

Università degli Studi di Salerno

Dipartimento di Informatica

Dottorato di Ricerca in Informatica
Curriculum Computer Science and Information Technology
XXXVIII Ciclo

PH.D. THESIS

Mind the Debt: Characterization, Detection, and Evolution of Technical Debt in AI-enabled Systems

GILBERTO RECUPITO

SUPERVISOR:	**PROF. FABIO PALOMBA**
CO-ADVISOR:	**PROF. DARIO DI NUCCI**
PHD PROGRAM DIRECTOR:	**PROF. ANDREA DE LUCIA**

A.A 2024/2025

SOFTWARE ENGINEERING LAB, UNIVERSITY OF SALERNO
HTTPS://SESALABUNISA.GITHUB.IO/

This research was done under the supervision of Prof. Fabio Palomba and Prof. Dario Di Nucci, with the financial support of the University of Salerno from November 1st, 2022, to October 31st, 2025.

The final version of the thesis has been revised based on the reviews made by Prof. Alexandros Chatzigeorgiou and Prof. Alexander Serebrenik.

The doctoral committee was composed by:

Prof. Carmine Gravino	University of Salerno, Italy
Prof. Panagiota Chatzipetrou	Örebro University School of Business, Sweden
Prof. Marc Oriol Hilari	Universitat Politècnica de Catalunya, Spain

First Release on October 2025. Revised on November 2025.

“Le vent se lève
il faut tenter de vivre”.
Paul Valéry

ABSTRACT

The widespread adoption of Artificial Intelligence (AI) in software systems introduces new layers of complexity to software quality assurance. As technologies evolve rapidly and development cycles accelerate, organizations often prioritize delivery speed over quality. This shift leads developers to make rushed design decisions, resulting in long-term maintainability and reliability issues. When these issues emerge in AI components, they manifest as AI-specific Technical Debt (AITD)—a phenomenon that can severely compromise software quality and lead to project failure.

Although AITD has received growing attention, prior studies have largely focused on categorizing debt types, offering a limited understanding of how to detect and manage them effectively. As a result, AITD continues to accumulate throughout system evolution.

The research project of this thesis aimed to fill these gaps, combining mining software repositories with mixed-methods research. The work (i) characterizes the properties and manifestations of AITD issues, (ii) proposes and validates novel automated detection approaches, and (iii) analyzes the evolutionary accumulation of AITD in AI-enabled systems over time. The work centers on three principal categories of AITD—architectural debt, Machine-Learning-specific code smells (ML-CSs), and data smells—and extends the investigation to self-admitted technical debt (SATD) to capture how debt is explicitly acknowledged by developers in source code comments.

The findings reveal distinct patterns for each debt type. Architectural issues are difficult for developers to perceive due to their system-level visibility. In contrast, ML-CSs and data smells are frequently overlooked, indicating their high prevalence in AI-enabled systems. For these two categories, the thesis proposes two tools to support AITD detection—CLAIRE and CODESMILE—both publicly available and empirically evaluated. Specifically, CLAIRE effectively generates accurate and human-understandable

representations of ML pipelines. CODESMILE achieves effective detection of 12 ML-CSs, with an average F1-score of 83%. Finally, ML-CSs and SATDs gradually exhibit long lifespans, and often accumulate unnoticed throughout system evolution.

As a key contribution, the thesis provides a comprehensive body of knowledge and actionable insights for understanding, detecting, and managing AITD. It supports practitioners by offering practical tools and empirical evidence to prevent debt accumulation, and it guides researchers by outlining current challenges and presenting a roadmap toward more mature, reproducible, and sustainable AITD management practices in AI-enabled systems.

CONTENTS

List of Figures xii
List of Tables xvi

PART I: Introduction and Background

1 Introduction 3
1.1 Context and Motivation 3
1.2 Research Statement 5
1.3 Research Contribution 8
1.3.1 Publicly Available Tools and Replication Packages 9
1.3.2 Structure of the Thesis 10
2 Background and Related Work 11
2.1 Literature Review on SE4AI 11
2.1.1 Lifecycle of AI-enabled systems 14
2.1.2 Quality in AI-enabled System 16
2.2 Technical Debt in AI-enabled systems 18
2.2.1 Code Debt 22
2.2.2 Architectural Debt 25
2.2.3 Data Debt 27
2.3 Self-Admitted Technical Debt (SATD) 29
2.4 Empirical Work on Technical Debt in AI-enabled systems 31

PART II: Characterization of AI-specific Technical Debt

3 Technical Debt in AI-Enabled Systems: On the Prevalence, Severity, Impact, and Management Strategies for Code and Architecture 37
3.1 Introduction 37
3.2 Research Method 40
3.2.1 Research Objective and Questions 40
3.2.2 Research Method 43
3.2.3 Context of the Study 43
3.2.4 Survey Design 45

3.2.5 Survey Validation 51
3.2.6 Survey Administration 53
3.2.7 Data Analysis 55
3.3 Analysis of the Results 56
3.3.1 Preliminary Analysis 56
3.3.2 **RQ$_1$**. How Frequent Are AITD Issues from the practitioner's Perspective? 59
3.3.3 **RQ$_2$**. How Severe Are AITD Issues from the practitioner's Perspective? 62
3.3.4 **RQ$_3$**. What Is the Impact of Introducing AITD Issues from the practitioners' Perspective? 65
3.3.5 Management Strategies 71
3.3.6 Causes of Technical Debt from the Practitioners' Perception 78
3.4 Threats to Validity 78
3.5 Conclusion 83
4 Unmasking Data Secrets: An Empirical Investigation into Data Smells and Their Impact on Data Quality 85
4.1 Introduction 85
4.2 Research Questions and Methods 87
4.3 On the exploration of data smells 89
4.3.1 Research Method Execution 91
4.3.2 Analysis of the Results 92
4.4 On the prevalence and impact of data smells on data quality 96
4.4.1 Data Collection 96
4.4.2 Data Smell and Data Quality Metrics Collection . . 97
4.4.3 Data Analysis to Address **RQ$_2$** 98
4.4.4 Data Analysis to Address **RQ$_3$** 98
4.4.5 Analysis of the Results 100
4.5 Threats to Validity 104
4.6 Conclusion 106

PART III: Detection of AI-specific Technical Debt

5 LLM-based Architecture to Create Graph Representations from ML Pipelines 111
5.1 Introduction . . . 111
5.2 CLAIRE: Extracting Logical Representation of ML Pipelines 115
5.2.1 Code Segmentation . . . 116
5.2.2 Graph Creation . . . 121
5.3 Evaluation . . . 123
5.3.1 Quantitative Evaluation . . . 123
5.3.2 Qualitative Evaluation . . . 132
5.4 Limitations and Future Work . . . 141
5.5 Threats to Validity . . . 143
5.6 Conclusion . . . 144
6 Detecting ML-specific Code Smells in ML-enabled Systems 145
6.1 Introduction . . . 145
6.2 Research Method . . . 148
6.2.1 Dataset Description and Projects Selection . . . 149
6.2.2 Data Extraction . . . 152
6.2.3 Building an ML-Specific Code Smell Detection Tool 152
6.2.4 Validation of the ML-Specific Code Smell Detection Tool . . . 156
6.2.5 Data Analysis . . . 161
6.3 Analysis of the Results . . . 163
6.4 Threats to Validity . . . 172
6.5 Conclusion . . . 174

PART IV: Evolution of AI-specific Technical Debt

7 On the Lifecycle of ML-specific Code Smells in ML-enabled Systems 177
7.1 Introduction . . . 177
7.2 Research Method . . . 179
7.2.1 Commit Data Extraction . . . 182
7.2.2 Public Data Availability . . . 184
7.3 Analysis of the Results . . . 185
7.3.1 **RQ**$_1$:When ML-Specific Code Smells are introduced 185

7.3.2 **RQ$_2$**: What tasks were performed when the ML-Specific code smells were introduced? 191
7.3.3 **RQ$_3$**: When and how are ML-specific code smells removed in ML-enabled systems? 193
7.3.4 ***RQ$_4$***: How long do ML-Specific code smells survive in the code? 200
7.4 Threats to Validity 202
7.5 Conclusion 205
8 On the Lifecycle of DL-Specific Self-Admitted Technical Debt 207
8.1 Introduction 207
8.2 Research Questions 209
8.3 Research Method 211
8.3.1 Dataset Description 212
8.3.2 History Mining 213
8.3.3 Data Analysis 215
8.4 Results 218
8.4.1 **RQ$_1$**: When and for how long do developers acknowledge DL-specific SATD in DL-enabled systems? 219
8.4.2 **RQ$_2$**: What tasks do developers perform when acknowledging the presence of DL-specific SATD in DL-enabled systems? 221
8.5 Threats to Validity 223

PART V: Conclusion, Contributions and Next Directions

9 Conclusion 227
9.1 **RQ$_A$:** Characterizing AI-Technical Debt 228
9.2 **RQ$_B$:** Detecting AI-Technical Debt 230
9.3 **RQ$_C$:** Evolution of AI-Technical Debt 231
9.4 Lesson Learnt 232
10 Open Issues and Next Directions 237

List of Publications 241

Bibliography 245

LIST OF FIGURES

Figure 2.1 Size of Google AI software in lines of code, the black box represents code *core AI*, and the light boxes illustrate supporting software code instead [161]. 13
Figure 2.2 Data smells catalog proposed by Foidl *et al.* [50] . 28
Figure 3.1 Structure of the pre-screening and the survey . . 47
Figure 3.2 Application of vignettes to define the Pipeline Jungle issue 50
Figure 3.3 Distribution of the roles of the participants 57
Figure 3.4 Distribution of the company sizes of the participants 58
Figure 3.5 Distribution of the team sizes of participants . . . 59
Figure 3.6 Distribution of the General skills of the participants selected for the questionnaire 60
Figure 3.7 Distribution of the Machine learning specific skills of the participants selected for the questionnaire 61
Figure 3.8 Prevalence of AITD issues in AI-enabled systems from the practitioner's perspective 62
Figure 3.9 Severity of AITD issues in AI-enabled systems from the practitioner's perspective 63
Figure 3.10 Effect of AITD issues on the selected quality aspects from the practitioner's perspective. Quality-related aspects are organized into three main categories: Quality Attributes (QA), Quality Metrics (QM), and Socio-Technical Factors (ST). 65
Figure 3.11 Frequencies of practitioners' answers on mainly-impacted quality-related aspects by AITD issues, categorized in quality attributes (QA), quality metrics (QM) and socio-technical factors (ST). 67

Figure 3.12 Frequencies of practitioners' answers of refactoring techniques useful for facing each AITD issue . 73
Figure 3.13 Effort depicted by the participant to identify AITD issues. 76
Figure 3.14 Effort depicted by the participant to refactor AITD issues. 77
Figure 4.1 An extended version of the catalog of data smells defined by Foidl *et al.* [50]. The new data smells are highlighted in bold. 93
Figure 4.2 Prevalence of Data Smells in the analyzed datasets 100
Figure 5.1 Architecture Overview: First, multiple LLMs extract logical segments from the source code. Then, a corresponding graph is created from those segments algorithmically. 115
Figure 5.2 Code Segmentation Process: Multiple (LLMs) collaboratively determine logical code segments in a three-stage process. In Stage 1, several LLMs propose segmentation for a code chunk. Stage 2 utilizes a different LLM to select one of these proposals. In Stage 3, yet another LLM generates a description and title for the selected segment. This process repeats until the entire code is processed, resulting in a list of labeled segments. 117
Figure 5.3 Segment Refinement: In some instances, LLMs may incorrectly place markers when an assignment is truncated. To correct this, we adjust the marker to the next valid position. For example, if the assignment for `nested_feature_transformation` continues to line 28 but the chunk ends at line 25, we manually relocate the marker to the end of the assignment to ensure accurate capture. 119

Figure 5.4 Example of CLAIRE-generated graph extracted from COMPAS Machine Learning pipeline showing “logical” (green), “execution order” (orange), and “data flow” (red) edges.. Red edges are labeled with the names of data-carrying variables. 122
Figure 5.5 Process of the Sequential Comparative Design for the Qualitative Evaluation. 132
Figure 5.6 Degree of clarity of goal understanding of the pipeline through the three representations. Stage 1 represents the answers after participants had only inspected either the code or the DAG representation. Stage 2 represents the results after the participants had inspected both the code and the DAG representation. Colors and bold labels represent the representation of reference for which the clarity is reported. 136
Figure 5.7 Usefulness of the DAGs elements reported by the participant during the artifact inspection. 139
Figure 5.8 Degree of adherence of DAG representation to the respective ML pipeline Code. 140
Figure 6.1 The process designed for the study. 149
Figure 6.2 Example of How CodeSmile Performs Type Inference. 154

Figure 6.3 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). 164

Figure 6.4 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). 165

Figure 6.5 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). 168

Figure 7.1 The process designed for the study. 180
Figure 7.2 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). 204
Figure 8.1 SATD introducing commit extraction 213
Figure 8.2 Frequency of SATD for each development stage. . 218
Figure 8.3 Distribution of SATD Lifetime since their introduction. 220

LIST OF TABLES

Table 1.1 Research studies discussed in this thesis and how they contributed to addressing current challenges. 9
Table 2.1 Code Smells Potential Negative Impact. 25
Table 3.1 Definitions of the collected AITD issues 46
Table 3.2 Prescreening questions 49
Table 3.3 Survey questions 52
Table 3.4 Task assignment 54

Table 3.5 Identification strategies proposed by AI practitioners for the AITD issues 72
Table 4.1 Distribution of Data Smells 99
Table 4.2 Spearman Test Results 102
Table 4.3 GLM Coefficient Statistics 103
Table 5.1 Average performance for selected LLM configurations. Best results per iteration are marked in **bold**. 127
Table 5.2 Combination of inspections with participant groups 135
Table 6.1 Descriptive statistics of the NICHE projects. . . . 150
Table 6.2 Descriptive statistics of the active projects divided by size. 151
Table 6.3 Rule-Based Conditions for Detecting Code Smells 157
Table 6.4 Performance Metrics of CodeSmile to detect each ML-CSs 159
Table 6.5 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 *'Merge API Parameter Not Explicitly Set'* (MAP). 167
Table 6.6 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). 170

Table 6.7 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). 171

Table 7.1 Segmentation of commits for analyzing the introduction of ML-CSs. 183

Table 7.2 Change operation tags for the rationale analysis. 184

Table 7.3 Overview of the number of projects, introducing commits, and initial set of commits. 185

Table 7.4 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). 187

Table 7.5 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). 189

Table 7.6 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). 190

Table 7.7 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. 193

Table 7.8 Overview of the number of projects, removing commits, and initial set of commits. 195

Table 7.9 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). 196

Table 7.10 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). 197

Table 7.11 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). 198

Table 7.12 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). 203
Table 8.1 Definition of SATD categories. 211
Table 8.2 Distribution of SATD Types Across Categories. Bug Fix (B), Feat (F), Enhance (E), Chore (C), Refactor (R) 221

LISTINGS

2.1 Example of Gradients Not Cleared before Backward Propagation in the Transformer project. 23
2.2 Example of the introduction of a SATD. 31
6.1 Example of Unecessary Iteration Smell. 146
6.2 Example of In-Place API Misused in the FederatedAI/FATE project. 155
8.1 Example of the introduction of a SATD. 208

PART I

INTRODUCTION AND BACKGROUND

1

INTRODUCTION

1.1 CONTEXT AND MOTIVATION

After decades of research, developing software without incurring quality degradation remains a persistent challenge. Software systems continue to accumulate issues related to bugs [28], vulnerabilities [75], and performance [113]. This is reasonable given the continuous evolution of the context, where software experiences daily challenges in keeping pace with timely business needs [87]. As projects evolve under continuous development, developers often rely on quick patches to deliver functionalities. While effective in the short term, such practices introduce deeper, hidden issues that silently contribute to degrading software quality. This phenomenon is recognized by Cunningham [34] as Technical Debt (TD).

Research investigating TD in traditional software has produced a rich body of knowledge, suggesting the use of established definitions, practices, and automated detection tools [88]. Thanks to this research contribution, developers can proactively monitor debt issues over time by adopting an automated static analysis tool (e.g., SonarQube [1]). However, these approaches are rooted in code, architecture, and design perspectives. Traditional TD frameworks primarily focus on managing and handling code deficiencies [36, 121], modularization issues [157, 201], and rushed design decisions [132] that accumulate over time.

The rapid rise of AI-enabled systems has exposed a critical warning in the evolution of software systems. As reported in the Gartner Summit 2024, more than 50% of companies developing Large Language Model (LLM) solutions for their services will likely abandon them due to costs, complexity, and accumulated technical debt by 2028 [53]. This critical

[1] SonarQube: https://www.sonarsource.com/products/sonarqube/

challenge highlights that maintaining an AI-enabled system is an upfront cost that increases over time. Avoiding it can lead to wasted efforts, even when integrating high-performing models [33].

The reason for being careful about the maintenance of AI-enabled systems is their nature, which makes them particularly prone to technical debt [161]. Their quality depends on dynamic and data-dependent artifacts—datasets, models, and pipelines—that continuously evolve and interact with one another. The addition of these artifacts within a software system alters its organizational, infrastructural, and technical characteristics. Novel expertise is introduced in the process, and the need for tailored ML Engineering practices overlays the traditional software engineering [163]. Furthermore, the unpredictable nature of AI models brings consequences beyond maintainability, often resulting in suboptimal solutions and inadequate quality assurance. Socio-technical issues may also emerge when AI systems violate ethical principles or produce biased results, leading to what has been termed *ethical debt* [130].

The urgent need to ensure quality motivates the definition of a novel category of debt: *AI-Specific Technical Debt (AITD)*. Sculley et al. [161] pioneered the discussion of technical debt issues unique to ML-enabled systems. Even when examining code and architecture artifacts, the entire system is redirected to a different process flow, where technical debt emerges at the system level. For instance, the *Pipeline Jungle* is a critical problem that arises when the continuous addition of activities in ML workflows leads to overly complex pipelines that become increasingly unmanageable over time. This redirection of quality concepts towards novel types of components brings an intrinsic shift in shaping technical debt to support the AI development process. Preliminary studies aimed to catalog and identify the issues that lead to AITD, and empirical evidence shows the relevance of debt unique to AI [18, 178]. Moreover, studies extended the exploration started by Sculley et al. [161], adding ML-specific code smells [206], data smells [50], and self-admitted technical debt (SATD) [115, 129] to the initial exploration.

However, the current understanding of AITD is largely conceptual, with limited empirical evidence for detailed interpretation. As a result, several challenges remain in research to create a substantial body of knowledge that helps developers continuously monitor the incidence of AITD.

1.2 RESEARCH STATEMENT

This thesis aimed to increase the body of knowledge by exploring AITD issues under three dimensions.

First, it is related to the level of definition and characteristics of AITD issues. While pioneering studies report experience of some AITD issues [161] or the type of debt that a practitioner can incur [18, 115], it is not clear what the prevalence of these AITD issues is, and how practitioners react to the presence of an AITD issue. Currently, aside from definitions of such phenomena, there is limited support for observing AITD in real-world systems and defining them through tangible properties. In other words, while studies report phenomena associated with AITD issues, developers are not able to recognize them without the association with specific properties visible in their artifacts. The lack of a clear characterization limits our ability to study the nature of *AI-Specific Technical Debt (AITD)* and constrains the development of reliable quality assurance practices for AI-enabled systems.

> ⚠ **Characterization:** Current research lacks knowledge about the characteristics and properties associated with AITD issues, limiting their observation.

Beyond the problem of characterization lies that of *detection*. Identifying AITD issues requires automated support that can reason about the interplay between data, models, and code. Recent efforts, such as the work of Zhang et al. [206], have extended the concept of code smells to machine learning applications by proposing a set of ML-specific smell types. Similarly, this effort is recognized in the definition of data smells,

i. e., data value-based indications of latent data quality issues caused by poor practices that may lead to problems in the future [50].

While these smells are connected to critical issues—such as data leakage or non-reproducible training—detection is tailored only to some of these issues, without full detection support for practitioners. Similar to what has been done for traditional code smells, various alternatives need to be evaluated to explore AITD issues further. Code smells are initially detected through rule-based approaches [107], textual analysis [124], and further refined through the use ML techniques [35, 51, 128]. Therefore, there is a need to explore and empirically evaluate validated approaches for detecting AITD issues, which have been continually improved over time.

> ⚠ **Detection:** There is a need for extended support through automated techniques in detecting AITD issues for real-world AI-enabled systems.

A further open question concerns the *evolution* of AITD issues across the lifecycle of AI-enabled systems. While traditional TD has been extensively studied from an evolutionary perspective [90, 182], comparable evidence for AITD is missing. We still do not know at which development stages AI-specific debt emerges (e.g., data preparation, model training, deployment), how long they remain in the system, or what triggers their removal. Without this temporal understanding, it is impossible to assess the lifecycle cost of AITD or to design practices that prevent its accumulation over time.

For instance, the introduction moment in the lifecycle may inform the development of refactoring techniques that recommend changes when debt is likely to be introduced, enabling developers to promptly improve code quality and prevent the accumulation of further debt. Similarly, evolutionary insights could reveal which debt instances are more frequently removed by developers, thereby uncovering an intrinsic prioritization strategy that naturally emerges in practice. Such knowledge can guide the design of automated or semi-automated support systems that align more closely with real-world developer behavior and maintenance practices.

> ⚠ **Evolution:** There is limited empirical knowledge about the lifecycle of AITD issues—specifically, when they are introduced, how long they persist, and how they are ultimately fulfilled or removed.

To address these three research gaps, this dissertation aims to advance the characterization, detection, and evolutionary understanding of AITD. The first research question is formulated as follows:

> 🔍 **RQ$_A$:** What types of AI-specific Technical Debt occur, and how do they manifest in AI-enabled systems?

By combining mining software repository analyses and practitioners' feedback, the thesis addresses this research question by exploring how AITD issues manifest in practice, their prevalence, and their severity. The second research question asks:

> 🔍 **RQ$_B$:** How can AI-specific Technical Debt issues be automatically detected in real-world AI-enabled systems?

This thesis explores and evaluates approaches to detecting AITD issues. By using static analysis and Large Language Models, the thesis addresses the research question to empirically discover and assess detection strategies, proposing publicly available online tools for AI practitioners. Finally, the third research question asks:

> 🔍 **RQ$_C$:** What are the lifecycle characteristics and persistence of AI-specific Technical Debt (AITD) issues?

This last research question focuses on understanding how AITD issues evolve. By capturing specific moments when AITD issues are introduced and removed, this thesis conducts software repository analyses to collect insights on their persistence and lifecycle.

The ultimate goal of the thesis is to provide empirical insights that collectively form a body of knowledge to understand and address AITD issues.

Together, these research questions aim to establish a comprehensive empirical and methodological foundation on AITD issues, contributing to comprehending how technical debt manifests, can be identified, and impacts the lifecycle of AI-enabled software.

1.3 RESEARCH CONTRIBUTION

This thesis provides different contributions to answer the research questions. The contributions of every study conducted in this thesis can be divided into three main groups, each addressing a related research question. Table 1.1 reports every specific contribution on the studies included in this thesis and how they are connected to each research question. Specifically:

- **Characterization of AITD issues:** This thesis includes an empirical analysis leveraging developers' perception to collect information about the prevalence, severity, impact on quality aspects, and management strategies for nine specific AITD issues. This study refers to the article [147]. Moreover, empirical analysis on the prevalence and the impact on data quality is conducted on the phenomenon of data smells, extending the range of AITD issues. This study refers to the publication [150].

- **Detection of AITD issues:** To address $\mathbf{RQ}_B$, a static analysis tool is built to detect 12 ML-specific code smells. This contribution, comprehensive of an empirical evaluation of the technique, refers to the publication [144]. A combination of static analysis and LLM-based techniques is built to help detect Pipeline Jungles, extending the set of detections on AITD issues with architectural granularity. This contribution, comprehensive of its quantitative and ML engineer's feedback evaluation, fulfills $\mathbf{RQ}_B$ for complex AITD issue detection, resulting in the article [146].

- **Evolutionary Aspects of AITD issues:** To address $\mathbf{RQ}_C$, empirical analysis is performed on the introduction and removal of ML-CSs

Table 1.1: Research studies discussed in this thesis and how they contributed to addressing current challenges.

ID	Study Name	Contribution	RG	Ref.
J01	Technical debt in AI-enabled systems: On the prevalence, severity, impact, and management strategies for code and architecture [147]	An empirical analysis of developers' perception of how AITD issues are characterized in real-world systems.	RQ_A	Chapter 3
C01	Unmasking Data Secrets: An Empirical Investigation into Data Smells and Their Impact on Data Quality [150]	An empirical analysis of the prevalence and the impact on data smells.	RQ_A	Chapter 4
J02	LLM-based Architecture to Create Graph Representations from Machine Learning Pipelines [146]	An empirical analysis to evaluate LLM-auto-generated DAGs to evaluate the clarity of ML pipelines.	RQ_B	Chapter 5
J03	Detecting ML-Specific Code Smells [144]	An empirical analysis to evaluate the effectiveness of rule-based detection of ML-Specific Code Smells.	RQ_B	Chapter 6
J03	On the Lifecycle of ML-specific Code Smells in ML-enabled Systems [144]	A large-scale empirical analysis on the evolutionary aspects of ML-specific code smell AI-enabled Systems and the activities that lead developers to introduce code smells in their systems.	RQ_C	Chapter 7
C02	On the Lifecycle of DL-Specific Self-Admitted Technical Debt [141]	A large-scale empirical analysis on the evolutionary aspects of AI-specific self-admitted Technical Debt and the activities that lead developers to introduce it in their systems.	RQ_C	Chapter 8

and their lifecycle. For each lifecycle moment, details on the tasks performed by the developer during the introduction and removal of ML-CSs are studied. This contribution refers to the publication of [144]. Evolutionary characteristics are also collected on the introduction of AI-specific self-admitted technical debt, aiming at understanding (i) when and for how long developers acknowledge SATDs in AI-enabled systems, and (ii) what tasks are performed during their acknowledgement. This contribution refers to the publication of [141].

1.3.1 *Publicly Available Tools and Replication Packages*

During experiment analysis, published tools and materials are publicly available online for further research. Specifically, our detection tools are publicly available on Github:

- CodeSmile, ML-CSs detector: https://github.com/giammariagiordano/smell_ai
- CLAIRE, ML Pipeline Inspector: https://doi.org/10.6084/m9.figshare.27400710

All the data, scripts, and replication items are available online, as discussed in [141, 144, 146, 147, 150].

1.3.2 *Structure of the Thesis*

This thesis is organized as follows, following the structure of our main research questions. Table 1.1 presents all the contributions distributed among the chapters of this thesis. Specifically, contributions are divided into three parts. Part II presents the analysis of AITD characterization. Part III presents the approaches, methods, and tools proposed for detecting AITD issues. Part IV presents empirical studies conducted to analyze the evolutionary behavior of AITD issues. Part V discusses the findings obtained during this research project and provides conclusions and insights for future directions.

