100	4,687	105
	1st Q.	177.5	5,614	178.5
	Median	318	7,481	327
	Mean	895.4	7,708	375.6
	3rd Q.	1,496.8	9,369	420.2
	Max	3,944	11,672	1479
Large	Min	136	12,756	133
	1st Q.	251.5	19,040	222.5
	Median	596	27,017	365
	Mean	3,831.3	62,085	776
	3rd Q.	1,793.5	73,721	915
	Max	64,439	268,628	4,914

Small: This group consists of projects where the number of lines of code falls below the 30^{th} percentile. It includes 107 "well-engineered" projects and 59 "not-well-engineered", each less than 4,683 LOCs.

Medium: The second group includes projects whose LOCs fall between the 30^{th} and 60^{th} percentiles. This category includes 127 "well-engineered" projects and 44 projects "not-well-engineered" with LOCs between 4,683 and 11,685.

Large: The final group comprises projects that exceed the $60th$ percentile in terms of LOCs. It includes 202 "well-engineered" projects and 27 "not-well-engineered" projects. This group includes projects with more than 11,685 LOCs.

Table 3 and Table 4 provide a statistical description for projects labeled "well-engineered" and "not-well-engineered", respectively.

As the final step, we select from the practices proposed by Munaiah *et al.* [17] the most closely related to production code *i.e.,* we considered for this study the adoption of CI practices. It is important to note that we neglected the other factors due to their influential impact on the source code. However, to give more robustness to our analysis, we also extracted the exact number of members of the Community (a.k.a. Contributors) using PyDriller [22]. Lastly, we decided to discard the "History" attribute and instead use the number of commits, as the former offers only a string value (*e.g.,* "*Evidence of sustained commit activity*"). In contrast, the latter provides a precise, quantitative measure of the project's development activity.

3.2 Data Extraction

After cloning projects, we ran a static analyzer, namely PySmell [4], to identify smells. We selected this tool for two reasons. First, PySmell can detect ten instances of smells, some of which are derived from Fowler's original catalog [9] (*e.g., God Class*), while others are specifically tailored for Python projects (*e.g., Complex Container Comprehension*). Second, PySmell is considered state-of-the-art in detecting code smells in Python projects, showing an average of 87%, 92%, and 89% of precision, recall, and F-Measure, respectively, and it was used in previous studies for similar purposes [10,26,4]. We first divided projects according to section 3.1; second, we ran PySmell over the experimental objects, calculating the smell distribution.

3.3 RQ$_1$. Analyzing Code Smell Distribution

To address **RQ**$_1$, we examined the distribution of code smells in projects classified as "not-well-engineered" and "well-engineered" based on their specific percentiles. We utilized non-parametric statistical tests to determine if the distribution of each smell varied significantly between these two groups. Specifically, we employed the Mann-Whitney test [16], a non-parametric version of the Wilcoxon rank-sum test. We chose it due to the sample size and the non-normal distribution of the data [7]. We also applied Cliff's Delta (δ) [5] to measure the effect size of the observed differences. This test is particularly useful for evaluating the extent to which the distribution of smells differs between groups according to their percentile. Before conducting these statistical tests, we normalized the frequency of detected smells using the MIN-MAX strategy in the range [0-1] [20], ensuring a uniform scale for analysis.

The results were considered statistically significant at $\alpha = 0.05$. We formulated the following null hypothesis:

H0: *There are no statistically significant differences in terms of frequencies of Smell S_i of the Group $_j$ between projects "well-engineered" and "not-well-engineered"*

Where $S_i \in$ {list of smells detectable by PySmell} and j $\in$ {Small, Medium, Large}.

3.4 RQ$_2$. Analyzing correlation between Software Engineering Practices and Code Smells

To address **RQ**$_2$, we built a statistical model to analyze if and how software engineering practices are related to the emergence of smell. In the following, we reported this study's interesting independent, dependent, and control variables and the statistical test applied.