2

BACKGROUND AND RELATED WORK

This Chapter provides the necessary background information to help explore existing work in AI-specific technical debt. First, it introduces the current state-of-the-art position of software engineering practices employed for artificial intelligence (AI). Subsequently, this section explores the notion of technical debt and its definition in the context of AI-enabled systems.

2.1 LITERATURE REVIEW ON SOFTWARE ENGINEERING FOR ARTIFICIAL INTELLIGENCE

The introduction of AI components into industrial and software contexts has sparked an intense interest in applying these solutions to enhance software engineering practices. Bosch et al. [20] argued that AI engineering builds upon software engineering by incorporating novel technologies and processes essential for developing such systems. Foidl and Felderer [48] defined a software system based on Machine learning (ML) employs algorithms to process data, utilizing ML models to autonomously make intelligent decisions based on extracted relationships, patterns, and data-derived knowledge. Martínez-Fernández *et al.* [99], analyzing the literature and the various ways to describe systems that involve AI, defined an AI-enabled system (or AI-based system) as a system consisting of different software and potentially other components, out of which at least one is an AI component. Based on this definition, an AI architecture can have traditional and AI software components. Sculley *et al.* [161] defined the typical structure of an AI-enabled system, where a tiny fraction of components aimed at the learning and inference process is surrounded by a series of traditional software components aimed at supporting the functionalities

of the AI component. Siebert et al. [166] specified AI software components, particularly those involving ML, as data-driven software entities. A data-driven software component is a software module that addresses a defined task (e.g., image segmentation) using data science methodologies, including ML, data mining, natural language processing, signal processing, and statistics. AI components, especially based on supervised ML or DL, differ fundamentally from traditional components because they are data-driven, i.e., their behavior is non-deterministic, statistics-oriented, and evolves in response to the frequent provision of new data [5].

Since the definition of an AI-enabled system allows comprehension of the architecture, other aspects emerge when software engineering practices are applied to AI. In this context, Tim Menzies then enunciates the five laws of software engineering for artificial intelligence to illustrate the importance of interaction between these two disciplines [103].

1. **AI software does not include AI components for the most part**: AI software includes many components that are supportive of the model and useful for performing operations such as data management and processing. The software may also include business components useful for providing the desired functionality of the system. Sculley et al. [161] analyzed the typical architecture of a machine learning system. Figure 2.1 represents, using the number of lines of code, the size of the different components of Google software. Given the architecture overview, only a few machine learning systems comprise machine learning code, but the majority is composed of components that support the core of the machine learning model.

2. **AI software needs software engineers**: All software needs installation, configuration, maintenance, and practices that can turn code into a product. Menzies then defines the future of software systems as being beyond the scope of a single discipline. There cannot be only software engineering or artificial intelligence, but on the contrary, promising ideas will arise from combining them.

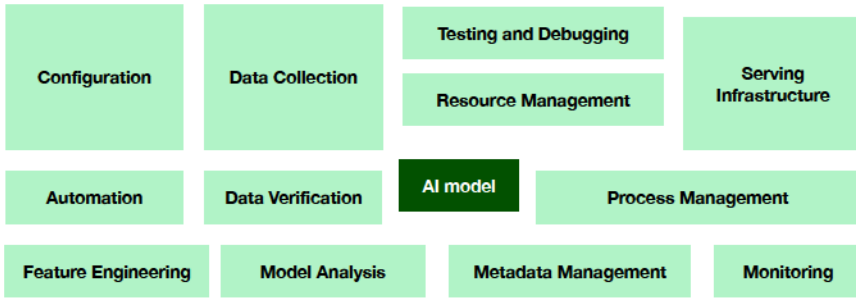


Figure 2.1: Size of Google AI software in lines of code, the black box represents code *core AI*, and the light boxes illustrate supporting software code instead [161].

3. **A bad software engineering practice implies a bad application of AI**: Since AI is also part of software creation, bad software engineering practices also affect AI software. Sculley *et al.* [161] reported that machine learning professionals use all the attributes available from enterprise databases to create predictive models. This leads to an increase in technical debt within the model, as the predictive models built suffer from a tight dependency on the enterprise data. When changes are made to the enterprise data, the predictive models must undergo updates and maintenance [160]. Violating software quality principles, such as the coupling and cohesion principle, also brings consequences in the artificial intelligence model.

4. **Better software engineering practice implies better AI application**: Although it is not necessarily true that applying software engineering methods can improve the quality of AI applications, several data science practitioners within industries have found extreme advantages in their use. Many frameworks, such as *scikit-learn* [1], offer several reuse-oriented features to integrate machine learning models and software engineering methods (e.g., continuous integration, cloud-based testing).

[1] https://scikit-learn.org

5. **Software engineers need particular types of AI**: Data analysis software often uses AI algorithms without going into detail about the design choices of the particular tool. Novielli et al. [114] point out that many models used in software engineering aimed at performing *sentiment-analysis* are built on data unrelated to the domain of interest. This leads to serious problems in terms of model performance. Thus, what are considered "general" AI tools are actually restricted to use in the domain in which they were trained. Software engineers must, therefore, use particular types of AI tools that are as well-suited as possible to the problem being addressed.

2.1.1 *Lifecycle of AI-enabled systems*

As already defined, applications of artificial intelligence, given its extreme importance in the industrial field, have evolved into large-scale systems. This has led to an increase in the complexity of managing certain systems. One of the main contributions that software engineering practices make to AI applications is lifecycle management. Steidl *et al.* [171] noted that creating a management system for the lifecycle of AI applications aims to have a well-fitted pipeline, maximizing the quality and the stability of AI components. Through the definition of different tasks that support the lifecycle of AI-enabled systems, including tasks related to the model and data management, AI applications are supported during all the phases defined in the pipeline, ensuring continuous monitoring and continuous improvement of the system [148].

These strong effects on the management process of AI-enabled systems led to defined lifecycle management models. One of the most widely used models is the CRISP-DM (Cross Industry Standard Process for Data Mining) [197], to provide the scientific community with a reliable and efficient model that supports data-intensive applications at each stage.

The internal process links identify the interactions between the different phases of the project, and depending on the output of each phase, the next phase to be executed is decided. The external links make explicit

how the model has a cyclic nature. At the end of the execution of the 6 phases, i.e., after the deployment phase of the model, new events from the environment may trigger the need to implement additional business requirements.

The six steps of the CRISP-DM lifecycle process are defined in:

- **Business Understanding:** This initial phase aims to extract the project objectives and requirements from a business perspective. The resulting knowledge will be converted into the definition of a specific data problem, and a project schedule will then be defined to achieve the objectives.

- **Data Understanding:** This phase involves the collection of data and proceeds with activities for the purpose of being able to understand the domain on which the project is proceeding. Quality issues will then be identified, and useful information will be extracted to achieve the objectives.

- **Data Preparation:** This phase involves all activities useful for the transformation of a raw dataset to the construction of the final dataset. Typical activities regarding the data preparation phase are data cleaning, attribute selection, and data restructuring.

- **Modeling:** In this phase, several techniques for model creation are selected and applied. In addition, a configuration of model parameters is conducted to find their optimal values and maximize model performance.

- **Evaluation:** Once the model has been built, it is important to evaluate the output of the previous phase and review each phase of the model in order to validate that the defined objectives have been met. The purpose is then to identify that the business objectives and quality requirements have been met.

- **Deployment:** The last phase aims to make the model usable by users. In this phase, the series of actions useful for carrying out the

deployment, organization, and presentation of the model will be defined.

Although AI provides results by extracting information from a static dataset [133], its pipelines are much more complex. They require significant infrastructure engineering in addition to code [62], including data extraction, data analysis, data transformation, model training, model validation, model serving, and model monitoring. MLOps is the set of practices designed to help model, develop, and operationalize the ML lifecycle, drawing on the DevOps principles [14]. By applying traditional DevOps practices to the pipeline, MLOps extends this set of features to include continuous training (CT) by collecting feedback and production data, allowing it to retrain and serve the new version of the model automatically [171]. To collect the necessary information for improving the AI model, the monitoring phase in the MLOps cycle is essential. Based on the metrics observed during this phase, it is possible to improve the model in several ways. Additionally, the monitoring phase is strictly connected to the quality assurance phase. Collecting data from the production allows us to identify possible quality issues in the components and the architecture. For example, collecting information about changes in data distribution allows the use of statistical techniques to identify data drifts [136]. Data drifts occur when the distribution of the dataset changes over time. Steidl *et al.* [171] identified through an interview that while data updates are introduced with the goal of improving the performance of the model, continuous change could lead to a significant drift of the distribution of the data, implying a decrease in the prediction capabilities of the model.

2.1.2 *Quality in AI-enabled System*

Due to the complexity of AI components, the impact of poor practices on these components is critical and not manageable as it is for traditional components. One of the source complications of operating with AI components is related to the maintainability of the AI-enabled system. This type of component provides non-deterministic results in which the testability

is not assessable with traditional testing approaches [104]. Moreover, the lack of requirements concerning the specification and robustness of AI models significantly impacts the quality of models designed for conventional systems [84]. Gezici *et al.* [55] conducted a systematic literature review on the software quality attributes used for AI systems. The main findings highlight the adaptation of practitioners of the ISO/IEC 25010 quality model to AI. Since this type of system has a different nature and applying traditional quality standards may not be suitable for the specific case in which AI is used, the findings highlight the lack of well-defined guidelines, frameworks, roadmaps, and models for measuring the quality of AI-enabled software.

Inadequate process definitions, quality metrics, attributes, and assurance techniques present new challenges for academia and industry. A systematic literature review by Martínez-Fernández *et al.* [99] concerning the main challenges in the field of SE4AI discovered that, besides traditional quality standards, there are specific quality attributes for machine learning and artificial intelligence related to safety and ethics. The specificity of these quality aspects highlights the need to define a new quality model that inherits the quality terms from traditional quality standards and adapts them to AI. The European High-Level Expert Group on AI presented the Assessment List for Trustworthy AI (ALTAI) [3]. It addresses ethical considerations and data privacy issues while highlighting the significance of reliability and quality. By introducing a series of self-assessment queries, the report helps AI developers identify prevalent quality issues often encountered in AI-enabled systems. Additionally, starting from the quality standard ISO/IEC 25010 [74] for traditional systems, the International Organization for Standardization (ISO) developed a new quality standard specific to AI systems: ISO/IEC 25059 [73]. The updated quality standard for AI systems emphasizes comprehensive product quality evaluation, including functional adaptability and correctness. These quality aspects are particularly relevant because models may not always guarantee correctness in all situations. The ability to utilize past data or outcomes for

future predictions is a critical point that can lead to the quality degradation of the model.

However, while quality aspects related to AI-enabled systems have been developed in the last few years, there is still a need to focus the research on several issues that could damage the overall quality of AI-enabled systems. In this regard, a few studies have delved into the potential effects of introducing quality problems. Thung *et al.* [180] performed an empirical study to assess the effect of bugs on machine learning systems. Analyzing three open-source machine learning systems, they found that most bugs affect the algorithm used by the system. Sun *et al.* [175] discovered seven highly diffused bug types by analyzing three machine learning projects. Golendukhina *et al.* [61] interviewed ten companies to extract quality issues in AI-enabled systems. They extracted 12 issues frequently encountered in the development process and identified how practitioners detect them. These pioneering studies call for intensified research into quality concerns within the AI domain. By highlighting the real-world consequences of quality issues, they emphasize the need for robust quality assurance measures to ensure the effectiveness and dependability of AI-enabled systems. Therefore, although various quality aspects have been explored, effective methods to manage and monitor the quality of AI-enabled systems still need to be established.

2.2 TECHNICAL DEBT IN AI-ENABLED SYSTEMS

Just as it is critical to conduct analysis and monitor the quality of one's product, it is equally crucial to perceive and identify issues that can harm quality. Many industries and various software professionals, to be able to meet the *time-to-market* (i.e., the amount of time required for a product, starting from the conception of the idea, to reach the stage of distribution in the market) and other business factors, find themselves facing what is referred to as *technical debt*.

Ward Cunningham coined the first definition of Technical Debt [34], describing it as a set of suboptimal design choices or implementation solu-

tions that can negatively affect data and code quality. This initial definition leads to the metaphorical connection of financial debt, where a software professional who makes suboptimal design choices can be compared to someone incurring a payment debt. When a person does not pay off their assigned debt on a regular and allowable schedule, they encounter a form of interest. This form of interest continues to grow in proportion to the time it takes to pay off the defined debt. Similarly, when a developer decides not to repay the debt created within the system and improve the introduced suboptimal solution, the interest accumulates, and the amount of technical debt increases. Thus, the cost of mitigating the system's technical debt increases proportionally to the time between its introduction and mitigation. In extreme cases, the interest accumulates to such an extent that it becomes impossible to mitigate, leading to extensive damage to the system and product abandonment. This case history is referred to as *technical bankruptcy* [176].

The financial metaphor of technical debt also led to the adoption of its terminology by software engineering professionals. Ampatzoglou et al. [6] analyzed the most commonly used terminology for representing and analyzing Technical Debt in systems and, in particular, extracted the definitions of technical debt *principal* and *interest*:

- **Principal**: aims to assess the effort required to address the current issue in a system that has an optimal level of design quality.
- **Interest**: aims to estimate the additional effort required to apply maintenance techniques and recover software quality.

From the general definition, several studies have been conducted to explore the occurrence of technical debt in depth. Tom *et al.* [181] explored the several types of granularity that highlight the level of the effect of each technical debt. In detail, the effect is related to the code (*code debt*), the design, and the architecture (*architectural debt*), in the environment of the application (*environmental debt*), from the relation of the community to the infrastructure (*knowledge distribution debt*) and the activities related to the testing of the system (*testing debt*). Subsequently, Li *et al.* [90] extended

the classification, including other technical debt types (*e. g., requirements debt and infrastructure debt*).

In the context of *code debt*, code smells, as defined by Martin Fowler [52], refer to certain issues that can significantly impact the maintainability of a software system. These issues are considered "*symptoms of poor design and implementation choices*", and if left unaddressed, they can deteriorate the overall quality of the system. Consequently, it is crucial to protect the quality of software systems by actively detecting and resolving code smells. To this end, researchers have conducted numerous studies to identify techniques for detecting code smells. Initially, structural metrics and historical metrics were explored, resulting in the proposal of various tools and methodologies [98, 107, 123, 124, 182]. However, with AI's advancement and general adoption, machine-learning techniques are proposed to identify code smells [96, 128]. The impact of code smells extends beyond just maintainability. Studies have shown that code smells can directly affect other key aspects of software quality, such as performance [25], change-proneness [78, 117], and reliability [76]. Moreover, the occurrence of code smells can affect the development process and the productivity of the development team [181], making it imperative to address these issues proactively.

With the rise of AI systems, researchers have also started exploring technical debt in the context of these systems. One notable study in this direction is the work by Sculley *et al.* [161]. This study analyzes technical debt in ML-enabled systems and highlights the challenges posed by such debt. The main outcome of this study highlights how ML-enabled and AI-enabled systems are particularly exposed to technical debt. These systems derive their nature from traditional systems and contain code and components. This characteristic made AI vulnerable to traditional technical debt. Moreover, these systems are also vulnerable to AI-Specific Technical Debt (AITD). In this context, Bogner *et al.* [18], through the conduction of a systematic mapping study, defined all the types of technical debt that are diffused in AI-enabled systems.

Before reporting on the background and the related literature on technical debt in AI-enabled systems, we provide the basic definitions used throughout the empirical study:

- **AI-enabled systems:** An AI-enabled system represents "a software system with functionalities enabled by at least one AI component, e.g., for image-, speech-recognition, and autonomous driving"[99]. In the context of this study, the term refers to systems integrating AI technologies to meet diverse business and technological requirements.

- **AI components:** An AI component is a module or a unit within an AI-enabled system "whose behavior relies on AI code or an AI library that provides an implementation of AI algorithms"[99]. Understanding the definition of AI components is crucial for comprehending the interactions of participants involved in developing, implementing, or maintaining these integral modules.

- **AI practitioners:** As AI practitioners, we intend all practitioners involved in building AI-enabled systems, from data management activities to deploying and serving the system into production. The activities involved in building AI-enabled systems include additional roles that oversee the development and management of the software system, including Data Scientists and Software Engineers [119]. This definition allows us to collect different perspectives and priorities on the quality of process and the product of AI-enabled systems.

- **AI-Specific Technical Debt:** Technical debt is a metaphorical concept introduced by Ward Cunningham in 1992 [34]. It refers to the compromises made during the design, implementation, or maintenance of software systems to meet immediate goals, often at the expense of the overall system's quality. Technical debt encompasses various factors, including suboptimal coding practices, hurried decision-making, lack of documentation, and architectural shortcuts [71]. While sometimes incurred unavoidably due to factors like

> tight deadlines or evolving requirements, accumulating technical debt can hinder future development efforts, increase maintenance costs, and impede the evolution of the software. As for traditional technical debt, we refer to AI-specific technical debt (AITD) as "a metaphor to indicate the future costs of repaying quality concerns introduced by suboptimal solutions integrated into the building process of AI-enabled systems" [147]. The set of suboptimal solutions that contribute to accumulating debt is defined as *AITD issues*. An example of AITD issue is the so-called Pipeline Jungle, which arises when an AI pipeline becomes overly complex and challenging to maintain over the evolution of the AI- The system is enabled due to the presence of many interconnected data processing and machine learning components.

The most notable types of debt in software development include Code, Architectural, and Data debt. In the following sections, AITD issues for these types of debt are explored and detailed.

2.2.1 *Code Debt*

Gesi et al. [54] conducted an empirical study on deep learning projects, focusing on the frequency of code smells. In addition to traditional code smells, they identified five new code smells specific to deep learning projects. This study enriches the understanding of code quality issues in the context of deep learning and highlights the importance of tailored approaches to tackle technical debt in ML projects. The code smells are:

- **Scattered Use of ML Libraries:** Scattered use of ML API in multiple files. Once ML API needs to be modified, the practitioners must modify places across several files.
- **Unwanted Debugging Code:** A debugging code fragment, method, or class is no longer used, but it is still part of the source code.
- **Deep God File**: A file containing multiple components of an AI system, e. g., model training, validation, and testing.

Zhang et al. [206] conducted an empirical study to identify code smells specific to machine learning applications. Through the analysis of 1,750 papers, including GitHub commits and Stack Overflow posts, they identified 22 ML-Specific Code Smells (ML-CSs). This type of code smells extracted from this study affects several quality-related aspects, including readability, error proneness, and efficiency. Each of these code smells is then linked to the relative stage of the ML pipeline. Finally, the code smells are classified as *Generic* and *Api-Specific* depending on whether they can occur in the presence of a specific ML-related library or not. An example of the identified code smells is *"Gradients Not Cleared before Backward Propagation"*: When gradients are not cleared before backward propagation, it can lead to incorrect results and complicate debugging and code optimization. An example of this code smell is reported in Listing 2.2.1.

```
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 2.1: 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 [189]. The function `optimizer.zero_grad()` should be used before the backpropagation step to mitigate this smell. Listing 2.1 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.* [206].

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

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

Table 2.1: Code Smells Potential Negative Impact.

Code Smell	Potential Issue
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.
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.
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.
In-Place APIs Misused	Some methods return a copy instead of modifying data in-place, leading to unintended behavior.
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.
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.
Memory Not Freed	If the machine runs out of memory during training, the process will fail.
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.
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.
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().
TensorArray Not Used	Use tf.TenTensorArray() is used to grow arrays in a loop, avoid inefficiencies, and excessive intermediate tensor creation.
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.

instruments that are not only tailored to the unique challenges of ML pipelines but also specifically adapted to the peculiarities of ML-CSs.

Table 2.1 shows the 12 ML-specific code smells detectable using CODESMILE [144].

2.2.2 *Architectural Debt*

Based on the definition of Bogner et al. [18], architectural debt variations signify a pivotal aspect of AI-based systems, highlighting the intricate interweaving of architectural components with their fundamental data. In this

increasingly data-centric landscape, the relationship between architectural design and underlying data is not just a superficial connection, but a profound entanglement that shapes the very essence of these systems. This section outlines all the AITD issues related to architecture.

CORRECTION CASCADES. This situation is recurrent when updating and reusing a model. Consider a model *a* for the problem *A*. This model is reused for the construction of a new model *a'* corresponding to the solution of problem *A'*. This correction, though initially beneficial in solving the problem, results in constructing a model that relies entirely on another model. This situation can deteriorate into a continuous worsening if corrections occur in a cascading manner, thereby continuing to generate dependencies between models.

GLUE CODE. Professionals who help build Machine Learning systems, particularly data science experts, tend to define *general-purpose* solutions as individual, stand-alone *packages*. Thus, when the package needs to be used in another model or system, there is a tendency to insert large portions of code that are not necessary for the application's purpose but are useful for adapting these components within the system. An instance of *glue code* can lead to excessive increases in maintenance costs over time. The continual addition of components within a system results in increased resources required to run the application until the system becomes completely blocked. Sculley et al. [161] found that in the academic community, only 5 percent of application code includes code related to machine learning operations, while the remaining 95 percent is representative of *glue code*.

PIPELINE JUNGLES. This type of antipattern within the system may evolve with each addition of data or new information. This new information may require additional data processing and preparation operations to be added to the standard process defined by the pipeline. Such continuous

additions increase the machine learning pipeline, leading to a problem that requires high identification and maintenance costs.

MULTIPLE LANGUAGE SMELL. The use of multiple languages may seemingly be an optimal solution that allows the use of different libraries. Still, it has the side effect of increasing the testing and comprehensibility costs of the components. The severity of this particular smell instance increases when responsibility for the element is transferred to other experts.

UNDECLARED CONSUMERS. Sculley *et al.* [161] define that *Undeclared Consumers* occur when free access to the model exists. Without access control, it is possible that some applications not identified as collecting the model's output could still be connected to it. This situation could lead to several problems. First, free access can provide a better understanding of the system, which is useful for identifying and exploiting potential security breaches and performing malicious attacks on the system. Secondly, the dependencies between the model's output and these undeclared consumers could lead to a tight coupling, in which each change on the main system propagates uncontrolled to all the other applications.

JUMBLED MODEL ARCHITECTURE. Gesi *et al.* [54] identified this issue from the recurring pattern of putting all the components of the AI-enabled system in a single component. This architectural approach often results in a reduction in the comprehensibility and maintainability of the system. In essence, it reflects the practice of conflating disparate elements within an AI system, which hinders the ability to comprehend its structure and makes it challenging to maintain or adapt over time.

2.2.3 *Data Debt*

Considering that data preparation is the first step in beginning to build a machine learning model, it is necessary to consider that data dependency debt requires special attention because of the danger it can cause, since it

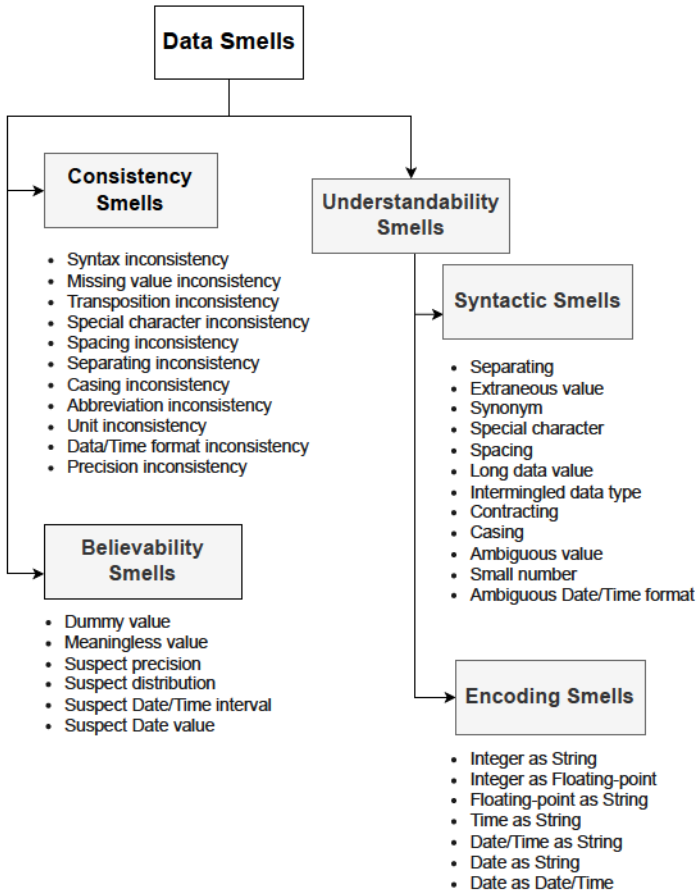


Figure 2.2: Data smells catalog proposed by Foidl *et al.* [50]

is possible that the technical debt employed during data preparation can affect all subsequent steps in the machine learning pipeline. We therefore define *Data Dependency Debt* as the situation where the data used within our model creates a chain of dependencies that is difficult to maintain.

DATA SMELLS Foidl *et al.* [50], analyzing the potential data quality issues in AI-based systems and their consequences, presented a catalog of 36 *data smells*. The authors conducted a study on more than 240 real-world datasets to validate the effectiveness of the proposed approach. Data

smells are defined as "*context-independent, data value-based indications of latent data quality issues caused by poor practices that may lead to problems in the future*" [50]. Through the catalog identified, data smells are defined into three main categories, as visible in Figure 2.2:

- **Believability Smells**: These smells relate to the believability of the data and include issues such as outliers, missing values, and incorrect data types. An example of this type of data smell is the *Dummy Value*. This smell arises when other values are used to represent a characteristic situation. For instance, the frequently observed use of "999" to denote missing values in a distribution. In specific contexts, such as registering telephone numbers, interpreting this value could be ambiguous since it might also represent an emergency number.
- **Understandability Smells:** These smells relate to the understandability of the data, often threatened by incorrect representations. An example of this type of smell is *encoding smells*, in which the encoding type used for the value representation is inappropriate. For instance, numeric values are represented as strings.
- **Consistency Smells:** These smells relate to the consistency of the data distribution. The use of inconsistent types, formats, or encoding could lead to the creation of ambiguous representations of the same value. A simple example is the *abbreviation inconsistency* that arises when different abbreviations are used to represent the same concept (e. g., "Dr." ,"Doc.", "dr.").

2.3 SELF-ADMITTED TECHNICAL DEBT (SATD)

One of the methods to explore issues that have not been explored in the literature is to rely on what the developers report in source code comments as Self-Admitted Technical Debt (SATD) [134]. Developers who are aware of a suboptimal solution in the system or a code flaw use explicit annotations in source code comments to acknowledge the presence of a technical debt.

Common annotations includes significative keywords such as *'TODO'* and *'FIXME'* [100].

An explicit example of SATD is reported in Listing 2.3, denoting an AI-specific issue in the project HUGGINGFACE/DIFFUSERS. [3] The reported issue highlights the need to leave an old solution to avoid ripple effects in the input processing parts of the project. However, the current situation of the project does not adopt the suggested solution, which involves the use of tensor inputs.

In AI-enabled projects, developers frequently acknowledge debts related to data, models, and pipelines that are not present in traditional software. For example, SATD may indicate dataset quality concerns (e.g., "need to balance classes"), model reproducibility issues (e.g., "fix random seed"), or architectural trade-offs in pipeline components (e.g., "replace CPU inference with GPU support"). This characteristic of SATDs makes them a valuable resource for exploring various issues that arise throughout the different phases of the ML pipeline. Comments made by developers often reveal debts related to data preprocessing, model training, evaluation, and deployment, highlighting the complexity and interdependence of the stages in the ML lifecycle. By analyzing these explicit acknowledgments, researchers can gain insights into which components of the AI workflow are most susceptible to technical debt and how such debts evolve.

Initial efforts to explore these issues were made by OBrien et al. [115], categorizing them using the mapping of traditional TD types, involving code, requirements, test, defect, design, and documentation [13]. Building on this foundation, Pepe et al. [129] introduced a more comprehensive taxonomy specifically designed for deep learning projects. Their taxonomy classifies AI-related debts according to the affected artifacts of the AI lifecycle, including data, model, training, evaluation, and the pipeline. It also emphasizes how domain-specific characteristics influence both the accumulation and elimination of debt.

[3] Example of AI-Specific SATD: https://github.com/huggingface/diffusers/blob/main/tests/pipelines/stable_diffusion_2/test_stable_diffusion_inpaint.py#L120

```
def get_dummy_inputs(self, device, seed=0):
# TODO: use tensor inputs instead of PIL, this is here just to leave
    the old expected_slices untouched
    image = floats_tensor((1, 3, 32, 32),
    rng=random.Random(seed)).to(device)@@
    image=image.cpu().permute(0, 2, 3, 1)[0]
    init_image =
    Image.fromarray(np.uint8(image)).convert("RGB").resize((64, 64))
```

Listing 2.2: Example of the introduction of a SATD.

2.4 EMPIRICAL WORK ON TECHNICAL DEBT IN AI-ENABLED SYSTEMS

The state of the art on assessing and evaluating technical debt in AI-enabled systems is still at a preliminary stage. All the studies that explore the prevalence of code smells are supported by the emergence of two main static analysis tools to identify code smells and coding standards violations: Pysmell [4] and Pylint[5]. Chen *et al.* [26] have developed PySmell, a static analysis tool that identifies 11 code smells through the use of structural metrics. On the other hand, Pylint is a robust static code analysis tool that surpasses PySmell by enforcing coding standards, detecting code smells, and checking for potential bugs. Its extensive features make it a popular choice for ensuring code quality and maintainability in Python projects. Through the use of these tools, some studies are conducted to observe traditional code smells and quality issues in AI-enabled systems. Van Oort *et al.* [188] analyzed 74 open-source ML projects to identify the prevalence of code smell related to coding quality standard conformance. The authors discovered that issues related to code duplication are very common in ML projects. Jebnoun *et al.* [77] performed a comparative analysis to analyze the prevalence of code smells in 59 open-source deep learning systems.

[4] https://github.com/chenzhifei731/Pysmell
[5] https://www.pylint.org/

The findings of this study highlight the high frequency of *Long Ternary Conditional Expression*, *Complex Container Comprehension*, and *Long Lambda Function*. Simmons *et al.* [167] performed a large-scale analysis to find the prevalence of coding standard violations. Their finding highlights the high prevalence of coding standard violations in Data Science projects, which are twice as prevalent as in non-Data Science projects. While research challenges still need to be addressed to understand the phenomenon of code smells in AI-enabled systems, the studies highlighted in this section provide interesting findings on their high prevalence.

On the other hand, focusing on AITD, the state of the art aimed to discover and extend the set of AITD issues presented, instead of empirical studies that analyze the prevalence and effect of their introduction. Tang *et al.* [178] analyzed code patches in 26 open-source machine-learning projects to identify machine learning-specific technical debt and refactoring techniques. Their findings highlight the diffusion of 7 new specific technical debt, related to the duplication of machine learning components and reusability issues. Moreover, they identified 14 new refactoring techniques as possible solutions to address part of the established AITD issues. Wang *et al.* [191] conducted a semi-structured interview to identify AITD issues and management techniques. Firstly, the participants in this study highlight that the term *technical debt* is not widely used in industries focused on AI-enabled systems. Consequently, the participants reported almost all of the AITD issues proposed in the interview without possible refactoring solutions to address them. Liu *et al.* [91] conducted an empirical study to extract self-admitted technical debt from 7 popular open-source deep learning projects. Their findings highlight the extremely high diffusion of self-admitted AITD issues, particularly those related to design debt. The results also highlight that self-admitted technical debt in deep-learning projects may be more prevalent than traditional ones, as the number of comments in the deep-learning projects analyzed is 57% higher. Washizaki *et al.* [192] conducted a systematic mapping study to identify software engineering design patterns and antipatterns in machine learning systems. Including grey literature in the analysis, beyond the an-

tipatterns defined by Sculley *et al.* [161], they identified a specific case of *Jumbled Model Architecture*, in which all the components that compose the machine learning pipeline are mixed in a single notebook. Moreover, they mapped the antipatterns identified with the stages of the machine learning pipeline and the traditional software development lifecycle.

PART II:

CHARACTERIZATION OF AI-SPECIFIC TECHNICAL DEBT

3

TECHNICAL DEBT IN AI-ENABLED SYSTEMS: ON THE PREVALENCE, SEVERITY, IMPACT, AND MANAGEMENT STRATEGIES FOR CODE AND ARCHITECTURE

3.1 INTRODUCTION

Contemporary software systems are more and more empowered by Artificial Intelligence (AI) modules, which companies and individuals use to make informed decisions [208] or automate demanding tasks requiring human workload [140]. The impact of AI on industry and society is pervasive, with a further increase expected in the coming years in multiple domains [139, 207]. In such a context, researchers, high-tech companies, and government agencies are actively engaging with the definition of novel quality assurance practices that may cope with the continuous evolution of AI algorithms and models [99]. For instance, the *International Organization for Standardization* (ISO) has recently introduced a new quality standard, i.e., the ISO/IEC 25059:2023 [73], that incorporates AI-specific characteristics, e. g., functional correctness of the model. These recent advances are designed to provide practitioners with mechanisms to control for the evolution of AI-specific quality properties [104]. However, they do not cover the *integration* of AI modules within more complex software systems, representing a key limitation. Indeed, as recently pointed out by Sculley *et al.* [161], machine learning modules only represent a small fraction of the code contained within machine learning-enabled systems, and their integration is often challenging and a source of quality concerns that may affect multiple maintenance and evolution properties, other than the overall effectiveness of these systems.

An example of AITD is the so-called *Pipeline Jungle*, which arises when an AI pipeline becomes overly complex and challenging to maintain over the evolution of the AI-enabled system because of the presence of many interconnected data processing and machine learning components.

In their seminal paper, Sculley *et al.* [161] argued that issues due to technical debt are highly diffused in practice and proposed an initial catalog of more than 20 types of technical debt affecting ML-enabled systems under different angles, e. g., issues causing code and architectural concerns. Later, Bogner *et al.* [18] extended such the initial catalog by conducting a systematic mapping study, through which 72 additional anti-patterns affecting multiple stages of an AI pipeline were identified. Most research efforts have been conducted to elicit and characterize the properties of AITD [4, 77, 95, 101, 109], while only a few attempts have been made to identify it [21, 48, 91]. As such, we argue that the scientific knowledge of AITD is still limited. More specifically, our research highlights that key aspects, such as the prevalence of the problem in practice, the severity and impact of AITD, and the mitigation strategies to address it, remain unexplored.

> **◎ Objective of the Study**
>
> *In this study, we aim to enlarge the knowledge of AI technical debt issues that affect the code and architecture, investigating their criticality and how practitioners mitigate those concerns in practice.*

On the one hand, our focus on these AITD issues is driven by our willingness to investigate quality concerns that may lead to ripple effects on the quality of an entire AI-enabled system [88, 164]: by investigating the debt items defined in the literature, we could therefore provide further insights on actually harmful concerns, assessing how critical they are in practice and what strategies to put in place to limit their adverse effects. On the other hand, no previous work targeted the analysis of those types of AITD [164]: as we intend to extend the current body of knowledge, we deemed these quality concerns as an ideal starting point for our research.

We address the main objective of the study by designing and executing a survey study based on an online questionnaire involving real-world practitioners experienced in developing and maintaining AI-enabled systems. We focus our analysis on 9 AITD issues defined by Sculley et al. [161], Gesi et al. [54], and Bogner et al. [18]. We opt for such a research method in light of analyzing the current state of the art. While the software engineering research community is actively engaging with the problem of AITD, scientific knowledge on the matter is still limited. On the contrary, we notice that practitioners seem to perceive the problem of AITD as critical —as proof of that, practitioners released the preliminary catalog of these types of quality issues in ML-enabled systems [161], and other prominent related resources(e. g., [21, 101]) have been released by practitioners. As such, our work aims to survey the state of the practice to inform about the next research avenues to pursue to support practitioners better when dealing with AITD issues. Our survey study involves 53 participants who were consulted about the frequency, severity, impact, and mitigation strategies for nine distinct AITD concerns. The key findings of our study revealed that practitioners are relatively unaware of the prevalence of these AITD concerns, yet they consider them severe and harmful.

The key findings of our study revealed that practitioners, while recognizing the low prevalence of AITD issues, consistently recognize the severity and potential harm these issues can inflict on the systems' quality. Furthermore, the study highlights that the primary response to these concerns involves manual inspections and ad-hoc refactoring strategies. The perception of participants on the prevalence and severity of issues suggests a potential gap in the practitioners' ability to identify these issues automatically. This gap likely contributes to underestimating their prevalence, underscoring the need for automatic approaches in detecting and addressing AITD issues. Based on our findings, we eventually come up with a set of challenges that the research community is called to address, other than additional implications that may impact future research actions on AITD.

Structure of the chapter. In Section 3.2, we describe the research questions driving our investigation and the methodical choices to address them. The analysis of the results is reported in Section 3.3.

Section 3.4 reports on the limitations of our work and the actions conducted to mitigate them.

3.2 RESEARCH METHOD

This section defines research objectives, outlines the rationale, and explains each step we performed.

3.2.1 *Research Objective and Questions*

Our empirical investigation analyzed practitioners' prevalence, effects, and mitigation strategies when dealing with AI Technical Debt (AITD) issues, i.e., suboptimal or compromised design and implementation choices made during the development and deployment of AI-enabled systems. The focus on the set of AITD issues that strictly affect the code and the architecture was motivated by the low research effort conducted on these two aspects and the high negative impact of gaining low quality on the architecture and the code stated by Shivashankar *et al.* [164] and Lenarduzzi *et al.* [88].

To pursue our investigation, we first applied the Goal-Question-Metrics approach [12] to extract relevant research questions that could address our objective. More specifically, the goal of our study was the following:

> **Our Goal. Analyze** AITD issues
> **for the purpose of** identifying
> **their** frequency, severity, impact, identification, and mitigation strategies
> **from the point of view of** AI practitioners
> **in the context of** building AI-enabled systems.

From the definition of the main goal, we defined the following research questions:

> **RQ1:** How frequent are AITD issues affecting the code and architecture from the practitioner's perspective?

The motivation behind studying the frequency of AITD issues from the practitioner's perspective is to gain insights into the prevalence of these issues in software development. While these issues are defined and discussed in the literature, it is still challenging to understand the actual presence in the state of the practice. By understanding how frequently practitioners encounter AITD issues, we can prioritize and address the most significant challenges. $\mathbf{RQ}_1$ helps devise strategies to mitigate risks, allocate resources effectively, and develop proactive measures. Investigating the frequency of the nine aforementioned AITD issues could provide valuable insights into real-world challenges and aid decision-making processes in software development.

Moreover, the set of issues proposed by Sculley et al.[161] presents different granularities to capture their presence. For example, Undeclared Consumers inspection requires system-level visibility, where the intrusion of unregistered access is possible only by accessing through API requests. Therefore, analysis of AI-enabled systems to capture the frequency of AITD issues is limited when examining the code and architecture statically. Moreover, while these issues have clear definitions that help to understand how they can appear in practice, metrics and tangible properties to observe them are not established. Therefore, this study aims to capture characteristics of these issues directly from practitioners. Their experience in building AI-enabled systems includes insights that are not easily extractable from the code.

> **RQ2:** *How severe are AITD issues affecting the code and architecture from the practitioner's perspective?*

Understanding the severity of AITD issues from the practitioner's perspective is crucial for effectively prioritizing efforts and addressing challenges. $\mathbf{RQ}_2$ helps allocate resources toward resolving the most critical

AITD issues and improving the overall quality of the development process. Additionally, assessing the severity of these nine AITD issues fosters a culture of continuous improvement by creating awareness and promoting collaborative efforts to address these challenges proactively.

> 🔍 **RQ3:** What is the impact of AITD issues affecting the code and architecture from the practitioner's perspective?

RQ$_3$ focuses on assessing the impact of introducing AITD issues from the practitioner's perspective. After assessing the severity of each AITD issue, **RQ**$_3$ aims to increase the awareness of the practitioners encountering AITD issues, highlighting the specific quality aspects influenced or affected by these issues. This analysis is crucial for gaining insights into the potential risks, challenges, and trade-offs associated with AITD.

> 🔍 **RQ4:** What strategies do practitioners use to identify and mitigate AITD issues affecting the code and architecture?

The last research question aims at finding possible strategies to manage the proposed nine AITD issues. The set of discovered strategies could inspire automated techniques for AITD management, creating an ideal set of guidelines that helps practitioners face AITD issues while building AI-enabled systems.

Altogether, the findings from the four **RQ**s would inform the software engineering research community about the AITD types that are more relevant in practice, their impact on multiple properties of AI-enabled systems, and how practitioners currently deal with them. These pieces of information may be exploited by researchers to increase awareness on the matter and to prioritize their research efforts, e. g., by investigating the AITD having the highest impact on practice, other than to devise novel, practitioner-informed techniques and methods that would support the identification and mitigation of AITD.

3.2.2 *Research Method*

As Easterbrook *et al.* [38] suggests, several research methods can be used to gain insights into AITD issues. We opted for a survey study to collect data from a larger sample of participants and understand the behavior of AITD issues in practice. We aimed to collect comprehensive information from a large sample of participants, particularly practitioners with experience in artificial intelligence systems, to improve our understanding of AITD issues and their impact on the development process. Our survey study explored the practitioner's perspective on each AITD issue, analyzing its frequency, severity, impact, and management strategies. We adopted Kitchenham and Pfleeger's survey design principles [79] to structure our survey process, comprising multiple steps. We initiated the survey process by extracting the study context (Section 3.2.3) to identify and select a set of AITD issues for analysis. Subsequently, we designed the survey structure and conducted a pre-screening (Section 3.2.4). To ensure its reliability, the survey underwent validation using empirical standards and a pilot test (Section 3.2.5). Finally, we executed and administered the survey (Section 3.2.6), collected the data, and performed data cleaning and analysis to address our research questions (Section 3.2.7). Detailed information about the conducted survey is provided in the following sections.

3.2.3 *Context of the Study*

The *context* of the study was represented by *objects*, i.e., the AI-related issues in the scope of the study, and *subjects*, i.e., the participants of the survey. As for the objects, our research focused on identifying potential AITD issues affecting the code and the system's architecture. We thoroughly analyzed the existing literature on the topic of explicit issues that can introduce technical debt specific to AI-enabled systems, including studies by Sculley *et al.* [161], Gesi *et al.* [54], and Bogner *et al.* [18]. From the initial set of 77 AITD issues collected from the three sources, we selected all the issues related to the architecture and the code, identifying 13 issues. We

chose issues with the most straightforward definitions, as many of the AITD problems we encountered were quite broad and difficult to pinpoint to a specific scenario. For example, we included *Pipeline Jungle* as it clearly indicates the affected object (e. g., the pipeline), the event (e. g., continuous growth), and the effect on the system (e. g., decreased maintainability of the pipeline). However, we excluded any AITD issues identified in the literature that lacked these specific elements (e. g., "Monolithic Pipeline"). The motivation behind this selection is twofold. On the one hand, the amount of data possible to collect from each participant can be limited by the participant's availability time. It is crucial to strike a balance and avoid making the survey too overwhelming, as this can lead to participants losing interest or abandoning the survey. On the other hand, it is imperative to ensure that any AITD issues in the survey are clearly defined so participants can accurately recognize and respond to them. This is especially critical during the vignette representation step, as unclear issues can result in biased answers based on a misrepresented situation of the vignette. As a result of this analysis, we collected nine AITD issues affecting the code and the architecture, as stated in Table 3.1

As for the *subjects*, our research focuses on gathering perceptions about issues that appear in AI-enabled systems, specifically in the code and the architecture. To obtain a comprehensive overview of the perception of AITD, we tried to gather insights from diverse perspectives in our participant selection process. We aimed to gain practical insights into the AI landscape, engaging with professionals actively involved in building AI-enabled systems. Within this process, distinct roles are assigned with unique priorities and goals, and individuals approach the AI component with varying levels of expertise — from model architecture experts to data scientists with limited experience in this field. Consequently, this study's subjects are AI practitioners with experience interacting with AI components. The eligible participants for this study can be involved in different AI components, such as data preparation, model training, model deployment, and application maintenance. Therefore, we defined the subject of this study as practitioners with a baseline understanding of AI, but

that includes different roles and perspectives among the several facets an AI-enabled system includes. For this reason, participants followed a rigorous pre-screening survey. From the answers of this part of the survey, we collect information about the eligibility of the participants and their background differences.

3.2.4 *Survey Design*

The survey was structured following the principles defined by Kitchenham and Pfleeger [81]. We also followed the recommendations provided by Reid *et al.* [151], which allowed us to increase the quality of the applied method and manage threats related to the participants' knowledge. In particular, we applied the following recommendations:

- We introduced a pre-screening survey to check the skill level of the participants.
- We added a skill and attention check question to check whether participants who self-report the requested skill meet our criteria (labeled in the pre-screening survey as PD12).
- We included examples with vignettes to increase participants' confidence in conducting our study.
- We included a pilot test to check the comprehensibility and clarity of the survey.

The participants' information collection was divided into two parts, as shown in Figure 3.1. The first part of the research focused on selecting participants with experience developing AI-enabled systems by pre-screening.

PRE-SCREENING In this part, we provided each participant with the survey introduction information that describes the role of the participants eligible for the study's context. In detail, we explicitly required that the participants' roles be related to artificial intelligence. Given the diverse

Table 3.1: Definitions of the collected AITD issues

AITD Literature Selection	
AITD Issue	Issue Definition
Jumbled Model Architecture (JMA)	Model architecture components are cobbled together and difficult to understand and maintain individually [54].
Pipeline Jungle (PJ)	These pipelines organically evolve as new signals are identified and new information sources are incrementally added. Without care, the resulting system for preparing data in an ML-friendly format may become a jungle of scrapes, joins, and sampling steps, often with intermediate file output. Managing these pipelines, detecting errors, and recovering from failures are all difficult and costly [18, 161].
Multiple Language Smells (MLS)	It is often tempting to write a particular piece of a system in a given language, especially when that language has a convenient library or syntax for the task at hand. However, using multiple languages often increases the cost of effective testing and can increase the difficulty of transferring ownership to other individuals [18, 161].
Undeclared Consumers (UC)	Often, a machine learning model m_a prediction is made widely accessible at runtime or by writing to files or logs that other systems may consume later. Without access controls, some of these consumers may be undeclared, silently using the output of a given model as an input to another system [18, 161].
Correction Cascades (CC)	There are situations in which a model m_a for a problem A exists, but a solution for a slightly different problem A' is required. In this case, learning a model m'_a that takes m_a as input and learns a small correction as a fast way to solve the problem can be tempting. However, this correction model has created a new system dependency on m_a, making it significantly more expensive to analyze improvements to that model in the future. The cost increases when correction models are cascaded, with a model for problem A'' learned on top of m'_a, and so on, for several slightly different test distributions [18, 161].
Glue Code (GC)	ML researchers tend to develop general-purpose solutions as self-contained packages. Generic packages often lead to a glue code system, in which a massive amount of supporting code is written to get data into and out of general-purpose packages [18, 161].
Deep God File (DG)	A file containing multiple components of an AI system, e. g., model training, validation, and testing [54].
Scattered Use of ML Libraries (SML)	Scattered use of ML API in multiple files. Once ML API needs to be modified, the practitioners must modify places across several files [54].
Unwanted Debugging Code (UDC)	A debugging code fragment, method, or class is no longer used, but it is still part of the source code [54].

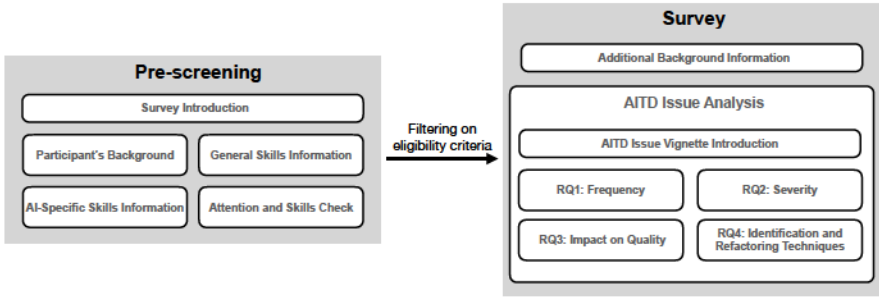


Figure 3.1: Structure of the pre-screening and the survey

interpretations of AI, practitioners may develop their understanding from various contexts, such as different tasks (e. g., classification or regression) and techniques (e. g., Machine Learning or Deep Learning). Therefore, we built a pre-screening survey that involves participants who work in all the AI-related fields, ensuring a baseline understanding. The pre-screening consisted of two essential sections of questions. In the first section, we asked participants about their role in their company and their level of experience in general skills related to building AI-enabled systems: Programming, Artificial Intelligence, and Software Engineering. The questions about the experience are presented on a Likert Scale, where the participant can explain his knowledge level on a 5-point scale (i.e., Novice, Basic, Intermediate, Advanced, Expert) [112]. The second section included questions to understand which part of an AI-enabled system the participant is responsible for. We proposed a sample of the AI pipeline to let the participants recognize the phases in which they are involved. Using the five-point scale, each participant reported their experience with the practices included in an AI application, starting from the data preparation phase to the model deployment phase. Additionally, we asked which activities explain their work. Finally, we included a competence and attention check question related to the practice used to build an AI-enabled system; In particular, we asked the participant which data balancing techniques should be applied in a specific context. Table 3.2 gives an overview of the questions delivered to the participants. Afterward, we defined a set of constraints in the pre-screening process to focus on the answers from

participants who considered having experience in the domain. The participant, to be considered appropriate for the survey, must respect the following constraints:

1. The participant must correctly answer the competence and attention check question (PD12).
2. The participant must know Programming at least equal to two out of five.
3. The participant must know about artificial intelligence (PD4) or software engineering (PD5) at least equal to three out of five.
4. If the participant respects the previous constraint, the participant must know the other field at least equal to two out of five.
5. The participant must know about monitoring, maintenance, and deployment of AI models (PD9) or knowledge in using machine learning libraries (PD10) at least equal to three out of five.

These criteria prioritize participants with expertise in systems, including software engineering, programming, and artificial intelligence. Moreover, a focus on participants' proficiency in managing various phases of AI-enabled systems has been incorporated. These phases include monitoring, development, and deployment. Proficiency in these areas signifies participants' practical experience in handling the complete lifecycle of AI-enabled systems. By utilizing these criteria, we have identified a group of participants who can effectively analyze issues and technical debt. Their practical experience and knowledge in AI-enabled systems could let the participants understand the situations represented through the vignette and relate experiences to the AITD issues proposed.