Independent Variables. Our goal is to understand how software engineering practices relate to the emergence of code smells. To this end, we focused on the software engineering practices that characterize a "well-engineered" project according to Munaiah *et al.* [17]. It is important to note that these dimensions are already included in the NICHE dataset [28]. As discussed in Section 3.1, we excluded Architecture, Documentation, History, Issues, License, and Unit Testing from our analysis. As a result, our set of independent variables includes Continuous Integration and Number of Contributors.

Response Variable. We considered the set of code smells detectable by PySmell as a response variable *i.e., Large Class*, *Long Parameter List*, *Long Method*, *Long Scope Chaining*, *Long Base Class List*, *Long Lambda Function*, *Long Ternary Conditional Expression*, *Long Message Chain*, *Complex Container Comprehension*, *Multiply-Nested*.

Table 5 shows the smells detectable by PySmell with their description.

Table 5: List of Code Smells Detectable by PySmell and the Related Description

Smell	Description
Large Class	A class that is excessively large
Long Parameter List	A method or function with an extensive parameter list
Long Method	A method that is excessively long
Long Scope Chaining	A method or function with multiple levels of nesting
Long Base Class List	A class definition with an excessive number of base classes
Long Lambda Function	A lambda function that is excessively long in terms of character count
Long Ternary Conditional Expression	A ternary conditional expression that is excessively long
Long Message Chain	An expression that accesses an object through an extensive chain of attributes or methods using the dot operator
Complex Container Comprehension	A container comprehension that is too complex is one that includes multiple nested comprehensions or conditions
Multiply-Nested	A container, such as a dictionary or list, with multiple levels of nesting.

Control Variable. Code smells can depend on variables unrelated to the independent variables. To mitigate potential threats to conclusions, we selected three control variables recognized as reliable estimators of code quality: Lines of Code (LOC), Number of Commits, and Number of Stars [19]. These control variables are already available in the NICHE dataset [28]. We limited our analysis to these metrics because there are no validated tools in the literature for extracting additional metrics. To ensure the validity of our findings, we manually assessed the potential for multi-collinearity between the variables in our study, and did not identify any multi-collinearity between variables.

Statistical Model. To assess possible correlation between independent, control, and the dependent variables, we employed the Kendall tau (τ) rank correlation coefficient [1]. Compared to other correlation metrics such as Spearman or Pearson, the Kendall tau coefficient provides several benefits: 1) It does not presume any particular type of relationship between variables, meaning a linear relationship is not required; 2) It does not necessitate that the data adhere to a normal distribution; 3) It does not require that values be equidistant; 4) It is highly robust against outliers; 5) The test can be used for a small sample size. It is important to note that Kendall's tau values are generally lower than other rank correlation coefficients, such as Spearman's, and cannot be directly compared. Given the lack of standardized interpretation, we followed Cohen's guidelines [6]: values between 0.1 and 0.3 indicate a weak correlation, 0.3 to 0.5 moderate, and above 0.5 high correlation. Statistical significance was assessed using two-sided p-values.

4 Analysis and Discussion of the Results

In this section, we report the results of our study and discuss the implications of our findings.

4.1 On the Distribution of Code Smells in ML projects

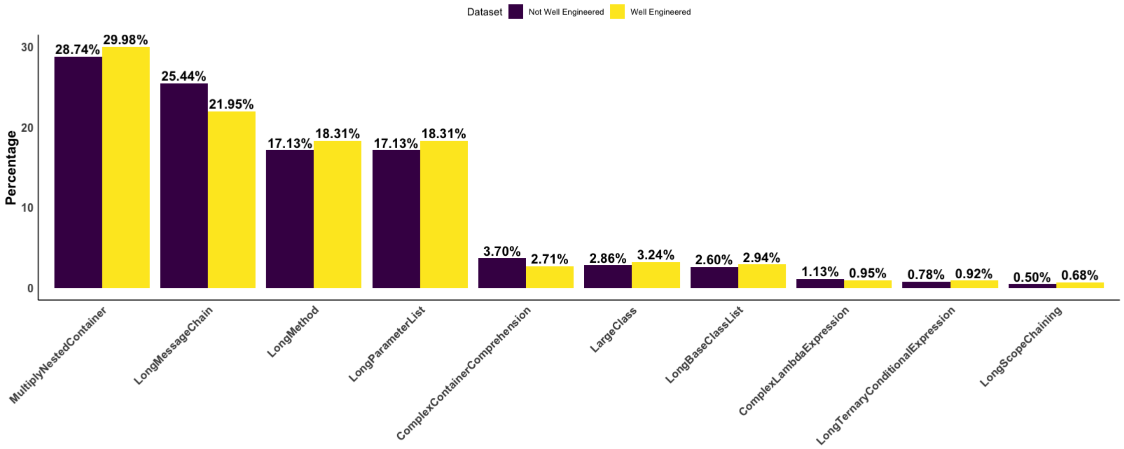

Fig. 2: Diffusion of Smells in Well and Not Well Engineered Projects.

Table 6: Results of The Mann-Whitney-Wilcoxon and Cliff Delta Tests

	Small		Medium		Large	
Code Smell	Mann-Whitney-Wilcoxon	Cliff Delta	Mann-Whitney-Wilcoxon	Cliff Delta	Mann-Whitney-Wilcoxon	Cliff Delta
Complex Container Comprehension	**1.538e-08**	**-0.50 Large**	**3.031e-10**	**-0.63 Large**	**8.385e-07**	**-0.58 Large**
Complex Lambda Expression	**2.704e-05**	**-0.36 Medium**	**1.616e-07**	**-0.51 Large**	**2.386e-10**	**-0.74 Large**
Large Class	**7.225e-09**	**-0.52 Large**	**7.615e-13**	**-0.71 Large**	**9.542e-09**	**-0.68 Large**
Long Base Class List	**4.386e-08**	**-0.49 Large**	**2.535e-13**	**-0.73 Large**	**1.016e-08**	**-0.68 Large**
Long Message Chain	**3.965e-11**	**-0.60 Large**	**4.313e-12**	**-0.70 Large**	**4.462e-09**	**-0.69 Large**
Long Method	**2.773e-11**	**-0.60 Large**	**3.571e-12**	**-0.70 Large**	**4.298e-09**	**-0.70 Large**
Long Parameter List	**2.773e-11**	**-0.60 Large**	**3.571e-12**	**-0.70 Large**	**4.298e-09**	**-0.69 Large**
Long Scope Chaining	**0.0001423**	**-0.32 Small**	**4.805e-08**	**-0.53 Large**	**1.862e-09**	**-0.71 Large**
Long Ternary Conditional Expression	**0.0005**	**-0.29 Small**	**1.197e-10**	**-0.64 Large**	**1.981e-09**	**-0.71 Large**
Multiply Nested Container	**3.664e-12**	**-0.63 Large**	**1.602e-12**	**-0.71 Large**	**5.079e-08**	**-0.64 Large**

In addressing the first research question, we first analyzed smell diffusion in absolute terms. Figure 2 illustrates the smell diffusion for "not-well-engineered" and "well-engineered" projects. From this figure, two key observations can be made. First, both groups exhibit the same top four smell frequencies. Second, we observed that "well-engineered" projects have a slightly higher propensity to be affected by the *Multiply Nested Container* smell (30% for "well-engineered" projects and 29% for "not-well-engineered projects"). These results align with previous findings by Zhang *et al.* [31], which identified Multiply Nested Container smell as the most frequent in Python code. This result could suggest that the framework of Muniah *et al.* [17] is not enough to capture the complexity of ML systems, and, in turn, suggests that other factors (*e.g.,* the number of stars or LOC) could be more relevant indicators for code smells.

Table 6 presents the results of the Mann-Whitney and Cliff's Delta tests. This comparison examines the distributions of "not-well-engineered" projects and "well-engineered" projects. Additionally, the results were organized based on their percentiles. The comparison shows significant differences in code smell distribution between the two categories. In many cases (*e.g.,* Complex Container Comprehension), extremely low p-values confirm strong statistical significance. Cliff's Delta also indicates a large effect size across all project sizes, suggesting a link between software engineering practices and reduced code smells. To conclude, we rejected the null hypothesis **H0**, *i.e.,* we identified statistical differences between code smell distributions between "well-engineered" and "not-well-engineered" projects.