After selecting the participants, we conducted the main survey, which was structured in two sections.

MAIN SURVEY First, we asked two questions related to the dimension of the team and the organization in which practitioners work to have a

Table 3.2: Prescreening questions

First Section		
Questions		Question Types
PD1	What is your role in the company?	Multiple Choice
PD2	How many years of experience do you have in your role?	Multiple Choice
PD3	Define your knowledge level in: Programming	Likert Scale
PD4	Define your knowledge level in: Artificial Intelligence	Likert Scale
PD5	Define your knowledge level in: Software Engineering	Likert Scale
Second Section		
Questions		Question Types
	Considering the section figure, I have experience in [...]	
PD6	[...] Data ingestion, data aggregation or other types of data preparation	Likert Scale
PD7	[...] Model selection and model training	Likert Scale
PD8	[...] Model validation, review process and improving model	Likert Scale
PD9	[...] Monitoring, maintenance and deployment plan	Likert Scale
PD10	[...] Using Machine Learning libraries and execution of the model	Likert Scale
PD11	Taking a look at the section figure represented above, Which of these activities explains better your work in AI systems? (One or more of it)	Multiple Choice
PD12	In which part of the dataset do you apply the data balancing techniques (e. g., Oversampling, Undersampling)?	Multiple Choice

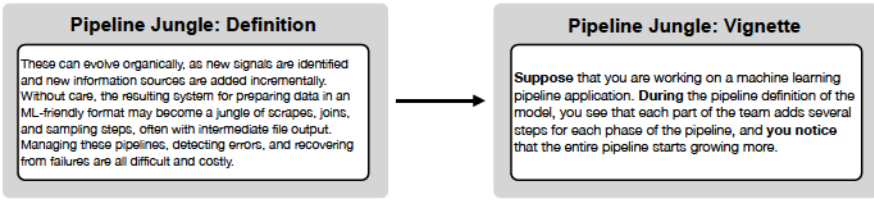


Figure 3.2: Application of vignettes to define the Pipeline Jungle issue

complete vision of their activities while developing AI-enabled systems. Then, we asked a set of questions to answer the research questions for each instance analyzed by the participants. In detail, the second section started with the representation of the vignette to let the participant recognize the issue of the situation and analyze it with a practical example. We followed the methodology of *Experimental Vignette Study* introduced by Atzmüller and Steiner [9] to create the vignette in a standardized, concise and precise format and increase the level of understandability of the survey from the practitioner's perspective. This method transforms the formal definition of an AITD issue into a more detailed and intelligible form through a vignette. We created a vignette for each instance to ensure that participants understand the context, activity, and problems associated with AITD entirely. These vignettes, as demonstrated in Figure 3.2, depict situations where AITD issues may arise, considering the timing of the situation, the event, and the actions of the practitioners.

By presenting issues related to AITD in the form of *situations*, participants can reason about the circumstances surrounding AITD issues rather than just its definition. This approach allows participants to provide information on the characteristics of AITD issues in a relatable and easy-to-understand way, based on common ground and mitigating the risk of diverse interpretations of an issue.

After understanding the issue through the vignette, each participant answered questions on the characteristics that address our RQs. We designed several questions to analyze the frequency and the severity of AITD issues ($\mathbf{RQ}_1$ and $\mathbf{RQ}_2$). In particular, we asked the participant (i) how often they had encountered an instance of such AITD in their experience, (ii) to

what extent and (iii) why this situation is problematic, and (iv) the effort needed to identify and refactor the AITD issue. To understand the impact of AITD issues ($\mathbf{RQ_3}$), we asked several questions based on six AI quality-related aspects: (i) understandability, (ii) performance, (iii) maintainability, (iv) evolution, (v) defect-proneness on other components, and (vi) coupling. The participants provided the agreement level on the statement proposed as a Likert Scale (Strongly Disagree to Strongly Agree). Finally, after analyzing each assigned AITD issue through the vignette proposed, the participant could provide further information on AITDs through the open question and release his email for future investigations. Table 3.3 illustrates the list of questions delivered to each participant.

3.2.5 *Survey Validation*

Before starting with the survey study, we conducted a pilot test recruiting five researchers experienced in performing online surveys, as proposed by Reid *et al.* [151], to assess the survey's quality in various areas, such as the clarity of the questions and the accuracy of the answers.

We considered the knowledge gap between the researchers in the pilot test and the AI practitioners involved in the survey; therefore, we included an industry participant with experience in AI-enabled systems. Specifically, we included a Backend Software Engineer in AI-enabled systems who works as a practitioner for feedback. The addition of an industry-related participant for the pilot test helped us to understand if the terms used in the survey could be understandable by the participants subject to the investigation since the focus of our study is to collect the perspective of practitioners. Several suggestions helped to increase the quality of the survey study. Regarding the pre-screening questions, participants suggested increasing the detail of scale values of the questions related to the general skills (PD3, PD4, PD5) and AI-specific skills (PD6-PD10). From this feedback, we defined values for general skills to range from Novice level to Expert and experience level in AI-specific skills from Very Poor to Excellent. Regarding the main survey, participants highlighted a lack of clarity for the

Table 3.3: Survey questions

Preliminary Questions		Question Types
S1D1	How many members are in your team?	Multiple Choice
S1D2	How many employers are in your company?	Multiple Choice
RQ1		Question Types
S2D6	How often do you encounter this situation?	Likert Scale
RQ2		Question Types
S2D1	How much do you find this situation problematic?	Likert Scale
RQ3		Question Types
S2D2	Why it is (or not) problematic?	Open Question
Please vote on the following sentences: I think that this situation ...		
S3D1	...may make other components of the application more defect-prone.	Likert Scale
S3D2	...makes the level of interpretability/explainability of the model more complex.	Likert Scale
S3D3	...makes it difficult to evolve the application.	Likert Scale
S3D4	...may impact and decrease the performance of the model.	Likert Scale
S3D5	...could contribute to create dependencies, increasing the coupling of the application.	Likert Scale
S3D6	...is hard to maintain, increasing the cost and the effort to maintain the application.	Likert Scale
RQ4		Question Types
S2D3	What are your practices of identification for this situation?	Likert Scale
S2D4	Could you tell us something more about identification?	Open Question
S2D5	How much effort do you think is necessary to identify this situation?	Likert Scale
S2D7	How much effort do you think is necessary for refactoring, mitigating, or improving this situation?	Likert Scale
S2D8	How do you refactor this situation?	Open Question
Additional Questions		Question Types
S4D1	Could you give us any additional feedback on your activity or any bad practices you've observed in your experience? *	Open Question
S4D2	If you'd like to be considered for future surveys or interviews, please enter your email address.*	Open Question
*= The proposed question is optional.		

vignette representing AITD issues, particularly for *Jumbled Model Architecture* and *Glue Code*. To address this issue, we rewrote the vignette, adding examples that help practitioners recognize the situation. Finally, participants suggested removing questions related to gender. Although gender is an important and relevant factor in various research contexts, our study of AI Technical Debt focuses specifically on technical aspects of AI and its implications. Furthermore, avoiding questions on gender ensures that participant privacy is respected and that the study remains focused on its core research objectives without inadvertently collecting sensitive or unnecessary personal information. For these reasons, we agreed with the

participants to exclude gender-related questions that could hinder their privacy.

3.2.6 *Survey Administration*

One of the key choices was selecting the platform to recruit the survey participants. Social media or a specialized recruitment platform could have been a reliable option. However, it is necessary to be cautious about potential biases in the sample, as social media users may not represent the target population accurately. Opting for a specific recruitment platform like Prolific can be beneficial due to its focus on academic research and access to pre-screened, willing participants who have expressed interest in participating in studies. Furthermore, these platforms often offer features to control demographics and manage participant compensation, simplifying the research process.

We recruited the participants through the Prolific platform[1], which allowed us to select experts in AI-enabled system development. We could control the recruitment process by defining filters that explicitly address the participants' research on specific domains. Given the absence of specific filters precisely aligned with the role we sought, we used filters focused on participants with expertise in software development techniques and computer programming.

The process of pre-screening analysis began on June 28, 2022, and lasted for nearly four hours to obtain answers from all participants. With these filters, we recruited 500 participants with experience in programming and knowledge of software development techniques. From the initial set, 188 participants correctly answered the pre-screening question, i. e., competence and attention check question, representing 37,8% of the selected population. We excluded those participants who did not meet the previously defined eligibility criteria and appointed 112 participants as eligible for the experiment. Afterward, some participants declared their unavailability to continue the experiment, leading to a group of *54* participants.

[1] https://www.prolific.co

Table 3.4: Task assignment

Participant	AITD Vignette
Group P1	Subsets A+B
Group P2	Subsets B+C
Group P3	Subsets A+C

This drop in participants could lead to insufficient information extractable from each participant. We had to consider the maximum time and effort a participant could spend on the survey and the amount of information that could be extracted. The between-subject approach, i.e., querying each participant about only a single AITD issue, could result in limited information for each AITD issue. The within-subject approach, i.e., asking participants questions about nine AITD issues, could expose the threat of survey fatigue, leading to careless answers or abandoning the survey. Because of this constraint, the survey is built upon the *mixed-subject design* approach [9], asking each group of participants questions about two different subsets of vignettes established beforehand.

In particular, we defined three subsets of vignettes describing AITD and three participant groups. The representation of the subset of the vignette was the following:

- Subset A: Glue Code, Multiple Language Smell, Undeclared Consumers.
- Subset B: Pipeline Jungle, Scattered Use of ML Libraries, Correction Cascades.
- Subset C: Jumbled Model Architecture, Unwanted Debugging Code, Deep God File.

Table 3.4 illustrates the definition of the subsets and their combination for the survey. Each group participant answered questions about six issues that can cause AITD. Using the mixed-subject design approach, each participant provided answers for six AITD issues, allowing us to obtain more answers for each AITD.

The main survey was conducted on August 3, 2022, and we needed two days to collect answers from all participants. Starting from 54 participants' answers, we collected 38 answers related to the issues of Group A, 36 answers related to Group B, and 33 answers related to Group C.

3.2.7 *Data Analysis*

Once the data were entirely collected from the participants, we manually filtered the answers from those who did not take the survey seriously; this step excluded one answer. As a result, we collected 53 answers, decreasing the number of answers for Group A to 37. The first type of question included multiple choice questions or in Likert Scale format. Such data in an ordinal scale was analyzed using non-parametric statistics, as proposed by Briand *et al.* [23]; specifically, we analyzed the distribution and the frequency of each value. The second type of question included open answers, which helped identify other quality-related aspects affected by the introduction of AITD issues and highlighted the solutions practitioners apply to refactor the issues. Such open questions were analyzed following the *iterative content analysis* method [195]. This qualitative analysis process, served to extend the set of quality-related aspects that we initially proposed to participants. By embracing the diverse perspectives of AI practitioners, we aimed to uncover a comprehensive spectrum of quality-related facets impacted by AITD issues including factors that analyze the system on a broader perspective, including all the factors related to quality that can be affected ($\mathbf{RQ_3}$). In addition, we use this process to examine useful refactoring techniques ($\mathbf{RQ_4}$). By adopting this approach, we ensure that our analysis includes all innovative and context-specific solutions that practitioners use, enhancing the practicality of our findings. In particular, we performed the following steps:

1. **Micro-Analysis**: The first manually analyzed the answers related to the open questions. The keywords of each answer are extracted and processed to obtain a label that identifies the concept. To avoid the subjectivity introduced by the first author, the second and third

authors assessed the quality of the label assignment. They manually analyzed the label of each open answer and provided feedback and suggestions to improve their suitability (e. g., merging two semantically similar labels). This phase ended by defining an agreement status between the label proposed by the first author and the reviewers' feedback.

2. **Categorization**: The analysis continued by categorizing the labels that describe similar concepts and splitting those explaining more than one concept into several categories. This step allowed for isolating information extracted from a single response and was conducted through an iterative process that allowed for a gradual and more precise grouping of labels.

3. **Saturation**: The categorization was performed until reaching saturation, observable when an iteration does not perform any changes in the set of categories.

Finally, we manually analyzed the answers to the question that asks the participants additional feedback, represented in question S4D1 of Table 3.3.

3.3 ANALYSIS OF THE RESULTS

This section reports the results of our survey, including the preliminary study and the analyses concerning the prevalence, severity, and impact of AITD, as well as the management strategies to mitigate its effects.

3.3.1 *Preliminary Analysis*

The study participants were evaluated based on specific criteria to gain insight into their work environment. The selection process of participants was based on their role, which was then analyzed. The results, as shown in Figure 3.3, reveal that the majority of the participants were software

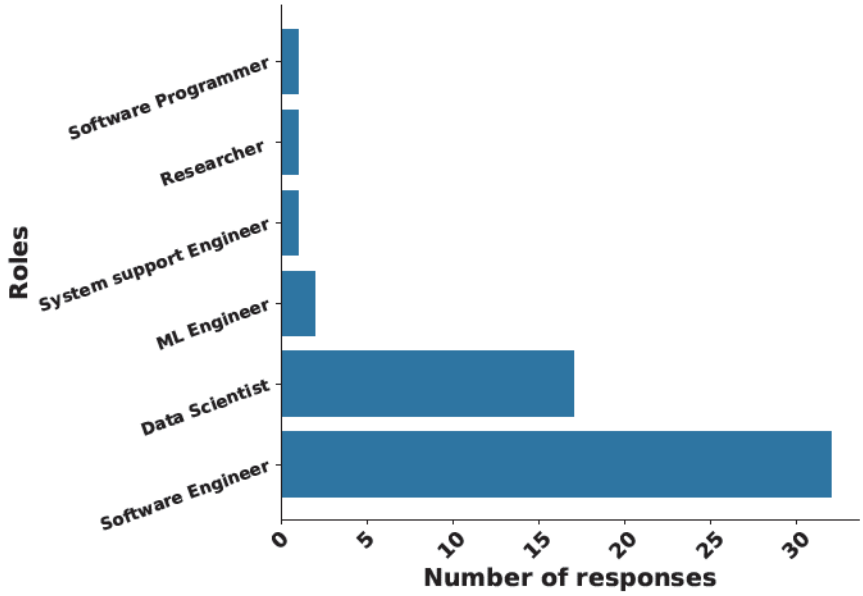


Figure 3.3: Distribution of the roles of the participants

engineers (59.3%) and data scientists (31.5%). However, the participants also included professionals in related fields, such as AI-enabled system management and development, including researchers, system support engineers, and software programmers. These findings suggest that the study captured perspectives from professionals involved in multiple aspects of AI-enabled systems.

As shown in Figure 3.4, a significant number of the surveyed participants, 24 out of 53, are employed in small companies with fewer than ten employees. The remaining participants are from medium- or large-scale organizations with more than ten employees. Most participants work in small teams, i. e., 41 in units of less than ten members, as depicted in Figure 3.5. Despite most participants being from small companies and teams, the collected data includes practitioners' perspectives from large companies and teams, with 13 participants working in a company with more than 100 employers and six working in teams with more than 20 members.

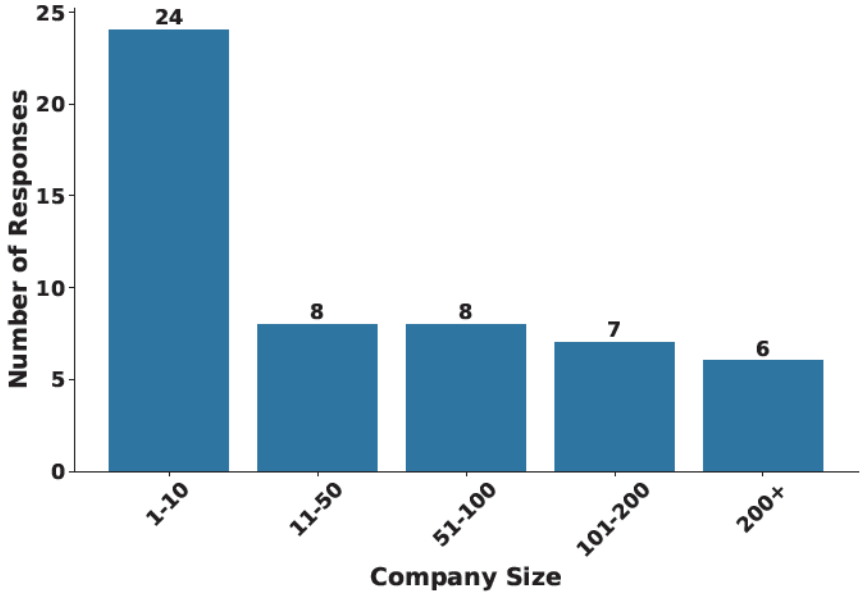


Figure 3.4: Distribution of the company sizes of the participants

The participants' skill levels were evaluated based on their knowledge of AI-related fields, as shown in Figure 3.6. 78% of the participants reported having a good understanding of AI, rated with three or higher out of five. The remaining 22% rated their AI knowledge as two out of five, but had strong Programming and Software Engineering skills. As for Software Engineering skills, 92% of the participants rated them with three or higher out of five. Lastly, 65% of the participants indicated advanced programming skills, with 30 rating them with four or higher out of five.

Finally, we analyzed the ML-Specific related skills for each stage of an ML pipeline by using ML libraries to build an ML application. The results depicted in Figure 3.7 highlight the notable proficiency of a significant proportion of survey participants in various machine learning-specific competencies. Remarkably, the findings demonstrate that most respondents (87%) exhibited sufficient proficiency in Model Deployment, with a similar trend observed for using ML Libraries. Participants who reported expertise in these two areas of the ML pipeline also demonstrated com-

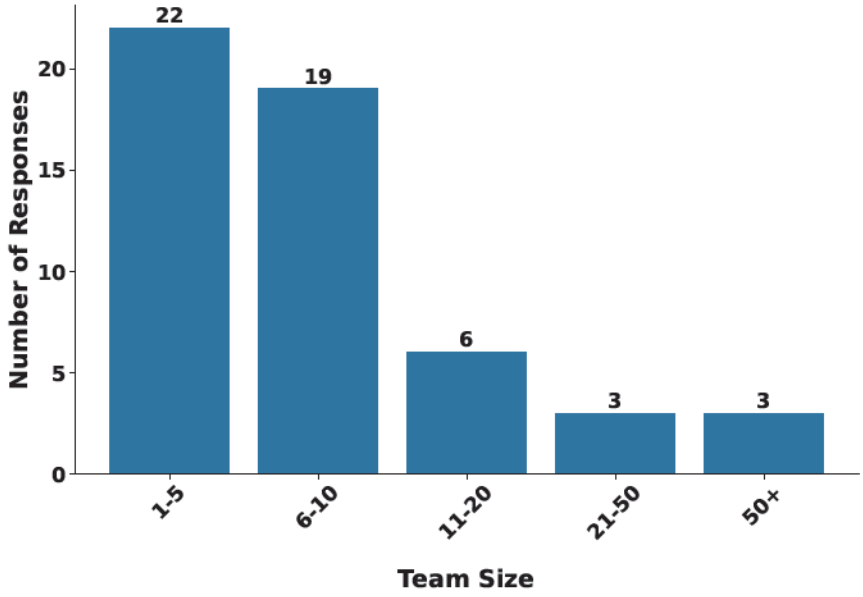


Figure 3.5: Distribution of the team sizes of participants

petencies in Data Preparation, Model Training, and Model Evaluation, with a considerable proportion (91%, 87%, and 89%, respectively) exhibiting adequate proficiency. Notably, most participants exhibited at least intermediate-level skills in these competencies. The competence and attention check question results complemented these outcomes, enabling accurate evaluation of participant eligibility for the survey.

3.3.2 RQ_1. *How Frequent Are AITD Issues from the practitioner's Perspective?*

We collected the practitioners' answers regarding the prevalence of AITD issues. Figure 3.8 displays the distribution of answers on a Likert scale, measuring the frequency of AITD issues found by practitioners. The results indicate that the majority of practitioners involved reported "never" encountering specific situations attributable to the AITD issues presented, mainly for *Undeclared Consumers* (14 answers), *Correction Cascades* (14

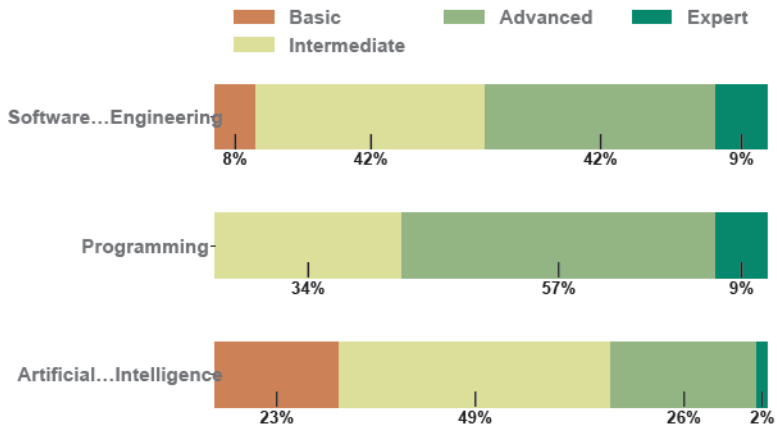


Figure 3.6: Distribution of the General skills of the participants selected for the questionnaire

answers), and *Multiple Language Smell* (14 answers). In addition, most participants reported "rarely" encountering issues of most AITDs, including *Undeclared Consumers* (15 answers) and *Scattered Use of ML Libraries* (17 answers). However, some participants reported encountering AITD issues "sometimes" in their experiences, such as in *Glue Code* (14 answers), *Jumbled Model Architecture* (10 *answers*), *Pipeline Jungle* (nine answers), and *Scattered Use of ML Libraries* (nine answers). Finally, less than 10 participants frequently encountered AITD issues (denoted as "often" or "always"). In detail, the unmanageable growth of the AI pipeline that provokes *Pipeline Jungle* is frequently encountered by nine participants (in which six denoted it as "often" encountered and three as "always" encountered). Other AITD issues are reported to be frequently encountered by seven participants or less. These findings highlight that while most participants reported rarely or never encountering AITD issues, a small portion of respondents have already experienced these challenges in their AI development experiences. Therefore, this study catches information on the properties of AITD issues from two types of AI practitioners, classified as follows:

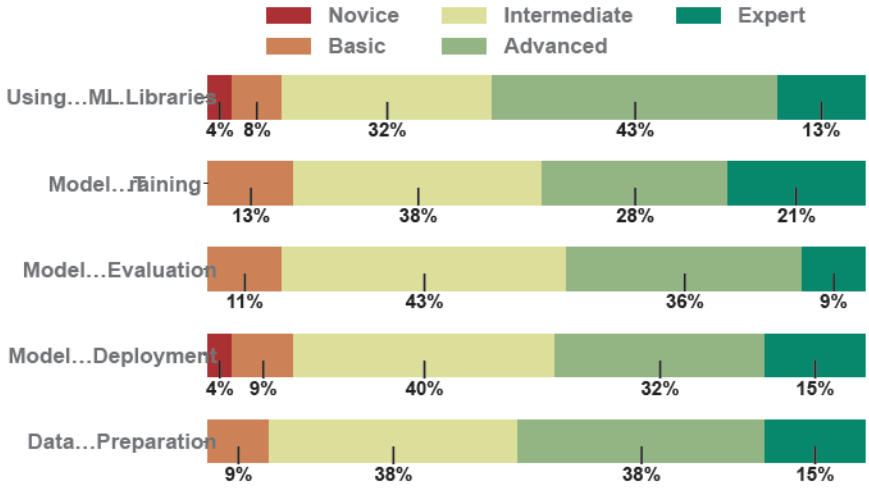


Figure 3.7: Distribution of the Machine learning specific skills of the participants selected for the questionnaire

- AITD unfamiliar practitioners: practitioners that never or rarely encountered the specific AITD issue.
- AITD familiar practitioners: practitioners that sometimes, often, or always encounter the specific AITD issue.

The analysis described in the following sections describes the results first from a general perspective of the perception of AITD issues. Subsequently, the results are discussed based on the familiarity level of the participants. It is essential to note that the number of participants in the two groups varies based on the answers that participants gave on the prevalence of each AITD issue. In the article, we reported the global results highlighting the aspects of the AITD issues. Tables and plots grouped by familiarity level are available in our online appendix.[2]

[2]Online Appendix: https://doi.org/10.6084/m9.figshare.24030456

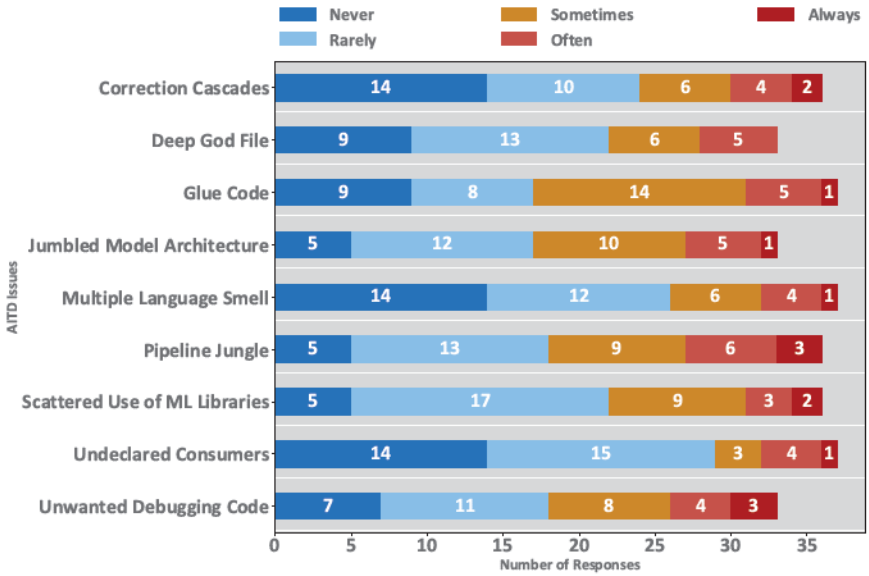


Figure 3.8: Prevalence of AITD issues in AI-enabled systems from the practitioner's perspective

3.3.3 RQ_2. *How Severe Are AITD Issues from the practitioner's Perspective?*

Considering the severity levels provided by the participants, six AITD issues out of 9 analyzed are reported to be "very problematic" or "extremely problematic" by participants. Almost all participants (30 out of 37) declared the issue of Undeclared Consumers to be of high severity. The leading cause that emphasizes the severity of *Undeclared Consumers* is free access to the model. The data collected from unknown customers is essential information used to understand the system's architecture and try to exploit it to perform malicious actions. Therefore, this AITD issue inside the system needs to be considered when analyzing possible unknown accesses. *Scattered Use of ML libraries* and *Deep God File* are reported to be highly severe by 27 participants. The uncontrolled increase of the size of the component and the scattered injection of libraries lead to an increase in the complexity of the class, which makes it challenging to maintain and understand the intended responsibility of the module. In detail, analyzing

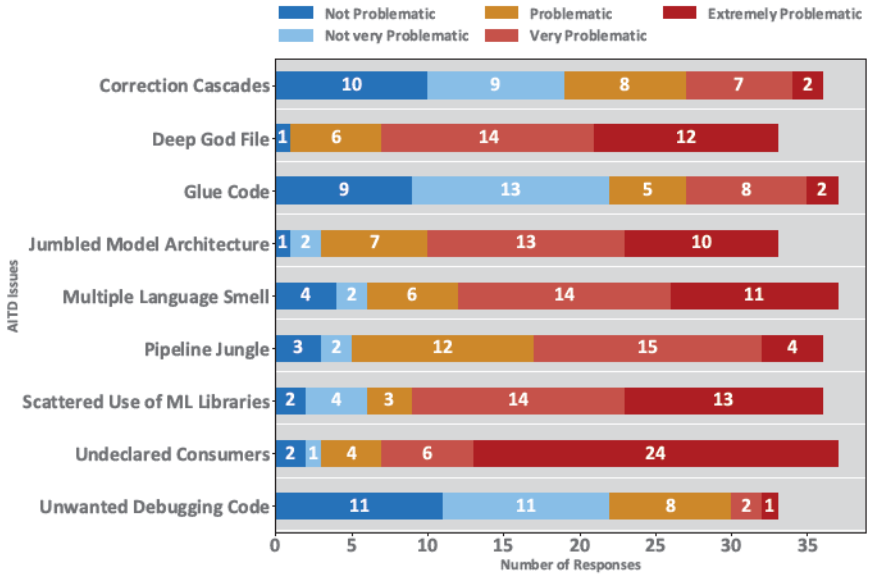


Figure 3.9: Severity of AITD issues in AI-enabled systems from the practitioner's perspective

Scattered Use of ML libraries, participant (P53) stated: *"Libraries need to be correctly understood. As the team is small and very specific, only particular and few libraries are involved. All team members must be aware and at least have some knowledge of the libraries and dependencies."*. Therefore, the increased complexity of the modules can also reflect damaging socio-technical factors.

Moreover, *Jumbled Model Architecture* and *Pipeline Jungle* are both reported as highly severe by most part of the participants, in which 23 participants denoted as "very problematic" for *Jumbled Model Architecture* and 19 for *Pipeline Jungle*. The continuous addition of steps and modules is a critical aspect that needs to be managed to avoid a critical explosion of complexity that can affect the whole system. In this context, a participant warned (P51): *"Once multiple steps or processes are added to each pipeline phase, it will require more time and developers to maintain its lifecycle. This increases the chances of a disorganized program and might delay the release date"*.

Finally, *Multiple Language Smell* has been reported to be highly severe by 25 participants. The main problem caused by this AITD issue is mainly related to the human resources involved in the lifecycle of the AI-enabled system. When multiple languages are introduced and used inside the system, only some practitioners have the skill to use a determined language. As the participants identified, operating could be challenging if some components are written in different languages.

On the other hand, most participants reported not all of the AITD issues analyzed were severe. Analyzing the answers that participants delivered on the severity related to *Correction Cascades* and *Glue Code*, answers are equally distributed towards the different levels of severity proposed, leading to a not clearly stated high severity of these AITD issues. However, some of the participants still noted the criticality of introducing these AITD issues. Introducing *Correction Cascades* issue could lead to having uncontrolled dependencies with external applications. This situation is critical if a change in the system could lead to undesired effects on the consumers' applications without the possibility of analyzing the consequences, as stated by Participant P16. Finally, *Unwanted Debugging Code* was reported to be low severe by 23 participants. While a few participants highlighted the problem of showing sensitive data through logging, most noted that this AITD would not significantly affect the system.

Subsequently, we analyzed practitioners' perceptions of the severity, considering their familiarity levels. The answers between the two groups do not present remarkably different severity levels for seven out of nine AITD issues, delineating that the severity perception denoted in the results is agreed upon by AITD familiar practitioners and AITD unfamiliar practitioners. However, some differences are engaging in two of the nine AITD issues analyzed. *Pipeline Jungle* has been denoted by both groups to be highly severe but with a particular remark analyzing *AITD familiar practitioners*. In detail, 11 participants out of 18 denoted this AITD issue as "very problematic" and one as "extremely problematic". Therefore, the severity of this AITD issue is strongly remarked on by practitioners who have experienced situations in which they encountered this problem. The

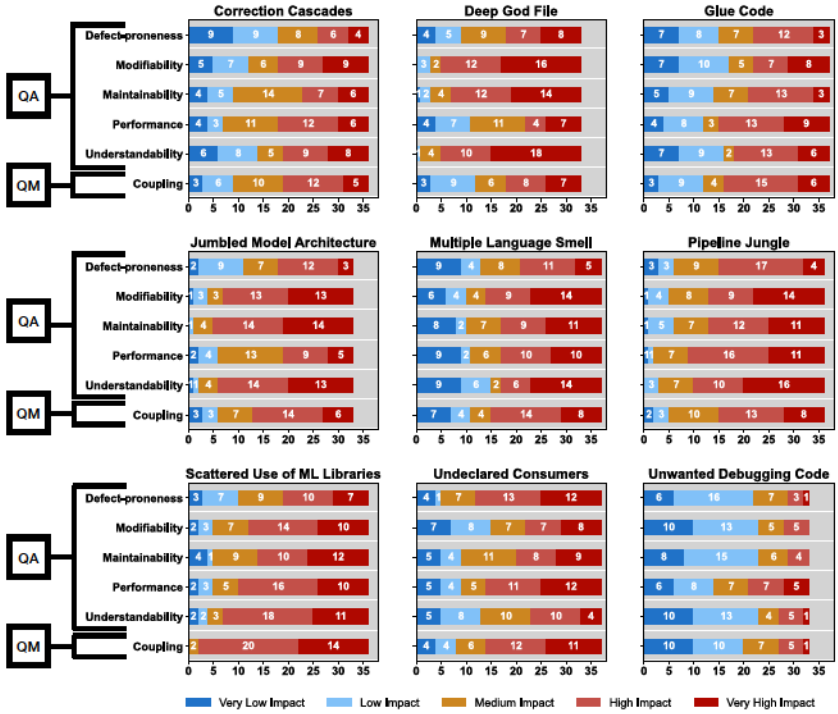


Figure 3.10: Effect of AITD issues on the selected quality aspects from the practitioner's perspective. Quality-related aspects are organized into three main categories: Quality Attributes (QA), Quality Metrics (QM), and Socio-Technical Factors (ST).

situation is similar when analyzing the answers for *Undeclared Consumers*, in which no *AITD familiar practitioners* reported low or medium severity of this AITD issue. Moreover, also *AITD unfamiliar practitioners* reported an extremely high severity for this AITD issue.

3.3.4 RQ_3. *What Is the Impact of Introducing AITD Issues from the practitioners' Perspective?*

To answer $\mathbf{RQ_3}$, we analyzed the quality related-aspects impacted by AITD, which could damage AI-enabled systems. First, we collected answers to the quality-related aspects and the distribution of the impact levels, as

illustrated in Figure 3.10. Second, we performed a content analysis to explore other quality-related aspects practitioners believe AITD could impact.

The content analysis process involved four iterations, each building upon the previous one to extract and categorize quality-related aspects affected by AITD issues. The fourth iteration identified the saturation state and the completion of the content analysis. In the first iteration, we organized 196 labels into 85 categories, providing an initial structure to the analysis. The second iteration narrowed the focus, isolating 31 categories directly related to quality-related aspects. Subsequently, in the third iteration, we identified 18 specific quality-related aspects that participants had highlighted. To illustrate this process, let us consider an example from the analysis of a participant's input (P42) in the context of the impact analysis of *Pipeline Jungle*. The participant mentioned "increased complexity leading to higher running costs and potentially harder debugging", and we extracted the label "Increased complexity and debugging difficulty" from this text. After extracting labels, we proceeded to categorize them. For instance, labels such as "Difficult to maintain" and "Difficulty of future maintenance operations" were grouped under a single category called "Maintainability issues" during this phase. Following these categorizations, we conducted an additional iteration to associate similar quality-related aspects into single categories. For instance, categories related to security, such as "Vulnerabilities", "Privacy Issues", and "Security", were grouped under the category "Security Issues". In summary, the four iterations of the content analysis process allowed us to progressively refine our understanding of quality-related aspects impacted by AITD issues, ultimately resulting in a structured and categorized set of insights.