Key findings of RQ$_1$.

RQ$_1$ indicates that code smell diffusion is broadly similar across "well-engineered" and "not-well-engineered" projects, with *Multiply Nested Container* being the most frequent in both groups. However, projects following SE practices are statistically associated with a lower overall presence of code smells.

Table 7: Results of the Kendall Correlation Coefficients for Projects Well-Engineered.

Dependent Variable	Continuous Integration	Community	Stars	Lines of Code	Commits
Complex Container Comprehension	-0.137**	0.076	0.051	0.306***	0.048
Complex Lambda Expression	0.047	0.066	0.106*	0.276*	0.068
Large Class	-0.062	0.136*	0.136*	0.331***	0.103*
Long Base Class List	-0.079	0.115*	0.135*	0.289**	0.077
Long Message Chain	-0.071	0.079	0.113*	0.360***	0.105*
Long Method	-0.078	0.104*	0.120*	0.368***	0.109*
Long Parameter List	-0.078	0.104*	0.120*	0.368***	0.109*
Long Scope Chaining	-0.037	0.056	0.098	0.303***	0.039
Long Ternary Conditional Expression	-0.040	0.020	0.104*	0.244*	0.005
Multiply Nested Container	-0.087	0.030	0.025	0.330***	0.050

4.2 On the correlation between Software Engineering Practices and Code Smells

Table 7 presents the results of the Kendall Correlation test for "well-engineered", large-sized projects. Due to space constraints, the other analyses can be found in our replication package [11]

Looking at the table, several key observations can be made. Firstly, there is a strong correlation between LOCs and all code smells. This result is expected, as an increase in LOC aligns with the definitions of code smells related to the LOC, such as *Large Class* and *Long Method.* The results indicate that as LOCs increase, the likelihood of code smells also increases significantly.

Examining the independent variables, our findings indicate a statistically significant negative correlation between CI tools and the *Complex Container Comprehension smell.* This suggests that CI tools could help maintain code quality by reducing the incidence of certain code smells. Community is positively related to *Large Class*, *Long Base Class List*, *Long Method*, and *Long Parameters List*, suggesting that the probability of certain code smells increases when the number of contributors increases.

Control variables such as stars, LOC, and commits exhibit a positive correlation with code smells. For example, projects with more stars tend to show smells like *Large Class* and *Long Message Chain*, possibly due to their larger size and more extensive evolution. This trend is reinforced by the positive correlation with commits, suggesting that frequent maintenance activities may contribute to smell proliferation over time.

> **🔑 Key findings of RQ$_2$.**
>
> **RQ$_2$** results show a positive correlation between control variables (Stars, LOCs, Commits) and code smells, suggesting that larger or more active projects are more prone to smell accumulation. We also found that a larger contributor base correlates with certain smells (*e.g., Large Class*), while CI tool usage is negatively associated with smells like *Complex Container Comprehension.*

5 Take-away Messages

Our results allowed us to formulate multiple reflections and implications.

CI Tools as Health Monitors. Our findings show a statistically significant negative correlation between the use of CI tools and the *Complex Container Comprehension* smell. This suggests that CI pipelines—especially when augmented with static analysis or quality gates—can play a role in proactively detecting and limiting code smells. Rather than merely automating builds and tests, CI may act as a structural safeguard, promoting consistent coding practices and early detection of complexity issues.

☞ *CI tools can serve as real-time health monitors of code structure, helping prevent the accumulation of code smells during development.*

More Contributors, More Complexity. We observed a consistent positive correlation between the number of contributors and several code smells, including *Large Class*, *Long Method*, and *Long Parameter List*. While community involvement is crucial in open-source ML projects, it can inadvertently increase structural complexity when coordination and coding conventions are lacking. Onboarding new developers without sufficient architectural guidance or automated checks may lead to divergence in coding styles and design decisions, amplifying the risk of technical debt.

☞ *Increasing the number of contributors can elevate structural complexity; scalable contributor strategies must include quality safeguards.*

Engineering Practices and Quality. Beyond CI and community size, our results show that ML projects labeled as “well-engineered” exhibit significantly fewer code smells across all size strata. This reinforces the value of applying general software engineering practices even in data-driven or experimental ML environments. Although not all individual practices could be quantitatively analyzed, the aggregate evidence supports their collective importance.

☞ *A disciplined engineering approach contributes to better code quality in ML projects, even in fast-paced or research-oriented contexts.*

Other Risk Factors. Control variables like LOC, number of stars, and number of commits were all positively correlated with smell presence. This is expected—larger and more active projects have more opportunity for smells to emerge—but it also implies that code quality monitoring should scale with project growth. Popularity (e.g., GitHub stars) does not imply structural cleanliness and may even mask accumulating technical debt.

☞ *Highly active or popular projects require proportionate investment in quality assurance to prevent degradation over time.*

Overall, our findings highlight the value of integrating software engineering practices into ML development. For practitioners, they offer evidence-based motivation to adopt CI and manage contributors effectively. For researchers and tool builders, they suggest directions for developing quality assurance solutions tailored to ML workflows. Future work may extend these insights across languages and domains to better align ML development with engineering discipline.

6 Threats to Validity

Some factors may have influenced our results. To address potential *construct validity* threats, we relied on the NICHE dataset, which includes only active and popular ML projects hosted on GitHub and provides a classification of engineering practices. While our use of PySmell for code smell detection introduces some limitations in terms of precision and recall, it remains a state-of-the-art tool for analyzing Python code and has been validated in prior work. In addition, still referring to the NICHE dataset, we used a binary classification where a project is considered well-engineered if the majority of the software engineering practices defined by Munaiah et al. [17] are fulfilled. While this approach is consistent with previous work, we acknowledge that it introduces a simplification: a project meeting just over half the criteria is treated the same as one meeting all, and differently from one falling just below the threshold. We highlight this as a limitation and an opportunity for future work to explore more granular scoring. Regarding *internal validity*, we mitigated the influence of confounding variables by controlling for project size (LOC), popularity (stars), and development activity (commits), which are known to correlate with code quality. In terms of *external validity*, our results are based on Python-based ML projects and may not generalize to other ecosystems. However, the dataset includes a wide range of projects in terms of size and complexity, and we plan to extend the analysis to other programming languages. Lastly, to preserve *conclusion validity*, we adopted robust statistical techniques, including non-parametric testing and Kendall's tau correlation, and verified the absence of multicollinearity among variables to ensure reliable interpretations.

7 Conclusion and Future Work

We investigated the relationship between software engineering practices and code smells in 566 Python-based ML projects from the *NICHE* dataset. Our analysis shows that projects adhering to established engineering practices, particularly Continuous Integration, tend to exhibit fewer code smells. Conversely, a higher number of contributors is associated with increased smell presence.

These findings highlight the importance of disciplined engineering and contributor management in maintaining code quality in ML projects. As future work, we aim to extend our analysis to other programming languages and explore practitioners' perceptions of code smells.

Acknowledgment

This work was partially supported by project FAIR (PE0000013), funded under the Italian NRRP (National Recovery and Resilience Plan), MUR program, and co-financed by the European Union – NextGenerationEU.

References

1. Abdi, H.: The kendall rank correlation coefficient. Encyclopedia of measurement and statistics **2**, 508–510 (2007)
2. Beck, K., Fowler, M., Beck, G.: Bad smells in code. Refactoring: Improving the design of existing code **1**
3. Cardozo, N., Dusparic, I., Cabrera, C.: Prevalence of code smells in reinforcement learning projects. In: 2023 IEEE/ACM 2nd International Conference on AI Engineering–Software Engineering for AI (CAIN). pp. 37–42. IEEE (2023)
4. Chen, Z., Chen, L., Ma, W., Zhou, X., Zhou, Y., Xu, B.: Understanding metric-based detectable smells in python software: A comparative study. Information and Software Technology **94**, 14–29 (2018). `https://doi.org/https://doi.org/10.1016/j.infsof.2017.09.011`, `https://www.sciencedirect.com/science/article/pii/S0950584916301690`
5. Cliff, N.: Dominance statistics: Ordinal analyses to answer ordinal questions. Psychological bulletin **114**(3), 494 (1993)
6. Cohen, J.: Statistical power analysis for the behavioral sciences. Routledge (2013)
7. Conover, W.J.: Practical nonparametric statistics, vol. 350. john wiley & sons (1999)
8. Fontana, F.A., Braione, P., Zanoni, M.: Automatic detection of bad smells in code: An experimental assessment. J. Object Technol. **11**, 5–1 (2012)
9. Fowler, M.: Refactoring: improving the design of existing code. Addison-Wesley Professional (2018)