Successively, we analyzed the relationship between each AITD issue and the quality-related aspects impacted. Figure 3.11 summarize all the categorization labels applied to quality-related aspects for each AITD issue. *Understandability* is the most impacted aspect, having a total of 53 answers, and seven out of nine of the AITD issues analyzed impact this aspect. Subsequently, the system's maintainability and security are strongly

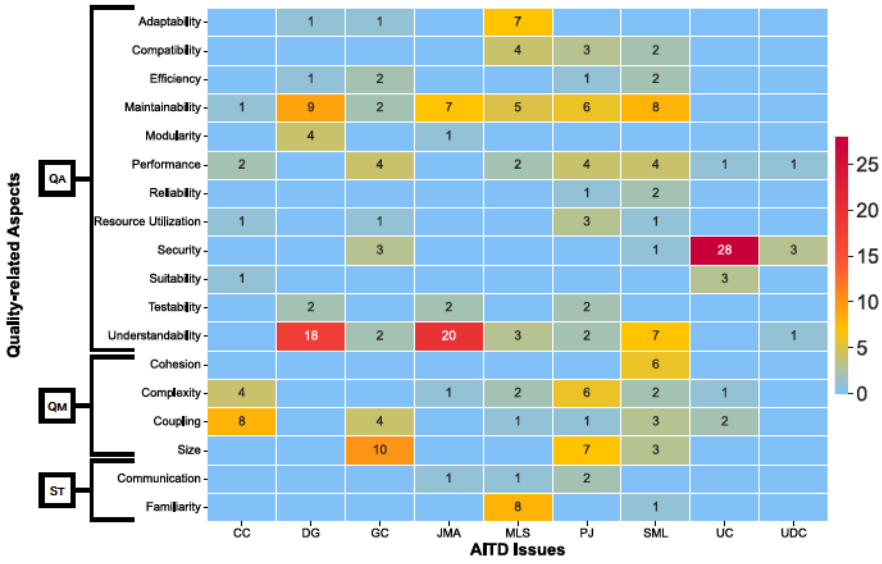


Figure 3.11: Frequencies of practitioners' answers on mainly-impacted quality-related aspects by AITD issues, categorized in quality attributes (QA), quality metrics (QM) and socio-technical factors (ST).

discussed with 38 and 35 answers, respectively, where security is mainly impacted by *Undeclared Consumers*. Finally, from the extracted aspects, eight of these are impacted by at least four AITD issues.

Additionally, Figure 3.11 specifies the number of answers of all the quality-related aspects impacted by each AITD issue. The number of answers highlighting the impact of the specified AITD issue is reported for each quality-related aspect collected. In the following, we analyzed deeper the relationship between AITD issues and quality attributes, quality metrics and socio-technical factors.

IMPACT OF AITD ISSUES ON QUALITY ATTRIBUTES Participants noted a significant impact of *defect-proneness* associated with almost all AITD issues, especially notable in the case of *Undeclared Consumers*, where 13 participants reported a "high impact", and 12 reported a "very high impact." Similarly, *Pipeline Jungle* was identified by 17 participants as having a high impact, with an additional four noting a very high impact.

Therefore, introducing AITD issues will likely increase the probability of defects. Specifically, participants emphasized this impact on AITD issues closely associated with the system's architecture, which they noted could increase architectural complexity. *Modifiability* is reported to be affected by nearly all AITD issues, particularly by *Deep God File* and *Jumbled Model Architecture*.

Consequently, participants indicated that these issues significantly complicate performing changes and maintenance operations, with 28 participants highlighting challenges with *Jumbled Model Architecture* when handling the *maintainability*. *Performance* is notably impacted by *Pipeline Jungle*, where 27 participants reported a high impact. This suggests that continually adding steps to an AI pipeline in an uncontrolled manner could substantially reduce system performance. *Performance* is also adversely affected by *Scattered Use of ML Libraries*. *Understandability* is impacted by more than half of the proposed AITD issues, indicating that various factors related to these issues limit practitioners' ability to analyze and comprehend the components of the AI-enabled system. Additionally, among the quality attributes reported as significantly impacted by AITD, three of the six proposed quality attributes were confirmed as impacted, along with nine additional attributes. *Understandability* emerged as the most affected attribute, with 53 mentions, and was impacted by seven of the nine AITD issues analyzed. System *maintainability* was also notably affected, identified in seven of the nine AITD issues with 38 mentions, including nine related explicitly to *Deep God File*. System security was another primary concern, denoted in total by 35 participants and particularly impacted by *Undeclared Consumers*, as indicated by 28 participants.

Considering the quality attributes leveraged exclusively by participants, eight of the extracted aspects were affected by at least four AITD issues. Regarding the additional quality attributes reported by participants, *Multiple Language Smell* was noted to impact *adaptability* and *compatibility* by seven participants. Moreover, other quality attributes are included and denoted to be affected by the presence of AITD issues, especially concerning the *testability* of the system and *Resource Utilization*.

IMPACT OF AITD ISSUES ON QUALITY METRICS Considering the impact of the first analysis represented in Figure 3.10, the coupling is significantly remarked as highly impacted in the presence of *Scattered Use of ML Libraries issue*, where 34 participants out of 36 denoted a high impact. Since this issue strictly creates dependencies between the system component and external libraries, it consequently increases the system's coupling. Considering the qualitative analysis of the quality-related aspects extracted by the participants, three participants confirmed this outcome as the main factor this AITD issue can cause. Moreover, participants noted an affection for AITD issues regarding the complexity and size of the system. In detail, the presence of *Glue Code* and *Pipeline Jungle* implies the increase in the system's size, as stated by ten participants and seven participants. Finally, a strong connection between the cohesion of the components inside the system is strictly affected by the presence of *Scattered Use of ML libraries*.

IMPACT OF AITD ISSUES ON SOCIO-TECHNICAL FACTORS Other than the quality attributes and metrics representing technical aspects of the AI-enabled system, participants leveraged the fact that the presence of these AITD issues can also affect socio-technical factors. Firstly, the presence of *Jumbled Model Architecture, Multiple Language Smell* and *Pipeline Jungle* are denoted to limit communication between team members. Interestingly, as stated by eight participants, the presence of *Multiple Language Smell* strongly affects the practitioners' familiarity with the system. Participants denoted that when a new component in a different language is added to the system's architecture, each future update and change will be more complicated if the practitioner faces more languages. Every practitioner working on a specific part of the system should have the knowledge to perform operations in two different languages. Specifically, a participant (P4) answered *"Because the two languages, or rather their input and*

output, may or may not be compatible and/or someone hired to maintain the code may or may not be able to code in both languages." The familiarity of practitioners in the system could be also affected by *Scattered Use of ML libraries*, as stated by a participant.

ADDITIONAL ANALYSIS ON AITD ISSUES IMPACT Analyses have been extended to consider the level of experience that participants have with AITD issues, collecting perspectives from both *AITD unfamiliar practitioners* and *AITD familiar practitioners*. The main outcome highlights that for three of the proposed AITD issues (*Scattered Use of ML libraries*, *Deep God File*, and *Unwanted Debugging Code*), there are no evident differences in the impact levels on the quality-related aspects as reported by both groups of practitioners. However, with the exception of *Unwanted Debugging Code*, practitioners from both groups recognize a critical impact on the six quality-related aspects we proposed. Interestingly, key differences were identified between these two groups. The high impacts observed in all the proposed quality-related aspects when considering the presence of *Pipeline Jungle* are mainly reported by *AITD familiar practitioners*. Specifically, none of the participants in this group reported a low impact for this AITD issue. Thus, the critical nature of this AITD issue is primarily recognized by practitioners who are aware of its presence and potential effects on the system. Similar patterns were observed when analyzing the impacts of *Deep God File* on the system's *understandability* and *Undeclared Consumers* on *defect-proneness*. However, among the nine proposed AITD issues, *AITD familiar practitioners* tended to report a higher impact compared to *AITD unfamiliar practitioners*, suggesting that the significant impact of these AITD issues is particularly acknowledged by practitioners aware of the consequences of these debts in the systems.

Analysis based on the familiarity level is also conducted for the quality-related aspects leveraged by the participants. In a broader context, *AITD unfamiliar practitioners* denoted 17 quality-related aspects impacted. In contrast, *AITD familiar practitioners* reported all the quality-related aspects extracted (18), adding *quality assurance* from the previous set. The

main differences between the two groups are notable regarding the number of responses for specific AITD issues. Regarding the quality-related aspects impacted by *Pipeline Jungle*, *size* and *performance* are almost denoted by AITD familiar practitioners. In contrast, *maintainability* is reported to be impacted by the same number of participants in the two groups (three participants for each group). The impact of *Glue Code* on the *performance* of the system is exclusively denoted by *AITD familiar practitioners*. *AITD unfamiliar practitioners* mainly denoted that this AITD issue impacts the *size* of the system (6 participants), also confirmed by *AITD familiar practitioners* (4 participants). *AITD unfamiliar practitioners* exclusively reported the impact of *Scattered Use of ML Libraries* to the *understandability*. AITD unfamiliar practitioners almost exclusively denoted the impact of *Correction Cascades* on the *dependencies* of the system, while only one participant in the AITD familiar practitioners reported it. The situation is analogous for *Deep God File* on *maintainability* and for *Multiple Language Smell* on the *familiarity*. All the other relationships in analyzing the impact of AITD issues on these quality-related aspects are reported in both groups but stated particularly from *AITD unfamiliar practitioners*, given a higher number of participants.

3.3.5 *Management Strategies*

We analyzed the identification strategies leveraging the question that asks the participants to define the set of practices used to identify the specified situations, as stated in question S2D3 of Table 3.3. Table 3.5 collects all the strategies for each issue. From the collected results, it is possible to see how manual inspection is the most used strategy for most issues. At the same time, for *Multiple Language Smell*, *Glue Code*, *Correction Cascades*, and *Unwanted Debugging Code*, more than 10 participants do not have an identification strategy for these issues. Some participants highlighted the need for a dedicated professional team to detect these problems inside the system, especially for *Undeclared Consumers* and *Pipeline Jungle*. Additionally, analyses based on the familiarity level of the participants are

Table 3.5: Identification strategies proposed by AI practitioners for the AITD issues

AI TechnicalDebt Issue	M.R.	A.I.	P.T.	N/A
Correction Cascades	**17**	1	5	13
Deep God File	**20**	4	2	7
Glue Code	**16**	4	3	14
Jumbled Model Architecture	**19**	4	4	6
Multiple Language Smell	**16**	2	4	15
Pipeline Jungle	**18**	5	7	6
Scattered Use of ML Libraries	**27**	2	2	5
Undeclared Consumers	**13**	6	9	9
Unwanted Debugging Code	**14**	5	2	12

M.R. = Manual Review
A.I. = Automated Inspection
P.T. = Professional Team
N/A = Don't Identify

conducted to understand the identification strategies identified by each group. For AITD familiar practitioners, the adoption of manual inspection is selected from most of the participants of this group for all AITD issues, especially for *Pipeline Jungle* and *Scattered Use of ML Libraries* that are selected from 11 participants. *AITD unfamiliar* group instead is mainly divided into participants who identify manual inspection as the main technique and participants who do not identify any identification technique. Particularly, *Scattered Use of ML Libraries* is identified through manual inspection by 16 *AITD unfamiliar practitioners*, while *Deep God File* from 12 *AITD unfamiliar practitioners*. The engagement of professional teams to identify AITD issues is almost reported by *AITD unfamiliar practitioners*, especially for *Undeclared Consumers*, where seven *AITD unfamiliar practitioners* and two *AITD familiar practitioners* report it. Additionally, the adoption of automated inspection is distributed similarly in the two groups. Finally, considering each issue, only a small number of the participants are supported using automatic tools to identify AITD. Participants use static analysis tools to identify *Pipeline Jungle*, *Scattered Use of ML Libraries*, and *Jumbled Model Architecture* (e. g., SonarQube). Concerning

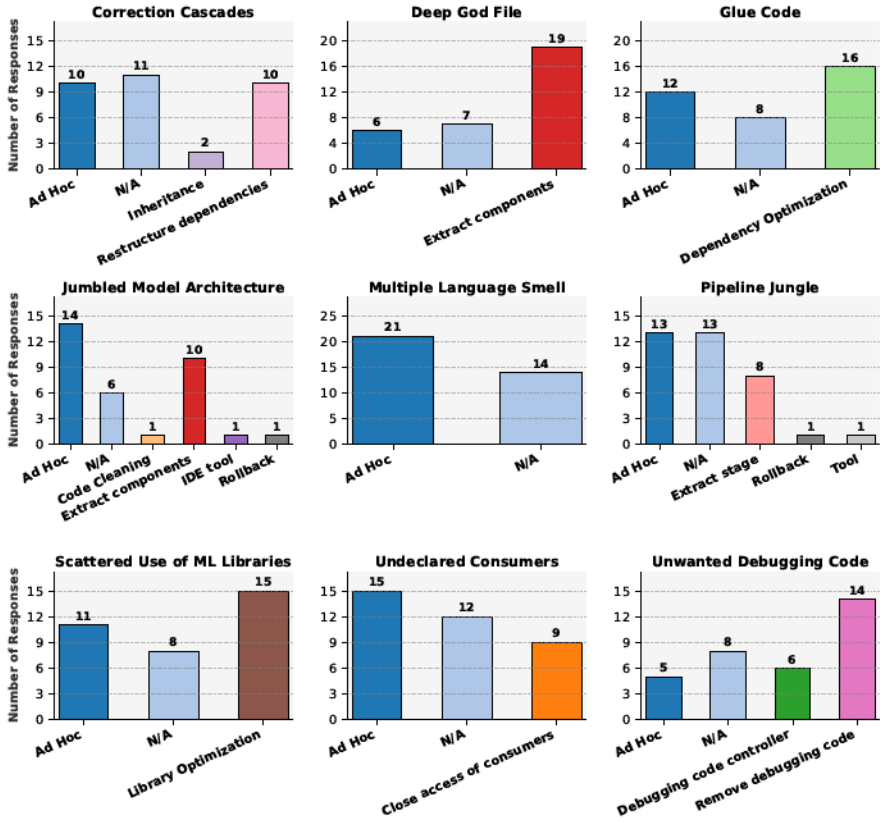


Figure 3.12: Frequencies of practitioners' answers of refactoring techniques useful for facing each AITD issue

Pipeline Jungle, two participants (P39, P40) identify and monitor the continuous growth of the pipeline through the orchestration of MLOps tools (e. g., Kale). In conclusion, while AITD is considered a possible threat to AI-enabled systems' quality, manually checking and reviewing the artifacts is the most commonly used technique.

The content analysis for the extraction of refactoring techniques was conducted over three iterations, with the third iteration indicating that a saturation point had been reached. First, 150 labels were identified to represent various refactoring techniques during the labeling process. Subsequently, the first categorization phase resulted in the identification of 47

distinct categories. Finally, the second categorization phase led to the identification of 15 strategies that practitioners employ to address AITD issues through refactoring. In the first iteration of this process, we extracted the main keywords that highlighted potential solutions to address the problem. For instance, a participant (P37) proposed a solution for *Undeclared Consumers*, stating: "Well, this is quite difficult because when there are unexpected users, it is very difficult to cut their access, and you need to talk with different departments. If it was OK, you would need a strong access system to avoid having this situation in the future, and only users with appropriate access can consume the info." From the participant's comment, we extracted the solution and labeled it as "Build an Access Control System." In the first categorization step, we simplified labels and grouped those denoting the same solution. For example, labels like "Build an Access Control system", "Restrict access of consumers", and "Close Connection to consumers" were grouped under the label "Close access of consumers". Finally, the second categorization iteration allowed the clustering of all the similar refactoring techniques. For example, techniques related to the splitting of a big component into smaller ones (e. g., "Extract stage", "Extract class", "Extract method") were grouped into a single category named "Extract components". In summary, the three iterations of the content analysis allowed us to systematically extract and categorize refactoring strategies practitioners use to address AITD issues, resulting in a structured set of insights.

Figure 3.12 collects all the labels used to represent a specific refactoring technique for each AITD issue. Each participant reported the technique used to face the AITD encountered, classified as *Ad Hoc Refactoring*, if the refactoring techniques involve manual code analysis to find an ad hoc solution to solve the issue. Otherwise, if the participant reported not using any technique to refactor the specified issue, the answer is classified as *Not Refactored*. For almost all AITD issues, most answers report the uses of *Ad Hoc Refactoring* (107 labels) and *Not Refactored* (87 labels). These results highlight the practitioners' criticality when facing these kinds of issues and the need to find an automatic technique to support developers during

the refactoring activities. Additionally, a good part of the participants highlighted the use of *Extract Components* to face three AITD issues (*Deep God File*, *Jumbled Model Architecture*, and *Pipeline Jungle*). Therefore, extracting a component could help define the modules used to build an AI model and increase the understandability of the whole pipeline. In this context, a participant (P34) highlights the need to isolate the components of each stage of the AI, specifying that: *"[...] Ensuring the five pipeline stages are written in adequately segregated code for troubleshooting purposes."* Regarding *Glue Code* and *Scattered Use of ML Libraries* issues, most of the participants denoted to perform an optimization of dependencies and libraries. In detail, a participant (P6) claimed: *"I would try to eliminate as many libraries as I could and gain a deeper understanding of the crucial ones that had to stay."*. Moreover, the set of answers for the refactoring techniques of *Undeclared Consumers* includes using a component that allows closing the access of consumers, reported by nine participants. To improve the reliability of the consumers that access the system, participants proposed implementing an access system that tracks the information about the application that accesses the model and restricts access only to registered users. Finally, the most reported technique for *Unwanted Debugging Code* is *Remove Debugging Code*. The ease of applying this refactoring technique is an additive hint that highlights the low severity of this AITD issue. Going deeper, analyzing the familiarity level of the participants that reported the stated refactoring techniques, almost all the two groups reported the same techniques. The most reported specific techniques for addressing AITD issues, such as *Dependency Optimization* for *Glue Code* and *Extract Components* for *Deep God File* are reported by participants of both groups similarly. A key difference between the groups is related to *Correction Cascades* issue. While most of the participants of AITD familiar practitioners proposed to perform ad hoc refactoring to address the issue or apply the use of the inheritance, the technique related to restructuring the dependencies is almost reported by AITD unfamiliar practitioners. Finally, automatic tools for addressing *Jumbled Model Architecture* and *Pipeline Jungle* are exclusively denoted by AITD familiar practitioners.

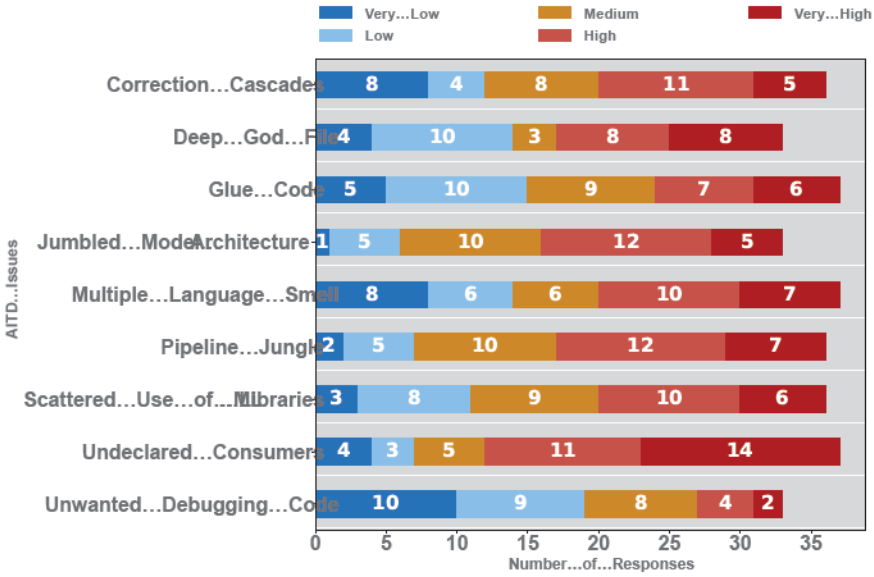


Figure 3.13: Effort depicted by the participant to identify AITD issues.

Figures 3.13 and 3.14 show the effort participants consider required to identify and refactor AITDs. In general, the participants' assessments align with the reported severity, with the majority of AITDs demanding substantial effort for both identification and refactoring, excluding *Unwanted Debugging Code*, *Glue Code*, and *Correction Cascades*.

Taking a closer look, the identification of *Undeclared Consumers*, which involves interaction with other applications, proved to be exceptionally challenging (14 participants classified the effort as "very high" and 11 as "high"). Additionally, AITDs closely tied to the architecture and AI pipeline, such as *Jumbled Model Architecture* and *Pipeline Jungle*, exhibited higher levels of effort required for their identification. This finding is especially confirmed by *AITD familiar practitioners*, in which all participants of this group confirmed at least a medium identification effort for *Jumbled Model Architecture* and *Undeclared Consumers* and only two participants from the group of 20 *AITD familiar practitioners* confirmed a low effort for identifying *Pipeline Jungle*.

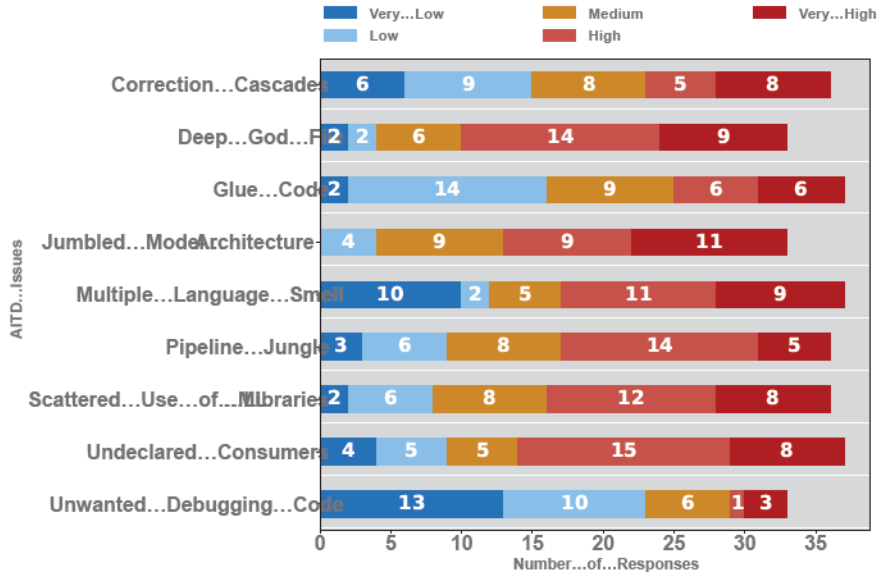


Figure 3.14: Effort depicted by the participant to refactor AITD issues.

Regarding the refactoring effort, nearly all AITDs were considered to demand a high level of effort by the majority of participants, particularly the *Deep God File* (with nine participants denoting "very high" effort and 14 participants denoting "high" effort). From the two familiarity level groups, this AITD issue is reported to require high effort, particularly by *AITD familiar practitioners*, where 10 participants denoted at least high effort, and only one denoted "low effort" to refactor it. The *Jumbled Model Architecture* requires high effort to be refactored, as stated by participants (with 11 participants characterizing "very high" effort and nine participants denoting "high" effort). In this case, the refactoring of this AITD issue is reported to require high effort by participants in both familiarity-based groups. Lastly, while the distribution of answers concerning the identification effort generally aligns with the refactoring one, for *Deep God File*, many participants deemed the refactoring process to require more effort than its identification.

3.3.6 *Causes of Technical Debt from the Practitioners' Perception*

In the final stage of our study, we asked participants for any additional feedback or concerns they had regarding potential issues that were not explored (as stated in S4D1 of Table 3.3). We received answers from 12 participants who provided valuable insights into common challenges that AI-enabled systems face, including technical debt resulting from failing to adhere to best practices and maintaining tools/libraries (P4, P32), as well as traditional software systems issues like a lack of standardization, privacy concerns, and time constraints (P5, P26, P41, P43). Interestingly, participants also noted that socio-technical factors can contribute to AITD. Specifically, communication issues or conflicts within teams (P18, P33, P34) were identified as potential causes of suboptimal solutions introduced by practitioners. In addition, incorporating new team members into developing AI-powered systems presents a distinct challenge. It requires specialized skills, which may not be easily accessible among less familiar team members (P47). This skill gap can lead to increased maintenance effort (P52) or the creation of convoluted and unorganized code, commonly called "spaghetti code." This complexity of code can obscure accountability for suboptimal solutions (P53).

3.4 THREATS TO VALIDITY

Several design decisions might have threatened the validity of our work. This section discusses our choices and how we attempted to mitigate the corresponding threats to validity.

EXTERNAL VALIDITY One of the primary issues that could potentially threaten external validity is selection bias in the recruitment process. Recruiting participants through social media or specific recruitment platforms may inadvertently result in gathering data from individuals who lack expertise in the domain of interest. To address this concern, we implemented a prescreening process to evaluate the eligibility of each partici-

pant before conducting the survey. This prescreening involved applying filters based on the practitioners' experience to ensure that only relevant and knowledgeable participants were included.

However, this design choice introduced another potential threat to the external validity of the study: the small sample size of 53 participants. This limited number of participants might not provide a comprehensive representation of the population of interest, impacting the generalizability of the findings. Nevertheless, opting for a smaller sample size was driven by the inherent difficulty in finding AI practitioners with expertise in the specific domain. Despite starting with a pool of 500 potential participants, eligibility criteria were carefully set to ensure that only individuals with relevant experience in software engineering, artificial intelligence, model development, and model deployment were included. By choosing a smaller sample size, the focus was placed on obtaining higher-quality answers that are more applicable and valuable within the specific context of the survey. Despite the potential limitations in generalizability, this approach aimed to prioritize the quality and relevance of the collected data. It was essential to balance the sample size and the depth of knowledge possessed by participants in the domain of interest.

In conclusion, while the representation of the population might not be fully reached to generalize the findings based on the practitioners' perspective, these findings serve as valuable hints that suggest the need for further investigation into these quality issues. The study outcomes provide insights that hold significance within the specific context of the survey and lay the foundation for future research in this domain.

INTERNAL VALIDITY Another significant concern is the participants' familiarity with AITD issues. Despite deliberately selecting participants with backgrounds in both Software Engineering and Artificial Intelligence, there remains the possibility that they may need more explicit knowledge of AITD issue definitions. To address this potential limitation, we employed the vignette experiment design method, which simplifies the presentation of AITD issues within scenarios commonly encountered during the design

and development of AI pipelines. Participants were provided with contextualized situations that helped them recognize and evaluate AITD problems based on their experiential knowledge, even without detailed familiarity with AITD terms. This approach enabled participants to effectively identify and assess AITD issues, drawing upon their experience. Therefore, participants can provide answers based on a common understanding of the issues presented through the vignette.

As for AITD issues, another significant threat could be related to the definition of AI that participants assumed. Recognizing that participants could have a different definition of AI based on their experience, the survey could involve different points of view, including participants unrelated to AI. To mitigate this potential concern, we conducted a prescreening phase to ensure all participants had a common baseline understanding of AI, comprehensive general skills, and AI-specific skills. Filtering the participants based on their answers, combined with a competence and attention check question, ensured that all the participants had a sufficient understanding of AI and were eligible for the survey. While a baseline is ensured among all the participants, it is important to acknowledge that the perceptions collected by the participants can vary depending on their experience in AI and their definitions.

Another potential threat to the internal validity of this study relates to the strategy used for assigning AITD scenarios to each participant. Traditional strategies, such as within-subject or between-subject approaches, can introduce certain limitations and threats. On the one hand, the within-subject approach can significantly increase the time required to complete the survey, leading to participant fatigue and impacting the quality of their answers. On the other hand, the between-subject approach may result in a limited number of questions answered by each participant, potentially reducing the overall depth and amount of the data collected. To address these concerns, we opted for a mixed-design approach, which allows for a balanced trade-off between the survey execution time and the number of questions each participant answers. By adopting the mixed-subject design, we aimed to ensure that participants had a manageable

survey completion time while still obtaining sufficient data from each participant. This solution enables us to capture broader insights while mitigating potential threats related to survey duration and participant burden.

In utilizing Prolific as a participant recruitment platform, an internal validity concern is the potential for incentive bias. Incentive bias may arise when participants on Prolific are motivated to complete surveys solely to receive compensation rather than genuinely engaging with the study. The financial incentive offered by Prolific could lead some individuals to rush through the survey or provide answers without thoughtful consideration, jeopardizing the quality and reliability of the data collected. To mitigate incentive bias, we introduced attention check questions during the prescreening phase, limiting the number of careless answers.