10. 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 (2021)
11. Giordano, G., Della Porta, A., Palomba, F., Ferrucci, F.: The yin and yang of software quality: On the relationship between design patterns and code smells – online appendix. https://figshare.com/s/bb51da5b17872e473d53
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. Gupta, A., Suri, B., Misra, S.: A systematic literature review: code bad smells in java source code. In: Computational Science and Its Applications–ICCSA 2017: 17th International Conference, Trieste, Italy, July 3-6, 2017, Proceedings, Part V 17. pp. 665–682. Springer (2017)
14. 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. Springer
15. Lu, H.M., Zhou, Y.M., Xu, B.W.: The potentially confounding effect of class size on the ability of object-oriented metrics to predict change-proneness: A meta-analysis. Jisuanji Xuebao/Chinese Journal of Computers **38**, 1069–1081 (05 2015). https://doi.org/10.3724/SP.J.1016.2015.01069
16. McKnight, P.E., Najab, J.: Mann-whitney u test. The Corsini encyclopedia of psychology pp. 1–1 (2010)
17. Munaiah, N., Kroh, S., Cabrey, C., Nagappan, M.: Curating github for engineered software projects. Empirical Softw. Engg. **22**(6), 3219–3253 (dec 2017). https://doi.org/10.1007/s10664-017-9512-6, https://doi.org/10.1007/s10664-017-9512-6
18. van Oort, B., Cruz, L., Aniche, M., van Deursen, A.: The prevalence of code smells in machine learning projects (2021)
19. Palomba, F., Tamburri, D.A., Serebrenik, A., Zaidman, A., Arcelli Fontana, F., Oliveto, R.: How do community smells influence the intensity of code smells? In: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE). pp. 1064–1074. IEEE (2018)
20. Patro, S.G., Sahu, D.K.K.: Normalization: A preprocessing stage. IARJSET (03 2015). https://doi.org/10.17148/IARJSET.2015.2305
21. 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)
22. 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)
23. 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)
24. Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Penta, M.D., 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 . https://doi.org/10.1109/TSE.2017.2653105, https://www.scopus.com/inward/record.uri?eid=2-s2.0-85040307811&doi=10.1109%2fTSE.2017.2653105&partnerID=40&md5=d2b0c8b235fdd2cfb1fa8d1ec2b401ea
25. Van Emden, E., Moonen, L.: Java quality assurance by detecting code smells. In: Ninth Working Conference on Reverse Engineering, 2002. Proceedings. pp. 97–106. IEEE (2002)
26. Vatanapakorn, N., Soomlek, C., Seresangtakul, P.: Python code smell detection using machine learning. In: 2022 26th International Computer Science and Engineering Conference (ICSEC). pp. 128–133 (2022). https://doi.org/10.1109/ICSEC56337.2022.10049330
27. Vavrová, N., Zaytsev, V.: Does python smell like java? tool support for design defect discovery in python. The Art, Science, and Engineering of Programming **1**(2) (Apr 2017). https://doi.org/10.22152/programming-journal.org/2017/1/11, http://dx.doi.org/10.22152/programming-journal.org/2017/1/11
28. 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). https://doi.org/10.1109/MSR59073.2023.00022
29. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A.: Experimentation in software engineering. Springer Science & Business Media (2012)
30. Yamashita, A., Moonen, L.: Do code smells reflect important maintainability aspects? In: 2012 28th IEEE international conference on software maintenance (ICSM). IEEE
31. Zhang, B., Liang, P., Feng, Q., Fu, Y., Li, Z.: Copilot refinement: Addressing code smells in copilot-generated python code (2024)