Another significant concern often arising when conducting questionnaire surveys is the potential use of external sources, which becomes more critical with the widespread availability of AI language models like ChatGPT[3]. Participants may collect and replicate the suggestions provided by ChatGPT when responding to our survey questions. Despite having conducted participant recruitment and data collection before the availability of ChatGPT, we must acknowledge the possibility that participants could have used other similar models that were accessible at the time of the study or turned to internet sources for information. We have taken several measures to address this threat of external source influence. We provided clear instructions to participants, explicitly stating that using external sources, including AI language models and internet browsing, is prohibited. We aimed to encourage genuine, independent answers that accurately reflect participants' knowledge and opinions. Furthermore, we have implemented validation and attention checks within the survey to assess the authenticity of participants' answers. These checks allow us to identify any potential AI-generated or internet-sourced answers that might affect the integrity of the data.

[3]ChatGPT: https://openai.com/chatgpt

CONSTRUCT VALIDITY A possible threat to the construct validity is related to the experience bias. This bias arises due to differences in experience and knowledge levels between researchers conducting the pilot test and industry practitioners who will participate in the main survey. The researchers' expertise and familiarity with AI systems may inadvertently influence the framing of survey questions, leading to potential mismatches with the real-world experiences of industry practitioners. To address this threat, a software engineer with experience in AI system development participated in the pilot test, providing valuable feedback from an industry perspective. This feedback informed refinements to the survey instrument, ensuring its relevance and clarity for industry practitioners.

Another potential threat we encountered was related to hypothesis guessing of research questions due to the comprehensive definition of AITD issues. Participants might have assumed that every issue presented should be regarded as problematic and directly responded in the affirmative. To mitigate this threat, we took proactive steps by broadening the definition of AITD issues to be more encompassing, representing them as vignettes and referring to them as *"situations"*. Additionally, we incorporated open-ended questions, enabling participants to provide detailed answers that allowed us to filter answers based on thoughtful consideration and genuine concerns. This approach ensured that we obtained more nuanced and reliable data, minimizing the impact of hypothesis guessing on the study's internal validity. By adopting these strategies, we aimed to foster a deeper understanding among participants while preserving the integrity of the research findings.

CONCLUSION VALIDITY A potential threat to the conclusion validity is related to the reliability of measures used to analyze participants' answers to address our research questions. We have implemented a strategy to enhance data collection to address this threat. In addition to the multiple-choice questions, we have incorporated open-ended questions that complement and support the answers provided in the multiple-choice format. By including open-ended questions, we encouraged participants

to provide more in-depth and nuanced answers, allowing us to capture additional insights and perspectives that may not be fully represented in the closed-ended options. This approach helped ensure a richer and more comprehensive data set, contributing to a more reliable analysis of participants' attitudes, opinions, and experiences regarding AITD issues.

3.5 CONCLUSION

In this chapter, we conducted a survey study with 53 practitioners to investigate the prevalence, severity, impact, and mitigation strategies applied to deal with nine distinct AITD issues. The study revealed multiple insights into how practitioners cope with the emergence of AI technical debt. Practitioners perceive these issues as highly severe and impactful, even though they are unaware of how prevalent these debt issues are in practice. Consequently, the automated support available to mitigate them is limited, and they often rely on manual inspections and ad-hoc refactoring approaches. The results of our work allow us to draw several considerations and implications for future research on AI technical debt, in addition to presenting a call for novel, automated methods that may support practitioners in identifying and refactoring AI technical debt.

The output of this work serves as the input for our future research agenda, which will focus on addressing the practitioners' needs that emerged from our analysis. On the one hand, we aim to design further empirical investigations into how AI technical debt is perceived and managed in practice. On the other hand, we strive to develop novel instruments to automatically detect and possibly refactor the technical debt affecting the source code and architectures. Additionally, this study initiated an exploration into a preliminary set of AITD issues. Our findings, while focused, open avenues for broader investigation into diverse types of AITD issues, particularly those related to socio-technical factors affecting practitioners' familiarity and communication issues.

4

UNMASKING DATA SECRETS: AN EMPIRICAL INVESTIGATION INTO DATA SMELLS AND THEIR IMPACT ON DATA QUALITY

4.1 INTRODUCTION

Artificial Intelligence (AI) is more and more diffused nowadays, being used by individuals and companies to make informed decisions [208] and automate tasks that would typically done by humans [140]. Indeed, AI-intensive systems, i.e., systems that embed artificial intelligence models and algorithms, have been recently deployed in multiple domains, with some recent applications showing highly efficient and accurate performance [105, 118].

However, the development of artificial intelligence-enabled systems differs from other types of software because the program and its effectiveness in solving a specific task heavily rely on the data and observations used to train models [19].

More specifically, AI-enabled systems are defined as *a system consisting of various software components, out of which at least one is an AI-specific component* [99]. In this context, data represents the primary source of producing business-oriented AI-enabled systems. Performing data analysis and validation is a crucial initial step for designers of machine learning components. Failure to properly analyze the training data may lead to model degradation [95]. Therefore, it is crucial to prioritize data quality to build a reliable and effective AI-enabled system. Data quality issues can arise for various reasons, e. g., data entry errors, inadequate data cleaning, or bias in the data. Addressing these issues requires a well-defined data quality management strategy that includes profiling, cleansing, and data enrichment [109]. While different tools and practices are available

to support feature engineering and data transformation for managing AI pipelines [148], the need to improve the practices related to quality assurance is continuously increasing [40].

Data quality degradation could also lead to technical debt for the whole system [49]. Data debt, primarily when introduced by data quality issues or data anomalies, can strongly impact AI-enabled systems, degrading model performance and causing problems to all the subsequent phases involved in the pipeline [18].

In analyzing technical debt specific to data, *data smells* are represented using the analogy of code smells. As code smells are defined as *symptoms of poor design and implementation choices* [52], data smells are *data value-based indications of latent data quality issues caused by poor practices that may lead to problems in the future* [50]. While other types of data quality issues are investigated in research to be addressed [63, 67, 97, 190], the knowledge about the effect and the presence of data smells is still limited to the catalog provided by Foidl *et al.* [50].

This limitation restricts the possibility of allocating research efforts to enhance the data quality management process of the system. Since the actual state of the definition of data smells is preliminary, it has been difficult to conduct studies to find management strategies for this type of data quality issue. By expanding the knowledge base of data smells, this study aims to provide researchers with a deeper understanding of how data smells can be identified and managed. Therefore, this study first explores the novel literature to update and extend the catalog of data smells. Subsequently, through empirical methods, we analyzed the prevalence and impact of data smells on data quality aspects in 19 real-world datasets. In particular, we updated the literature review and introduced three new types of data smells (i. e., *Redundant Value Smells*, *Distribution Smells*, and *Miscellaneous Smells*) and 12 new data smells to the existing catalog. We identify data smells in the most frequently used dataset and examine their relationship with data quality.

To sum up, our work provides the following main contributions:

- An updated catalog of data smells, which advances the state of the art by providing the definition of novel data smells, along with the information on how to detect them;
- Results of an empirical investigation into the prevalence and impact of data smells on data quality metrics, which can be used by researchers and practitioners to understand how data smells may impact the training of AI solutions;
- A publicly available replication package [149], which includes the whole set of data and scripts used in the study and that can be used by researchers to build on top of our findings.

Structure of the chapter. The chapter continues with the following sections: Section 4.2 explains the main goal and the overall method to address our research questions. Section 4.3 details the technique and the results obtained to produce the new catalog of data smells. Section 4.4 details the method and the results of the empirical analysis for the correlation of data smells and data quality. Section 4.5 discusses the threats faced and mitigated in this study. Section 4.6 concludes the article with the key findings.

4.2 RESEARCH QUESTIONS AND METHODS

The *goal* of the study is to address the existing gaps and limitations in understanding data smells and their impact on data quality. Specifically, it aims to contribute to the field by undertaking two primary objectives. On the one hand, it seeks to create an updated and unified catalog of data smells, building upon the previous literature review conducted by Foidl *et al.* [50]. On the other hand, the study intends to understand the relationship between data smells and data quality. To formalize and address the main goal of our study, we applied the Goal-Question-Metric approach proposed by Caldiera *et al.* [12]. In detail, we defined the goal of our study:

Our Goal.
Purpose: Understand
Issue: the characteristics of
Object: data smells, their prevalence, and their relationship with data quality aspects
Viewpoint: from the points of view of researchers and data engineers.

From the goal, we defined three main research questions. First, since there is a lack of a unified definition of the data smells defined in the literature, we wanted to elicit a complete and unified catalog of the main data smells defined in the literature with a name and a description of their features. In particular, we asked:

> **RQ1:** What specific data smells are documented in the existing literature and what are the tools to detect data smells?

Additionally, we needed a comprehensive analysis of data smell prevalence as a foundational step in developing effective data quality management strategies. Identifying common data smells and their frequency allows for prioritizing efforts in data cleaning, validation, and improvement. Therefore, we identified the following research question:

> **RQ2:** What is the prevalence of data smells in real-world datasets?

Finally, practitioners and researchers need to understand the relationship between data smells and data quality metrics to understand their severity and develop targeted strategies for data quality enhancement. By exploring significant relationships, we pinpointed the specific data quality dimensions most affected by certain data smells. With this aim, we formulated the following research question:

> **RQ3:** How do these data smells contribute to the degradation of data quality?

We leveraged two methods to address the main goal of the study and the related research questions. First, we followed the method by Wohlin [199] to update systematic literature reviews to explore data smells. This method provides a structured framework to collect, evaluate, and synthesize existing literature on data smells. Following the systematic literature review process, we aimed to identify and consolidate comprehensive information on documented data smells, their categorizations, and tools proposed for their detection, starting from the basis of the knowledge defined by Foidl *et al.* [50].

The tools delineated in the review process served as the foundation for the analytical phase to address the second and third research questions. Identifying data smells, we conducted a prevalence analysis within widely utilized datasets. In the final stage, we employed the metrics established by Elouataoui *et al.* [41] to undertake a correlation analysis to delve into the intricate relationships between the identified data smells and various data quality metrics.

4.3 ON THE EXPLORATION OF DATA SMELLS

To collect a comprehensive overview of the data smells defined by researchers so far, we extended the taxonomy provided by Foidl *et al.* [50]. In particular, we followed the guidelines by Kitchenham *et al.* [80] and Wohlin *et al.* [199], conducting four main steps, namely (i) database search, (ii) snowballing, (iii) application of exclusion/inclusion criteria, and (iv) quality assessment.

DATABASE SEARCH As a first step, we identified the key terms to use within the queries, extracting them from the research questions. Then, we found alternative spellings and synonyms for these terms and applied boolean operators as conjunctions, in particular, the *"OR"* operator for the union of alternative spellings and synonyms and the *"AND"* operator for the concatenation of the key terms.

These steps led to defining the following search query:

("data smell*" *OR* "data defect*" *OR* "data debt*") *AND* (("tool" *OR* "technique*" *OR* "strateg*" *OR* "identification" *OR* "refactoring") *OR* ("dataset*") *OR* ("definition*" *OR* "catalog*") *OR* (("machine learning" *OR* "artificial intelligence" *OR* "ai" *OR* "deep learning" *OR* "dl")))

The search query was executed against three databases, namely IEEE-EXPLORE, ACM DIGITAL LIBRARY, and SCOPUS.

INCLUSION & EXCLUSION CRITERIA The study selection criteria are intended to identify the primary studies that may address the research questions. Inclusion and exclusion criteria were based on the research questions and piloted to ensure they can be reliably interpreted to classify primary studies [80]. In our case, we filtered out resources based on the following exclusion criteria:

EC1. Articles not written in English.

EC2. Restricted license papers, so papers whose full-text read was not available for free.

EC3. Articles that were not peer-reviewed.

As for inclusion criteria, we defined the following:

IC1. Articles defining data smells.

IC2. Articles assessing tools about data quality and smells.

SNOWBALLING According to the guidelines formulated by Wohlin [198], we applied the exclusion and inclusion criteria defined before and after each iteration of backward and forward snowballing. The snowballing process has been iterated until a state of saturation (i. e., the snowballing process continued until the last iteration does not allow the inclusion of new articles).

QUALITY ASSESSMENT Before data extraction, we defined a list of questions to help assess the quality of the selected paper with the final goal of having high-quality resources and discarding the papers that did not provide enough details. Following, we report the set questions:

Q1. Does the paper define one or more data smells?

Q2. Does the paper provide instances of the impact of the data smell on a machine learning system?

Q3. Does the paper clearly define how the method is conducted to assess the presence of data smells?

Q4. Does the paper define one or more tools or techniques to identify or refactor the presence of data smells in a dataset?

Q5. Are the main aspects of the paper clearly defined?

The first four questions can be considered mutually exclusive. All the questions can be answered with "Yes" (Score = 1), "Partially" (Score = 0.5), or "No" (Score = 0); the final quality score for each paper was computed by summing up the score of the answers to the two questions, to be accepted the article should have at least a score of 1.5.

4.3.1 *Research Method Execution*

We began our exploration by comprehensively searching databases, which yielded an initial collection of 197 articles. We applied the inclusion and exclusion criteria to refine this pool and were left with a core set of just four papers. Following the methodology recommended by Wohlin et al. [199], we initiated a process of forward snowballing from the seminal work of Foidl *et al.* [50], as well as backward and forward snowballing for newly identified articles during the literature review. This method allowed for adding 14 other articles during the first iteration and eight more during the second iteration, bringing our total to 26 papers. The third iteration was significant in helping us determine the saturation state of our search. By

applying quality assessment criteria, we identified a final set of 12 papers that were eligible for inclusion. These selected contributions augment and refine the taxonomy, thus enhancing our understanding of the domain. This process allowed us to effectively narrow down a large set of articles to a smaller, more relevant set.

4.3.2 *Analysis of the Results*

4.3.2.1 *Data Smells Catalog*

Starting from the initial catalog defined by Foidl *et al.* [50], we collected data on smell definitions and types from 12 articles. To answer the first research question, we reported all the main data smells defined in the literature in a unified catalog divided by classes of smells containing a name and description.

In detail, starting from the original catalog defined, we found three new categories of data smell and 12 new data smells.

Figure 4.1 outlines various types of data quality issues known as "smells" within different categories such as *Believability*, *Understandability*, *Consistency*, *Redundancy*, *Distribution*, and *Miscellaneous*. These smells are early indicators of potential problems in datasets, which can affect data analysis and machine learning models. In the following, we describe each category in more detail:

Believability smells encompass issues related to the credibility and trustworthiness of data. New smells identified in this category are *Multiple Value Smell* [174] that describe multiple data in a unique value (e. g., the age and the gender of a specific person are represented as a unified value). In contrast, *Splitted Value Smell* represents a single data split in more columns [174].

Understandability Smells deals with issues affecting data comprehensibility, and it involves two subcategories. *Encoding Smells* highlight problems like representing integers as strings or vice versa, which can lead to confusion during data processing [50, 60]. *Syntactic Smells* cover issues such as special characters, spacing inconsistencies, and data values that

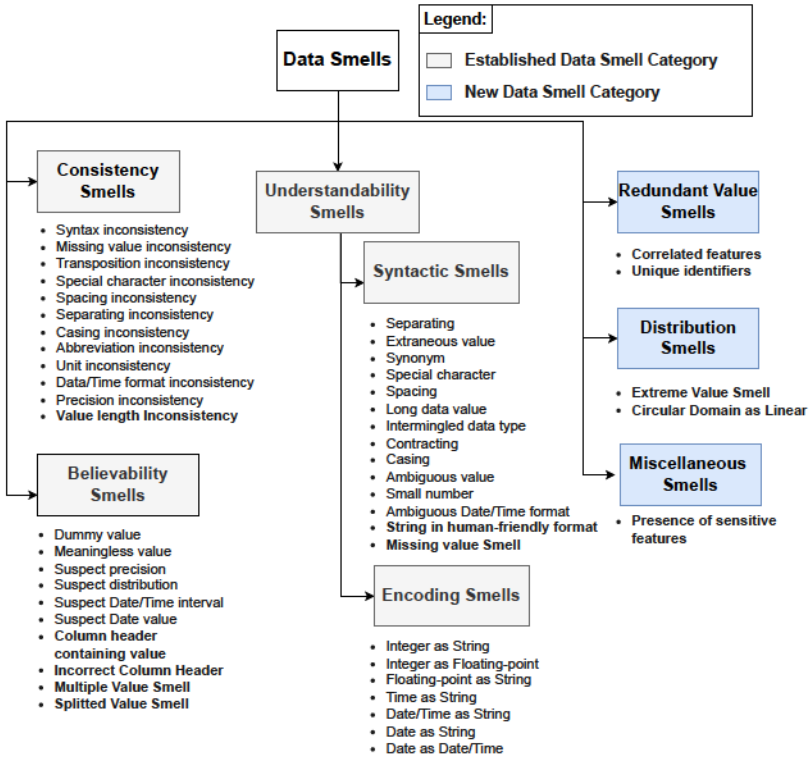


Figure 4.1: An extended version of the catalog of data smells defined by Foidl *et al.* [50]. The new data smells are highlighted in bold.

are too long to understand. New smells retrieved and grouped in this category are *String in human-friendly format* [165] and *Missing value Smell* [22, 39, 48, 65, 131, 159, 165, 174].

Consistency smells relates to the uniformity and consistency of data elements. Problems like *Syntax Inconsistency* and *Special Character Inconsistency* show discrepancies in how data values are used. Inconsistencies in spacing, casing, and unit measurements can also be problematic. Identifying and rectifying these inconsistencies is crucial for accurate data analysis and reporting. A new smell identified in this category is *Value Length inconsistency* [72], related to some of the values of a set that are represented

with a significant difference in terms of length. *Syntactic smells* represent the inappropriate use of values that increase the difficulty of interpreting and using the data [50]. A new smell identified under this category is *String in human-friendly format*, representing the use of human-friendly values. This increases the complexity of data analysis, as such values may pose challenges in automated processing. *Redundant value smells* point to data elements or features that provide little to no additional information for the training of AI models. *Correlated Features* [165] indicate redundant data when two features have a linear relationship, potentially introducing noise in models. *Unique Identifiers* [165] signifies redundant data that can be removed to improve data quality and model performance.

Distribution smells are related to the values in terms of the whole range of values represented. New smells under this category are *Extreme Value Smell* [22, 72], in which some of the values strongly differ from the distribution, and *Circular Domain as Linear* [72], in which a feature that has a range of values limited in a set of values (i. e., days of the week) are represented as linear. A possible solution to refactor this data smell is data binning, grouping values in intervals to select a single value representing the whole range [72].

Finally, *Miscellaneous Smells* covers various data quality issues that do not fit into the previous categories. The presence of *Sensitive Features* warns against including high-impact features that may introduce bias and unfairness in predictions [165].

Addressing these data smells is essential for ensuring the quality and reliability of data, which is fundamental for making informed decisions and building accurate machine learning models. Data cleaning and preprocessing techniques can help mitigate these issues, making the data more suitable for analysis and model training.

4.3.2.2 *Data Smells Identification Tools and Strategies*

The results highlight three main tools helpful in identifying data smells. To get further information about the characteristics of these tools, we

analyzed the documentation to understand their characteristics and which data smells can be detected using them.

The first tool is *Rule-Based Data Smell Detection* [50]. This tool uses an open-source data validation approach focusing on rule-based data smell detection. This tool is designed to identify smells such as Long Data Value, Casing, and several Encoding Smells. It includes a user-friendly graphical interface for uploading CSV files, enabling users to adjust the suspicion level based on predefined settings. Moreover, users can define parameters for each detection method individually. Subsequently, a machine learning-based version of data smell detection tools is implemented and proposed by Foidl *et al.* [50]. This version uses machine learning algorithms to detect data smells. In detail, different models are implemented and used to detect several *Inconsistency Smells* and *Encoding Smells*. Furthermore, to detect smells that rely on the semantics of the value, the tool uses an NLP approach (i. e., Word2Vec) to detect a *Believability Smell* (i. e., Synonyms). Finally, *Data Validator* is a component inside the framework TensorFlow Extended (TFX) that automatically collects information about the data schemas used for training machine learning models and reports quality issues to the user. From the set of quality issues able to define, Data Validator reports the presence of *Encoding Smells* (i. e., *Non-boolean value for boolean feature type* and *Extreme Value Smell*).

👉 **Answer to RQ$_1$.** Data smells can be classified into eight categories, namely, *Believability Smells*, *Encoding Smells*, *Syntactic Smells*, *Consistency Smells*, *Redundant Value Smells*, *Distribution Smells*, and *Miscellaneous Smells*, with a final catalog of 50 data smells. From the original study, 12 more data smells are added to the catalog, and three new data smell categories are defined. Finally, the results show three data smell detection tools to identify a significant part of the data smells.

4.4 ON THE PREVALENCE AND IMPACT OF DATA SMELLS ON DATA QUALITY

We conducted several steps to answer **RQ$_2$** and **RQ$_3$**. First, we collected datasets that allowed us to address our objectives. Subsequently, we selected one of the tools retrieved by the systematic literature review to identify data smells and assess their prevalence. Afterwards, we selected a set of data quality metrics that are measurable to understand the quality properties of the data and to perform a correlation analysis between the presence of data smells and the retrieved data quality metrics.

4.4.1 *Data Collection*

The first step of this phase was data collection, involving gathering datasets from the literature in tabular form to be analyzed by the detection tools. We based our research on the data studied by Le Quy *et al.* [85], which provides an overview of several real-world, tabular datasets used for fairness-aware machine learning and analyzes correlations between different attributes using a Bayesian network. We added the datasets reported by Hirzel *et al.* [70] to enhance this initial set. The datasets are divided by their application domain into different categories, namely: financial datasets, criminological datasets, healthcare and social datasets, educational datasets, and miscellaneous datasets; from the datasets reported in the papers, we extracted those related to classification tasks, resulting in a total of 19 datasets.

For each dataset, we provided a description and a set of metadata, including name, path, protected attribute, privileged classes, and favorable labels, to use in the data analysis phase.

4.4.2 *Data Smell and Data Quality Metrics Collection*

The second step of this phase consisted of analyzing the collected datasets. To carry out this phase, we built a tool wrapping the data quality tool DSD[1] and implemented a module to compute metrics about data quality.

This module implements the classes to compute the quality metrics of our datasets. We decided to rely on the metrics described by Elouataoui *et al.* [41]; since they introduced a set of metrics to assess the quality in the context of big data processes, we decided to consider only a subset of metrics, excluding all the time-related metrics and the process-related metrics. The final set includes the following metrics:

- **Completeness (Com)**: In big data environments, the collected raw data are usually incomplete and lack contextual information. Thus, data completeness is a crucial criterion when assessing data quality [41]. It can be defined as:

$$\frac{Number_of_non_empty_values}{Total_values} \times 100 \tag{4.1}$$

- **Uniqueness (Uni)**: Large-scale datasets are usually redundant since the data are gathered from multiple sources; therefore, the same information can be recorded more than once in a different format [41]. It can be defined as:

$$\frac{Number_of_unique_rows}{Total_rows} \times 100 \tag{4.2}$$

- **Consistency (Con)**: Consistent data should be defined as data presented in the same structure and types and coherent with data schemas and standards [41]. It can be defined as:

$$\frac{Number_of_values_with_consistent_types}{Total_values} \times 100 \tag{4.3}$$

[1] https://github.com/mkerschbaumer/rb-data-smell-detection

- **Readability (Read)**: Data validity is not limited to data format but also refers to data semantics. Indeed, raw data may contain misspelled words or even nonsense words, especially when the database is overwhelmed by human data entries [41]. It can be defined as:

$$\frac{Number_of_non_mispelled_values}{Total_values} \times 100 \tag{4.4}$$

4.4.3 *Data Analysis to Address* RQ_2

To address RQ_2, we run our wrapper of the DSD tool against the collected datasets. More specifically, the tool reports the presence of each of the considered data smells on each dataset, hence allowing us to assess the prevalence of data smells.

4.4.4 *Data Analysis to Address* RQ_3

To address RQ_3, we performed a statistical analysis based on hypothesis testing to test the effect of data smells on the data quality metrics. We defined two sets relative to the data smells analyzed (DS) and the data quality metrics retrieved (DQ).

$$DS = \{EV, MV, CA, SS, FS\} \tag{4.5}$$

$$DQ = \{Com, Uni, Con, Read\} \tag{4.6}$$

Then, we formulated the null hypotheses as follows, considering each combination of data smells and data quality metrics:

> $H_0(ds, dq)$: There is no statistically significant relationship between the data smell $ds \in DS$ and the data quality metric $dq \in DQ$.

When a null hypothesis can be rejected with high confidence, the test reports the possibility that there could be a negative effect that the data smells have on the data quality metrics:

Table 4.1: Distribution of Data Smells

Data Smell	**Total Count**
Extreme Value Smell (EV)	137
Missing Value Smell (MV)	110
Casing Smell (CA)	24
Suspect Sign Smell (SS)	20
Floating Point Number As String Smell (FS)	2

$H_a(ds, dq)$: There is a statistically significant relationship between the data smell $ds \in DS$ and the data quality metric $dq \in DQ$.

After selecting the tool to retrieve data smells and define the relative null hypotheses, we designed the statistical analysis to investigate the correlation between data smells and data quality. In detail, we defined x_{ds} as the *independent variable* representative of the number of occurrences of the data smell $ds \in DS$ and y_{dq} as the relative *dependent variable* representing the value of the data quality metric $dq \in DQ$. We analyzed the association between each independent variable and each single dependent variable to check for potential significant correlations. In particular, we applied Spearman's correlation rank coefficient [23]. The decision to use Spearman's correlation was based on our observation of the non-normal distribution of each data smell.

Guided by the observed correlations, we decided to deepen our understanding of the underlying relationships by employing a generalized linear model (GLM) [179]. Acknowledging that correlations do not imply causation, a GLM allows us to explore the dependencies between variables more nuanced, accommodating various distributional assumptions and potential nonlinearities. Before building the model, it is necessary to understand if multicollinearity could be present among all the variables [203]. In our study, we predefined a significance level (α) at 0.05 for hypothesis testing.

4.4.5 *Analysis of the Results*

PREVALENCE OF DATA SMELLS Table 4.1 summarizes the results we achieved. Considering the smells we analyzed, the most common is the *Extreme Value Smell* with a total count of 143 smelly instances, the second one is the *Missing Value Smell* with a total number of 119 instances, and the third is *Casing Smell* with 24 instances.

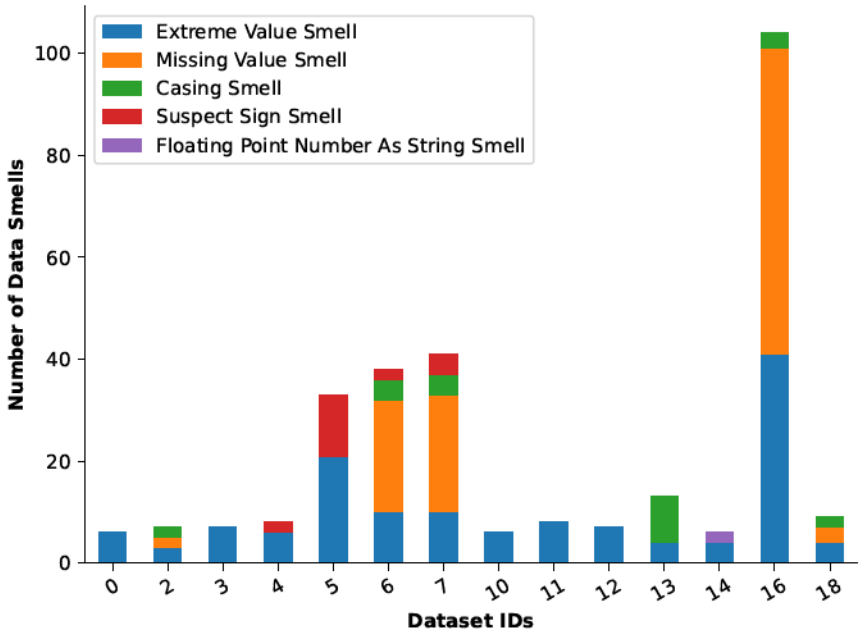


Figure 4.2: Prevalence of Data Smells in the analyzed datasets

Figure 4.2 illustrates the number of features affected by smells for each dataset. The datasets that do not have instances of data smell are not included in the plot to increase interpretability. Therefore, 14 out of 19 datasets present at least an attribute affected by data smells. The most diffused data smell in the several datasets is *Extreme Value Smell*, present in all the datasets affected by at least one data smell. While the prevalence of data smell is high, the distribution of the instances is not equally distributed across all the datasets. One of the datasets analyzed ("speed

dating", with Dataset ID *16*) presents 104 instances of data smells, composed of 60 instances of *Missing Values*, 41 instances of *Extreme Value Smell*, and three instances of *Casing Smells*. *Floating Point Number As String Smell* is present only in one of the datasets analyzed (*heart disease*, with DatasetID 14).

> 👉 **Answer to RQ$_2$.** To summarize the results, the most common data smell detected is the Extreme Value Smell with a total number of 137 smelly instances, and the second one is the Missing Value Smell with 110 instances. Most datasets have at least one attribute affected by data smells, resulting in 6 data smells found.

CORRELATION BETWEEN DATA SMELLS AND DATA QUALITY We first analyzed the correlation between each data smell and quality metric to answer the last research question.

Table 4.2 reports the results of the Spearman Correlation Rank test. Asterisks denote the significance levels, and no asterisks denote a p-value higher than 0.05, meaning low statistical significance.

The strength and significance of these correlations vary across measures. *Extreme Value Smells*, *Missing Value Smells*, *Casing Smells*, and *Suspect Sign Smells* exhibit statistically significant correlations with *Uniqueness*, suggesting a robust relationship. Regarding *Consistency*, *Extreme Value Smell*, and *Floating Point Number as String Smell* demonstrate a significant positive correlation, while *Missing Value Smell* displays a significant negative correlation. *Readability* shows strong associations with almost all the proposed smells, confirming a significant correlation with *Extreme Value Smells*. Finally, *Completeness* is highly correlated to the *Missing Value Smells*, highlighting a strong negative correlation, followed by a statistically significant but slight correlation with the *Extreme Value Smell*.

The observed variations in correlation strength and statistical significance emphasize the nuanced relationships between these smells and data quality aspects, contributing valuable insights to our understanding of the factors influencing text characteristics.

Table 4.2: Spearman Test Results

Variable	Independent Variable	Spearman Statistic	P-value
Uniqueness	EV	0.2038	***
	MV	0.2672	***
	CA	0.1667	***
	SS	0.1150	**
	FS	0.0578	
Consistency	EV	0.1533	***
	MV	-0.1833	***
	CA	0.0576	
	SS	0.0525	
	FS	0.2078	***
Readability	EV	0.3978	***
	MV	0.0213	
	CA	-0.2038	***
	SS	0.1376	**
	FS	0.1051	*
Completeness	EV	-0.1357	**
	MV	-0.9524	***
	CA	-0.0746	
	SS	-0.0002	
	FS	0.0311	

Significance levels: $p < 0.001$ (***),$p < 0.01$ (**),$p < 0.05$ (*)

From the outcome of the Spearman correlation ranks, we built the GLM model, selecting the independent variables that have a statistically significant association with the relative dependent variable. The results obtained from the applied Generalized Linear Model (GLM), as presented in Table 4.3, offer valuable insights into the significant relationships between data smells and data quality, revealing several correlations.

For each data smell, the reported relative coefficient represents the impact on the data quality metric, accompanied by the corresponding significance level. These correlation coefficients are calculated based on different proportions of the dataset, indicating how a unit increase in the specified data smell influences the associated quality attribute. It is impor-

Table 4.3: GLM Coefficient Statistics

Model	Variable	Coefficient (significance)
Uniqueness	EV	9.17×10^{-6}
	MV	2.65×10^{-7}
	SS	6.23×10^{-7}
	CA	8.67×10^{-5} (***)
Completeness	EV	4.03×10^{-6}
	MV	-0.0002 (***)
Consistency	EV	4.90×10^{-6}
	MV	-1.02×10^{-5} (***)
	FS	-6.29×10^{-6}
Readability	EV	0.0001 (**)
	SS	3.62×10^{-5}
	CA	-0.0001 (***)
	FS	-0.0025 (**)

Significance levels: $p < 0.001$ (***),$p < 0.01$ (**),$p < 0.05$ (*)

tant to note that even if the model indicates a low impact for the presence of a single instance of the data smell, the cumulative effect of multiple instances can be substantial. Thus, understanding the potential effects of data smells becomes crucial, especially in high quantities. The presence of *Missing Value Smells* exhibits a negative correlation, emphasizing its slight but statistically significant effect on *Completeness*.

The model related to the *Consistency* exhibits a highly significant negative correlation with *Missing Value Smells* but presents a very low coefficient, indicating that even if the variable could have a statistically significant effect, the effect of the data smells could be critical for this data quality aspect if the number of occurrences of *Missing Value Smells* is high. Differently, the effect of this data smell on *Completeness* is higher with a very low p-value, leading to a strong effect. Regarding *Uniqueness* of the

data, *Casing Smells* have a slight but statistically significant effect. More interestingly, the presence of *Extreme Value Smells* is positively correlated to *Readability*, leading to adding values that differ significantly from the distribution increases the amount of information the data gives. *Floating Point Number as String Smells* have a highly significant effect on *Readability* of the data. *Casing Smells* also slightly affects *Readability*, in which the relationship is extremely significant.

In summary, while the effect of a single instance of a data smell may appear negligible to data quality, the noteworthy significance levels associated with each correlation underscore that having data smells in high quantities relates to data quality. The cumulative impact of multiple instances becomes increasingly pronounced, emphasizing the potential detrimental effects on data quality.

> 👉 **Answer to RQ$_3$.** The analysis reveals nuanced relationships between data smells and data quality metrics. Data smells have a small but significant relation with data quality aspects. A single instance can slightly contribute to either decreasing or increasing data quality, as specified in the coefficients. However, when accumulating them, data quality is strongly affected.

4.5 THREATS TO VALIDITY

THREATS TO THE EXTERNAL VALIDITY This study's primary concern is the potential bias introduced by selecting database sources. To mitigate this threat, we adhered to the guided practices outlined by Wohlin *et al.* [198] by including main database sources. Further, we conducted two iterative rounds of snowballing to ensure a comprehensive exploration of articles addressing data quality issues.

While the study aimed to explore deeply data quality issues, specifically focusing on identifying data smells, there was a crucial decision to balance the scope. A more extensive exploration could have introduced various other types of data issues unrelated to the concept of smells. Although our

broad investigation led to the identification of new data smells and even novel categories of such smells, it is acknowledged that certain data quality issues associated with the concept of smells may remain undefined.

As for selecting the 19 datasets for studying the correlation between data smells and data quality aspects, careful consideration was given to address potential biases. The choice of these datasets was guided by the need to use representative datasets commonly used in AI applications and research. It is important to note that while the chosen datasets provide a robust basis for examining correlations, the variability among datasets may influence the generalizability of specific findings.

THREATS TO THE CONCLUSION VALIDITY The choice of instrumentation is critical to conclusion validity, particularly in detecting data smells and measuring data quality aspects. Firstly, utilizing the tool published by Foidl *et al.* [50] provides a foundation for identifying and assessing data smells. By leveraging an established tool, the study benefits from a standardized and structured approach to data smell detection. Moreover, using metrics defined and validated by Elouataoui *et al.* [41] adds another layer of rigor to the measurement process. From the set of metrics defined, we deliberately chose to focus exclusively on metrics tied to the structural elements of the data. With this decision, we intentionally omitted metrics associated with external factors, such as how the data are applied in the context of an application. The study gains access to well-established measurement tools by aligning with these metrics. While this study does not explore other aspects in the context of the data quality, it enhances the replicability of the research as a starting point.

THREATS TO THE INTERNAL VALIDITY While our study explores the correlation between data smells and data quality metrics, we recognize the threat related to internal validity—the potential influence of confounding variables. Although we observe correlations between data smells and metrics, it is essential to acknowledge the existence of other factors that may be tightly correlated and impact the metrics used in our study. These

confounding variables could introduce complexities, as correlations found may be influenced by factors related to the data-gathering process or the presence of different types of data quality issues beyond the scope of this investigation.

4.6 CONCLUSION

This study outlines the state of the definition of data smells in AI-enabled systems. First, we defined a new taxonomy with 12 new data smells and three new categories to extend the catalog of data smells. Then, we analyzed the presence of data smells to understand their relationship with data quality. The outcomes highlight the emerging severity of these types of data quality issues, that while the effect could be irrelevant in small amounts, the impact of such issues can be significant when introduced in high quantity. Therefore, this research enriches data smells and emphasizes their tangible implications on data quality. The high severity recognized serves as a call to increase the awareness of researchers and practitioners and leverage the need to institute robust strategies and best practices in data governance and quality assurance. As AI plays a pivotal role in various domains, understanding and addressing data smells becomes crucial for ensuring AI-enabled systems' reliability, trustworthiness, and effectiveness. As implications of these results, it is necessary to define new detection and refactoring strategies to support the practitioners in the data quality management process. On the one hand, such detection strategies could empower practitioners to identify and isolate specific instances of data smells, providing a more granular understanding of their presence and facilitating targeted interventions. On the other hand, defining new refactoring strategies to fix data smells effectively could support practitioners in guaranteeing high data quality and allow for investigating the effect on the overall AI-enabled systems.

DATA AVAILABILITY

The data collected as part of the systematic literature review, the scripts used to analyze and generate data, charts, and plots discussed when addressing our research goals, are publicly available at [149].

PART III

DETECTION OF AI-SPECIFIC TECHNICAL DEBT

5

LLM-BASED ARCHITECTURE TO CREATE GRAPH REPRESENTATIONS FROM ML PIPELINES

5.1 INTRODUCTION

Machine learning pipelines are a foundational abstraction for organizing the complex sequence of tasks required to develop, deploy, and maintain ML-enabled systems. The benefits of using machine learning pipelines are numerous, ranging from preventing bugs to change tracking and standardizing the ML lifecycle [68]. The complexity and multitude of activities involved in building AI-enabled systems require abstractions of the ML production process to prevent the continuous proliferation of activities that often lead to intricate workflows. Furthermore, the represented ML pipeline should span the entire lifecycle, covering data acquisition and experimentation, operational deployment, and serving. The model should also extend to retraining, maintenance, and subsequent incremental development. Building on the analogy of DevOps practices, research has focused on finding strategies to enhance the maintainability and control of ML pipelines, as well as defining MLOps strategies, guidelines, and tools [82, 148, 177]. Developers began designing pipelines, serving as standard references and often relying on tools such as Kubeflow Pipelines for orchestration, MLFlow for experiment tracking, KServe for serving, and Prometheus for monitoring and analytics.

While these tools support many MLOps activities, several challenges remain, including those related to the amount of information that can be extracted from ML-based systems [135] and their representation [7, 8, 17]. First, the heterogeneity of pipelines makes them hard to understand. Small pipelines often lack clear separation between the different stages, whereas larger pipelines can contain complex structures that conceal de-

tails [17]. Second, data scientists often create experimental workflows from notebooks. The transition to fully structured MLOps projects is an active challenge in research and practice, including the automated transformation of ML code in pipelines and having an overall overview of the ML development process [8]. Finally, in a fast-evolving environment of ML development, frequent changes occur with the emergence of new data, models, and preprocessing steps [7]. Given all the cases, there is the risk of falling into Pipeline Jungle [147, 161].

Pipeline Jungle observation requires the transformations of pipeline code into structured and informative workflows, as noted by Oakes et al. [116]. To achieve this aim, several syntactic approaches, such as MLInspect [66], have been proposed to extract the model representation of the workflow in ML Pipelines through API matching. However, the syntactic API-based approach suffers from a strong limitation: it lacks semantic understanding, creating representations that are not structured on a logical flow basis. Specifically, they do not express the actual complexity, based on the number of logical steps. This limitation has two critical effects. On the one hand, static representations create a high number of nodes when many APIs are included, even if the code is semantically simple. To illustrate this matter, let us consider the case of an ML pipeline related to the COMPAS Dataset.[1] The pipeline is composed of 54 Lines of Code and four logical steps: data loading, preprocessing, training of a logistic regression model, and evaluation on test data. An API-based inspection, performed with ML-Inspect, extracts a graph-based representation of 38 nodes. This example well illustrates how an API-based representation might be excessively fine-grained with respect to the actual complexity.

On the other hand, API-based approaches are tailored to the set of ML libraries used. Therefore, they are not applicable to a pipeline that contains novel libraries and custom code. This limitation is further exacerbated in a continuously evolving pipeline that rapidly adopts and replaces novel technologies. As a consequence, ML engineers are unable to have reliable

[1] Available here: https://github.com/DLR-SC/PipelinejungleEvaluation/blob/main/run_94264/compas.png

indicators on how complexity changes over time, missing the opportunity to prevent the emergence of Pipeline Jungles.

To overcome these challenges, we argue that a semantic segmentation of the pipeline is needed. To this end, LLMs may represent a viable instrument. Indeed, various studies have shown that LLMs can properly support code understanding tasks [110, 126, 153]. This chapter presents CLAIRE (**C**ollaborative **L**LMs **A**pproach for **I**ntelligent **R**epresentation **E**xtraction of ML Pipelines), which aims to enhance the understanding, maintainability, and scalability of complex ML workflows. The approach exploits LLM-multi agent systems, building upon recent studies that demonstrate promising capabilities of LLM-based multi-agent systems [29, 69, 93] in reducing hallucinations and unexpected faults.

CLAIRE generates a Directed Acyclic Graph (DAG) that captures the *functional and logical relationships* within the code, rather than focusing solely on syntactical aspects. This representation includes three types of edges: logical, execution order, and data flow. To enhance accuracy and reduce hallucinations, the tool uses a collaborative ensemble of small Large Language Models (LLMs), which work together to create more reliable pipeline representations. To the best of our knowledge, this architecture to support the evolutionary activities of ML-enabled systems represents a premiere of our work.

We evaluated CLAIRE in two steps. First, we collected 29 validated ML notebooks, split into ML steps, and evaluated its capabilties in performing semantic code segmentation. By comparing DAG representations with the original code, we investigate the fidelity of the DAG in reflecting pipeline logic and its effectiveness in highlighting critical stages of the ML process. Then, we conducted a qualitative analysis, collecting insights from 25 ML engineers to assess its accuracy in capturing ML pipeline structures and its utility in supporting pipeline comprehension. The goal of this step was to analyze its strengths and limitations in understanding, diagnosing, and refining ML pipelines, as well as identify areas for enhancement.

The results indicate that CLAIRE effectively captures the overall logical structure of ML pipelines, providing a clear visualization that significantly

aids in understanding the high-level workflow. LLMs demonstrated high performance in segmenting the code, yielding the best results when integrating MISTRAL-24B. The qualitative analysis revealed an increase in clarity level when adopting CLAIRE DAGs, particularly for preliminary inspections. CLAIRE can support evolutionary activities, such as modifying, extending, and optimizing ML pipelines. Participants argued certain detailed relationships, particularly in complex data flows, and elicited feedback for future improvements. To sum up, we provide the following contributions:

1. CLAIRE, an automated approach for generating DAG representations of ML pipelines, which focuses on capturing functional and logical relationships from code;
2. An exploration of the capabilities of collaborative LLMs to capture pipeline activities of ML-enabled systems;
3. A comprehensive evaluation of the approach, demonstrating its effectiveness in enhancing the understanding and diagnosis of ML pipelines;
4. An online, publicly available replication package [145] to promote transparency and facilitate replicability, reproducibility, and further research.

The remainder of the paper is organized as follows. Section 5.2 presents the CLAIRE approach, while Section 5.3 describes the evaluation method and reports the results. Section 5.4 discusses the limitations identified through the evaluation and outlines opportunities for future work. Section 5.5 addresses threats to validity, and Section 5.6 concludes the paper with final remarks.

5.2 CLAIRE: EXTRACTING LOGICAL REPRESENTATION OF ML PIPELINES

In this section, we introduce CLAIRE (**C**ollaborative **L**LMs **A**pproach for **I**ntelligent **R**epresentation **E**xtraction of ML Pipelines), a novel approach for automatically generating logical graph representations of ML pipelines. Figure 5.1 overviews the architecture of our approach.

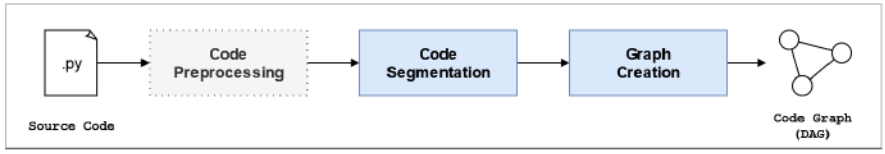


Figure 5.1: Architecture Overview: First, multiple LLMs extract logical segments from the source code. Then, a corresponding graph is created from those segments algorithmically.

The input is represented by the source code of the ML pipeline to analyze. This code undergoes a preprocessing step in which all empty lines and comments are removed, as these elements can cause misinterpretations by the LLMs during later processing stages. Afterwards, our approach proceeds with the two steps: 1. *'Code Segmentation'*, where multiple LLMs segment the code into logical segments;

2. *'Graph Creation'*, which generates a DAG to capture functional and logical relationships of the code. Finally, a deterministic algorithm extracts three types of edges among segments based on (1) logical dependency, (2) execution order, and (3) data flow. The final output is the labeled DAG representation of the input code, where each node corresponds to a logical segment of the code and each edge represents one of the three types of relationships. An example of such a DAG can be found in Figure 5.4.

The following subsections describe the workflow of CLAIRE in detail.

5.2.1 *Code Segmentation*

Once the code is preprocessed, it can be divided into logical sections. This step groups together code blocks that are semantically cohesive and correspond to stages of the ML pipeline. Given the code representing the ML pipeline, we aimed to extract the segments that compose it, a task referred to as *Code Segmentation* in the field of source code analysis [172]. As the task is designed to comprehend the meaning of program statements to define boundaries for each segment, it aligns with the capabilities of LLMs.

The *'Code Segmentation'* process applies multiple iterative stages, as illustrated in Figure 5.2. Specifically, the workflow works through the following steps:

1. *Chunking*: The next available chunk in the script, i.e., a small, manageable portion of the source code, is extracted from the source code. Details on how chunks are identified are provided later in this section.

2. *Stage 1 - Segmentation Proposals*: We prompt a set of n *voting agents* to each indicate where they determine the **first** segment should end, denoting the end line in the given chunk, starting from its beginning.

3. *Stage 2 - Best-Fit Selection*: All suggested segments from Stage 1 are integrated into a single prompt, from which a new *coordinator agent* chooses the most suitable solution after evaluating them one by one.

4. *Stage 3 - Segment Labeling*: Finally, we prompt the annotator agent to provide a title and a description for the extracted segment.

5. *Repeat*: Steps (1) to (4) are repeated until the end of the code is reached.

Once the end of the code is reached, the final result is a list of segments that originate from the original code.

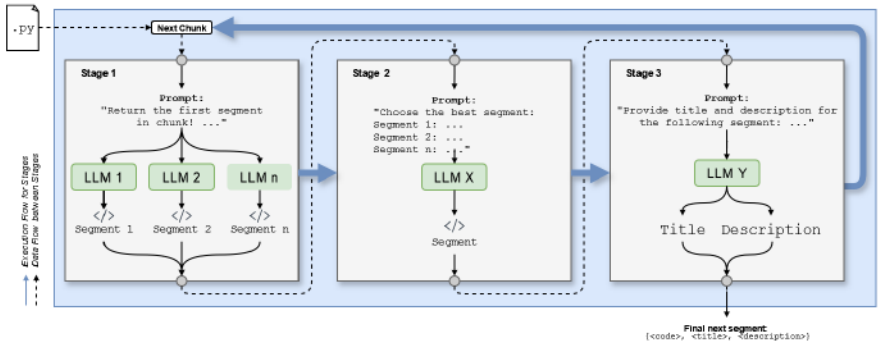


Figure 5.2: Code Segmentation Process: Multiple (LLMs) collaboratively determine logical code segments in a three-stage process. In Stage 1, several LLMs propose segmentation for a code chunk. Stage 2 utilizes a different LLM to select one of these proposals. In Stage 3, yet another LLM generates a description and title for the selected segment. This process repeats until the entire code is processed, resulting in a list of labeled segments.

Chunking. Language Models have a model-specific context size. It denotes the number of tokens that it can memorize at once. Even though LLMs commonly offer a context size of 8K tokens and far beyond[2] which could fit several hundred lines of code, in practice, it is not necessarily recommended to make use of the full size. In fact, research has shown that even for explicitly long-context models performance significantly degrades when valuable information is buried in long text [92]. Therefore, the longer and complex the content, the more difficult it is for LLMs to properly understand and memorize the content. We decided to provide the code to the LLMs in small chunks of 30 lines each. For every iteration of all three stages, the beginning of the next chunk is based on the ending position of the previous segment (see Figure 5.2). This implies that the *beginning of a chunk is also the starting point of the next segment.* We applied this solution to avoid overlapping or skipping statements analyzed by *voter-agents*.

[2]See for example https://medium.com/@contact_27876/comprehensive-comparison-of-large-language-models-llms-0da7e894e419

Stage 1 - Segmentation Proposals. As outlined before, the input to Stage 1 is the next code chunk, which also marks the beginning of the next segment. Code segmentation admits multiple viable solutions, with no single one being universally optimal. Therefore, the purpose of this stage is to identify different feasible solutions on where the next logical segment could end. We do this by individually prompting a set of `n` LLMs to each insert a marker into the code chunk at the location where they determine the current segment should end. Although the task could be accomplished with a single LLM only, we believe that utilizing multiple models provides the benefits of diversity and robustness. If one of the voter occasionally produced responses without respecting standard format, the other models act as a fallback, reducing hallucinations of individual models [37]. For the same reasons, multiple studies have presented work showing that a collaboration among multiple intelligent agents increases quality across a range of tasks [37, 94, 202].

The LLMs are restricted to placing their marker within chunk boundaries. For ease of analysis and later graph generation, we require each extracted segment to be syntactically valid in itself. We check whether markers are placed at locations that we consider invalid (e.g., within the arguments of a function call or other nested code), which can happen if, for example, the end of a chunk cuts a function call in the code prematurely. In those cases, we programmatically move the marker to the next possible valid location (e.g., the end of the function call), which can exceed the original length of the chunk. We illustrate such an example in Figure 5.3. Here, the LLM placed the marker of the end of the segment in line 30. However, as this line marks the middle of the object instantiation of `ColumnTransformer`, we consider this an invalid location. Hence, we programmatically determine the end and move the marker to the end of the instantiation (line 33). Reason for this choice is that splitting nested code will leave undesirable artifacts in the final DAG representation, for example a single function that got split into multiple nodes.

The reason to adopt a marker placement strategy with this heuristic refinement is to reduce reliance on the exact accuracy of the voter-agent



Figure 5.3: Segment Refinement: In some instances, LLMs may incorrectly place markers when an assignment is truncated. To correct this, we adjust the marker to the next valid position. For example, if the assignment for `nested_feature_transformation` continues to line 28 but the chunk ends at line 25, we manually relocate the marker to the end of the assignment to ensure accurate capture.

output and ensures that the extracted code aligns precisely with the intended input segment, limiting the task of the voter-agent to find the correct end of the segment over than replicating the code.

Stage 2 - Best-Fit Selection. Stage 2 aims to obtain a final choice on the `n` solutions that emerged from Stage 1. For this purpose, all the distinct suggestions are listed and combined into a single prompt. We also include the number of votes for each solution, as this shows the majority of the votes. Therefore, the coordinator-agent decides the most appropriate solution among the suggestions. It receives the identical segment definition that was used for the LLMs in Stage 1 and it is encouraged to evaluate each solution before making a final choice. Again, we require the LLM to output its response in the specific format `<Choice: [number of solution]>` so that we can parse the correct segment solution from it. At this point, the benefits of employing multiple LLMs over a single model become apparent, as unreasonable segments are more likely to be filtered out, resulting in reduced hallucinations in the final output.

Stage 3 - Segment Labeling. Once the final segment for the current chunk has been identified, we acquire a title and a description for it by prompting an *annotator agent*. Here, the LLM should strictly and briefly describe the code of the provided segment while avoiding to mention and assume any code outside the provided scope (i.e., outside the provided

segment). Finally, we asked to provide a short and descriptive title, limiting the length to 6 words. This format allows a good readability also to humans and brings the essence of the segment to the point. Examples of such titles format could be "`Configuring Model and Training Data for Fitting`" or "`Setting Up CIFAR-10 Model`".

Prompt Design. In all stages, LLMs are called with carefully designed prompts. LLMs tend to be sensitive to lexical changes in prompts, even when differences are barely noticeable to humans [205]. In fact, recent research illustrates that even single-character changes in prompts can dramatically impact LLM's responses [106]. During our research, this lead to a few prompting decisions that we found worked the best for the task of code segmentation.

First, we refer to a logical segment as a *"code group"*. This term appeared to align the LLMs understanding of a logical segment after preliminary analysis. Other terms, like *"code block"* or *"code step"*, seemed biased towards aspects focusing on the significance of the segment towards the complete Python code. Using these terms would more frequently lead to undesired responses such as *"[...] Therefore, I will not choose any of the candidates [...]"* (for Stage 2).

Second, we guide the LLMs towards first understanding the provided code chunk and then performing a brief reasoning before making a choice. For example, in Stage 2 we ask the LLM to describe each of the provided solutions and evaluate their suitability before providing a final decision. This kind of response will place the model's reasoning into it's immediate memory during generation, enabling the LLM to provide a more logical response that is based on actual reasoning [193].

Third, contrary to the idea of few-shot prompting, where prompts include one or more examples of the desired output, we observed increased code hallucination specifically from the provided examples when including samples in the prompt. This distorts the LLM's internal representation of the code chunk, resulting in decreased segmentation quality. In addition, it makes parsing of the response more difficult, as the returned code

contains unrelated, hallucinated code frequently. Therefore, we do not include any example of code or desired outputs in the prompts.

Lastly, as mentioned earlier, in Stage 1 we experiment with varying the system prompt among the `n` agents to encourage diverse segmentation solutions. We include one individual persona-description (inspired by [162]) per LLM, such as *"You prefer breaking code into very small, fine-grained segments"*. Apart from that, segmentation diversity largely originates from the use of different models as well as the nondeterministic nature of LLMs.

For the sake of readability, we do not include the designed prompts in this paper. However, the interested reader can find the prompts in our replication package [145].

5.2.2 *Graph Creation*

Our overarching goal is the creation of a **D**irected **A**cyclic **G**raph (DAG) that represents the logical flow of the corresponding code. To extract these edges, we leveraged traditional static analysis techniques. Our DAG consists of nodes and edges. A node represents a segment of the code and has three properties `title`, `description` and `code`. All three were identified during the preceding *'Code Segmentation'* (Section 5.2.1) task. An edge represents a correlation between two nodes. Edges are directed and we define and differentiate between three types of edges: *logical*, *execution order* and *data flow*, characterized as follows.

Logical Edge. A logical edge $e_l = (A, B)$ between two segments A and B exists if there exists a variable v assigned in A whose value is read in B. If v is only reassigned in B and it's value subsequently read in third segment C, then the edge does not extend from A to C; it only exists from B to C. In other words, two segments are connected by a logical edge if the segments use each others variables. Therefore, a single node can be the source and/or target node of multiple logical dependencies.

Execution Order Edge. As the name suggests, this type of edge represents the order of execution of segment nodes. It provides a definite execu-

tion order for each of the nodes, which, for single-threaded applications, depicts a single path through all nodes.

Dataflow Edge. Finally, data flow edges track the flow of data loaded from external sources (e.g. training data loaded from a .csv file). A data flow edge $e_d = (A, B)$ between two segments A and B exists if there exists a variable u assigned in A that loads external data, and u is then used in B. As per definition, e_d can only exist if there also exists a logical edge e_l from A to B. To recognize the presence of a data-loading function in the code, we manually define several dozen data loading functions from libraries commonly used in Machine Learning applications[3] for simplicity.

Figure 5.4 shows an example DAG generated from a public Machine Learning pipeline. The corresponding code aims to train and test a Machine Learning pipeline to predict recidivism (the tendency of a convicted criminal to reoffend). The graph shows the major logical steps of the pipeline as well as the flow of the training and test data.

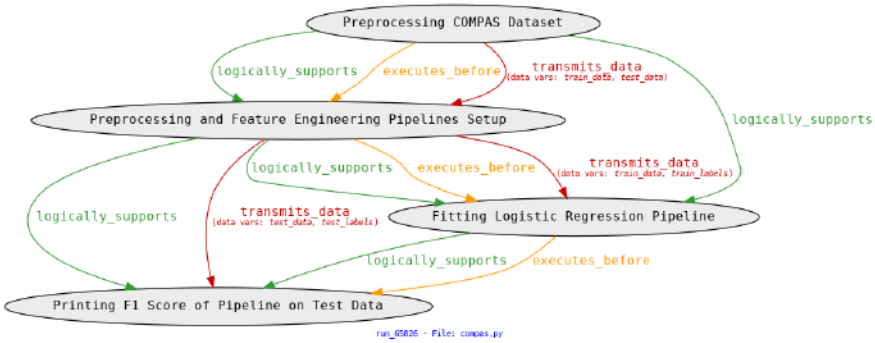


Figure 5.4: Example of CLAIRE-generated graph extracted from COMPAS Machine Learning pipeline showing "logical" (green), "execution order" (orange), and "data flow" (red) edges.. Red edges are labeled with the names of data-carrying variables.

[3] A few examples of data loading functions are `read_csv`, `loadtxt` and `datasets.load_iris` from the *pandas*, *numpy* and *sklearn* libraries respectively.

5.3 EVALUATION

To evaluate CLAIRE, we formulated the following overarching research question:

> **RQ1**
>
> *How effective is* CLAIRE *in generating accurate and clear ML pipeline representations from source code?*

Addressing this question requires combining complementary perspectives. A purely quantitative evaluation captures the performance of LLMs in segmenting code into pipeline steps by comparing the generated partitions with a predefined ground truth. While this enables us to measure accuracy and establish objective performance boundaries, it does not reflect how practitioners perceive the clarity or usefulness of the resulting representations. Conversely, a purely qualitative evaluation with practitioners reveals perceptions of clarity and adherence to the underlying pipeline but lacks objective evidence of accuracy.

Hence, we adopted a mixed-method empirical approach [173]. First, we conducted a quantitative evaluation to assess the effectiveness of collaborative multi-agent LLMs in code segmentation. Second, we conducted a qualitative study with ML engineers to analyze their perceptions of the clarity and fidelity of the generated DAG representations.

5.3.1 *Quantitative Evaluation*

Code segmentation is a non-trivial task by nature, as there can be multiple different yet valid segmentations of the same code. Since our methodology utilizes LLM ensembles to identify logical segments in the code, we are interested in how different ensemble setups perform.

5.3.1.1 *Context Selection*

To ensure that our evaluation reflects realistic ML pipelines, we require a set of annotated code samples representing ML Pipelines. However, to the best of our knowledge, there is no publicly available dataset of ML pipelines that is annotated at the level of individual pipeline steps. In line with research practices, we utilized 29 curated and real-world Python scripts from the dataset provided by Biswas et al. [17], who systematically analyzed representative ML Python pipelines to characterize their properties and their applications. Subsequently, we created a ground-truth set enabling us to evaluate the performance of our LLM-based approach. The first two authors independently performed a manual annotation of the 29 scripts, splitting each one into code segments representing the corresponding ML pipeline steps. We distribute this manually annotated dataset as part of our online replication package [145].

5.3.1.2 *Code Segmentation Evaluation*

To compare the ground truth and LLM-generated segmentations, we measured their difference using Mojo-Distance. Other similarity metrics, such as Jaccard similarity or cosine similarity, primarily focus on set overlap or vector space representations, which are not well-suited for capturing the structural boundaries within segments. In contrast, MoJo distance was explicitly designed to evaluate partitioning tasks to quantify the cost required to transform one segmentation into another. Initially proposed by Tzerpos and Holt [184], the Mojo distance has been helpful in recent studies [46] for code and software partitioning, as it effectively captures both over-segmentation and under-segmentation errors by penalizing mismatched segment boundaries. We specifically adopted the MoJoFM measurement [194], which is a widely used extension of the original MoJo distance. The standard MoJo metric assesses the dissimilarity between two partitions by calculating the minimum number of "Move" and "Join" operations needed to transform one clustering into another. However, the original formulation has two main limitations: (i) the algorithm may

yield suboptimal results, and (ii) the measure is asymmetric, which makes it sensitive to the comparison order. MoJoFM addresses these issues by normalizing the number of operations relative to the worst-case scenario, resulting in a similarity percentage scale value. A value of 100 indicates identical partitions, while a zero signifies complete dissimilarity. During the investigation we observed hallucinations introduced by the LLMs. When their output did not provide a complete partitioning of the system or contained format errors, the segmentation was considered a failure, yielding a dissimilarity score of zero when computing MojoFM. Therefore, we measured also the failure rate, defined as the ratio of failed executions over their total number.

5.3.1.3 *LLM Selection*

For each script, we repeated the LLM-based segmentation six times and averaged the results to obtain a more significant outcome. We evaluated a range of different Language Models for the task, exploring small- to mid-sized models between 3B and 24B parameters. This choice was driven by multiple considerations. First, organizational constraints limit the LLM selection pool to open-source Llama-based, Mistral, and Falcon LLMs. This constraint is driven by the compact size and efficiency of these models, which allowed us to balance performance with memory requirements; we chose these lightweight versions over models with a larger number of parameters (e.g., 70B+) to achieve effective segmentation within a resource-efficient setup. Second, through our experimentation, we found these models to provide sufficient capacity for accurate segmentation without context size overflow, making them well-suited for dynamic chunking and logical segmentation tasks. Third, as open-source models, they provided a cost-effective and accessible solution, which can be adopted in industrial settings where models may pose integration and scalability challenges. For our comparison, we selected the following models : `Falcon 3 (3B)`, `Llama 3 (3B)`, `Phi (3B)`, `Mistral (7B)`, `Falcon 3 (7B)`, `Llama 3 (8B)`, `Llama-Pro (8B)` and `Mistral (24B)`. Afterwards, we combined different models across various configurations to architecture CLAIRE:

- *Homogeneous ensembles*, i.e., a combination of three identical models in Stage 1 of our pipeline, to assess a baseline for each model type;
- *Heterogeneous ensembles*, i.e., a combination of three different models in Stage 1 of our pipeline, to assess the performance of mixed models;
- *Ensembles of one to four models* in Stage 1 of our pipeline, to assess the effect of ensemble sizes.

For a fair comparison between the different models, as recommended in the empirical guidelines for Software Engineering concerning LLMs [11], we used a fixed value of *0.1* to set the temperature of the LLMs. Due to high computational complexity, it was not possible to combine all model types in all possible combinations; therefore, we selected specific combinations based on the following scheme:

1. First, we obtain a baseline for the performance of homogeneous ensembles (Table 5.1 - Iteration 1);
2. Second, we select the best performing resulting models and combine them into an heterogeneous configurations of voter agents (Table 5.1 - Iteration 2);
3. Finally, we evaluate the performance of the best ensemble (hetero- or homogeneous) by varying the number of LLMs used in that ensemble (Table 5.1 - Iteration 3).

Apart from assessing the quality of segmentation boundaries, this setup allowed us to estimate the robustness of different model configurations based on their failure rate.

5.3.1.4 *Results*

Our empirical results are summarized in Table 5.1.

Table 5.1: Average performance for selected LLM configurations. Best results per iteration are marked in **bold**.

Iter.	Idx.	Ensemble	Avg. Mojo	Median Mojo	Min. Mojo	Max. Mojo	Std. Dev. Mojo	Failure Rate
1	A	3x Falcon (3B)	28	20	10	89	23.99	61.2 %
	B	3x Llama (3B)	46	49	0	86	22.80	39.1 %
	C	3x Phi (3B)	0	0	0	9	1.57	96.0 %
	D	3x Mistral (7B)	74	79	10	90	17.62	6.60 %
	E	3x Falcon (7B)	67	74	10	89	21.28	10.0 %
	F	3x Llama (8B)	58	63	12	92	20.70	25.0 %
	G	3x Llama-Pro(8B)	1	0	0	10	2.70	95.8 %
	H	3x Mistral (24B)	**78**	**80**	12	94	15.44	**4.60 %**
2	I	1x Mistral (24B) + 1x Mistral (24B) + 1x Mistral (7B)	**77**	**83**	0	92	17.06	**3.3 %**
	J	1x Mistral (24B) + 1x Mistral (7B) + 1x Falcon (7B)	69	77	0	92	17.06	11.6 %
	K	1x Falcon (7B) + 1x Mistral (7B) + 1x Llama (8B)	66	75	0	89	22.95	13.6 %
3	L	4x Mistral (24B)	**80**	**83**	0	95	17.33	**3.3 %**
	M	2x Mistral (24B)	79	**83**	0	95	17.43	**3.3 %**
	N	1x Mistral (24B)	79	82	0	95	17.33	**3.3 %**

Mojo-based Accuracy. Iteration 1 (A–H) evaluates homogeneous ensembles composed of three identical models. Performance varies widely: ensembles of `Phi 3B` (C) and `Llama-Pro 8B` (G) failed to produce meaningful results, whereas `Mistral 24B` (H, 78% average Mojo) and `Mistral 7B` (D, 74%) achieved the highest similarity.

Despite these large differences in performance, a clear trend is visible where model performance generally improves with the number of its parameters. For example, 3B models failed to score higher than 46% of similarity, while all larger models (with the exception of `Llama-Pro`) score 58% and beyond.

Iteration 2 (I–K) explores heterogeneous ensembles combining top performers from Iteration 1 (`Mistral 24B`, `Mistral 7B`, `Falcon 7B`, and `Llama 8B`). These ensembles reached 77% (I), 69% (J), and 66% (K) aver-

age Mojo. Although reasonably strong, none surpassed the best homogeneous configuration (H), indicating that mixing models does not necessarily enhance accuracy.

Iteration 3 (L–N) examines whether scaling the number of high-performing models further boosts performance. Using only `Mistral 24B`, ensembles with 1–4 members produced nearly identical results (79–80% average Mojo). Thus, increasing the number of identical high-performing models appears to provide no accuracy benefit beyond that of a single model.

Robustness. It is reasonable to assume that mixing diverse model types increases the system's robustness. In order to test this, we track the robustness of each ensemble by measuring their number of failed executions (failure rate). On one hand, we can observe that the failure rate varies greatly among homogeneous ensembles (Iteration 1), ranging from as little as 4.6% (`Mistral 24B`) to 9system'sxtttPhi 3B). On the other hand, heterogeneous ensembles (Iteration 2 + 3) appear to stabilize failure rates, yielding between 3.3% and 13.6%. This is in line with the observed accuracy measurements, as high failure rates negatively impact the accuracy score. In fact, the failure rate is likely a main driver for accuracy. Table 5.1 summarizes the standard deviations of ensemble accuracies. Across most ensembles—excluding C and G—the standard deviation hovers around 19. At first glance, C and G appear remarkably robust given their very low variance; however, this is only because their results are consistently poor, a form of 'robustness' that is of little practical value.

5.3.1.5 *Implications of the Results*

Accuracy. Among the numerous model ensembles, a large variability can be observed in both Mojo-based accuracy and failure rates. The best performance was achieved by Mistral models (Table 5.1 H and D), closely followed by `Falcon` (E). Specifically `Mistral 24B` delivers strong performance right out of the box, even when used as a single model (N).

> **Take Away Message 1.** Mistral-based models show the highest performance in code segmentation, with `Mistral-24B` reaching 78% of Mojo-Distance.

At first glance, the results indicate that heterogeneous ensembles (`I`–`K`) achieve higher accuracy compared to homogeneous ensembles made up of weaker models (`A`–`H`). Specifically, ensemble `I` (2× `Mistral 24B` + 1× `Mistral 7B`) differs from ensemble `H` (3× `Mistral 24B`) only by replacing one 24B model with a weaker 7B. Consequently, `I` performs better than the weakest homogeneous configuration (`3× Mistral 7B`) but worse than the strongest (`3× Mistral 24B`). In all instances, the heterogeneous configurations outperform the homogeneous ensembles that consist solely of the weaker models included. This suggests that combining different models can help reduce the negative impact of weaker members. However, it also means that the ensemble may not reach the peak performance of its strongest component. Overall, the performance of the ensemble tends to balance between the strengths and weaknesses of its models.

Definitely, results show that heterogeneous ensembles follow the same directions of similarity of homogeneous model. While a variety of models can be expected to enhance the creativity of the ensemble, we noticed that if the architecture contains best models the Mojo-Distance consistently increase. Therefore, results suggests that the selection of a proper LLM as an agent prevails over the heterogeneity.

> **Take Away Message 2.** Heterogeneity in LLM selection does not improve particularly the Mojo-Distance similarity when compared to simply using the strongest LLM.

In Iteration 3, the number of LLMs participating in the voting process is varied. While ensembles with four voters (`L`) show a slight improvement, the performance remains strong even with just a single one (`N`). Therefore, increasing the number of agents in the voting process does not provide significant benefits. Interestingly, adding more voters does not lead to a decrease in performance, suggesting that the ensemble successfully avoids

confusion from conflicting outputs. Overall, the improvements observed are negligible when compared to a single strong model.

> **Take Away Message 3.** Incrementing the number of LLMs in the voting process does not particularly increase the code segmentation performance.

Robustness. The effect of individual, high-performing models being a main driver for overall performance also applies to robustness. Table 5.1 shows that ensembles containing additional numbers of high-performing models (`Mistral 24B` and `Mistral 7B`) also report decreased failure rates. Equally, adding weaker models such as Falcon or Llama (`J` and `K`) increases failure rates to 11.6% and 13.6%, compared to only 3.3% for the Mistral-only ensemble `I`.

> **Take Away Message 4.** Adding LLMs that poorly perform in code segmentation also decrease the robustness of the answers.

While heterogeneous ensembles improve accuracy over those composed of only weaker models, they do not always improve robustness over homogeneous models. For example, ensemble `J` (`Mistral 24B` + `Mistral 7B` + `Falcon 7B`) has a failure rate of 11.6%, which is higher than each of its members alone (`Mistral 24B`: 4.6%, `Mistral 7B`: 6.6%, `Falcon 7B`: 10%). This is counterintuitive, as we would expect diverse ensembles to lower the failure rate, yet the opposite effect is observed in this case. Breaking this pattern, ensemble `I` (2× `Mistral 24B` + 1× `Mistral 7B`) achieves a failure rate of 3.3%, which is lower than the homogeneous ensembles of either `Mistral 24B` or `Mistral 7B` as a homogeneous mix (`H` and `D`). In general, these observations suggest that heterogeneous ensembles can sometimes lead to more robust behavior than their homogeneous members, depending on the combination of model adopted.

The patterns observed in homo- and heterogeneous ensembles are further confirmed when examining ensembles with different numbers of `Mistral 24B` models (Iteration 3). Specifically, we tested ensembles that contain 1, 2, 3, and 4 models (`N, M, H, L`) and found that they maintain

consistently high accuracy and low failure rates. This reinforces our earlier finding regarding accuracy: because the individual LLMs do not influence each other's decisions, an ensemble cannot exceed the performance of its strongest member. Therefore, combining multiple `Mistral 24B` models does not improve the peak accuracy beyond that of a single model.

> **Take Away Message 5.** Adopting heterogeneous models or varying the number of LLMs does not differ in terms of failure rate.

Finally, just as we could expect the failure rate to decrease for heterogeneous ensembles, we could expect their standard deviation to decrease, as more heterogeneous ensembles could tend towards producing more consistent results. However, looking at 5.1, this effect can not be verified with the current data. We determine only minor differences between the average standard deviation of homogeneous ensembles (which is 20.02 for `A-H`, but excluding `C` and `G` as outliers) and heterogeneous ensembles (which is 19.02 for `I-K`). This indicates that heterogeneous ensembles are not significantly more consistent in their performance than homogeneous ones.

To summarize, our study first suggests that the accuracy of an ensemble cannot exceed that of its strongest individual member. To achieve such an effect would require an adjusted architecture that allows models to impact and improve each other's decisions. However, mixing different models can help to balance high and low accuracy which could provide a safer choice when it is unclear which model performs best on a given task. Furthermore, our results suggests no particular improvement when adopting heterogeneous ensembles over homogeneous counterother'sFurther investigation on other types of LLMs are needed to confirm these findings. Finally, when a single high-performing model is available (e.g. `Mistral 24B`), using just that one may be sufficient, as adding more strong models yields little to no accuracy gain, and a single model already achieves near-optimal robustness.

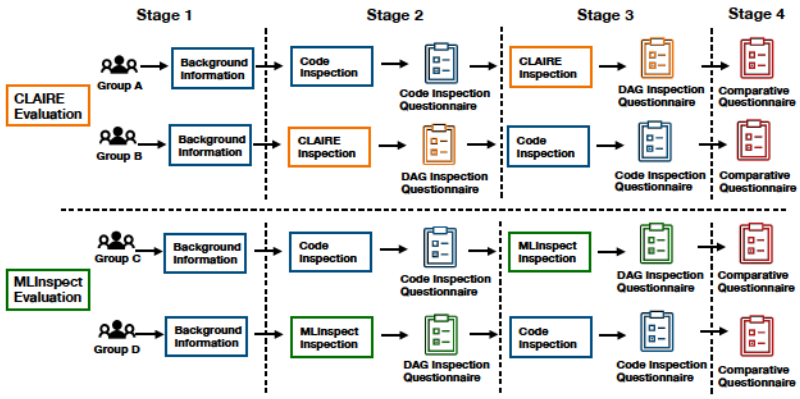


Figure 5.5: Process of the Sequential Comparative Design for the Qualitative Evaluation.

5.3.2 *Qualitative Evaluation*

After evaluating the segmentation accuracy of LLMs for ML pipeline representation, we conducted an additional study to assess the *usefulness* of the generated DAG representations in supporting pipeline comprehension.

As described earlier, our approach produces a DAG-based representation of ML code. This representation is intended to support program analysis tasks, but its value depends on whether it accurately reflects the logical structure and workflow of the underlying code. We adopt a *qualitative* survey study to capture participants' perceptions on the usefulness of the DAG's. Unlike quantitative metrics, qualitative feedback provides insights into how engineers interpret logical structures and pitfalls in the DAGs and understanding issues originating from inspecting ML Pipeline representations. This analysis complements the quantitative evaluation of segmentation accuracy, allowing us to examine not only whether the DAG is precise, but also whether it is practically meaningful for engineers. The qualitative evaluation investigates two aspects: (i) the adherence of the DAG to represent the source code and (ii) its clarity in aiding ML engineers' understanding of pipeline phases. To effectively capture participants' perceptions, we employed a *comparative evaluation* [200], contrasting the

two representations (DAG and code) to assess (i) the accuracy of the DAG, reflecting its reliability in correctly capturing pipeline information and (ii) the usefulness of the DAG in supporting ML pipeline comprehension. These two aspects help to perceive how the participants can work with ML pipeline representations to understand the code.

We chose this comparative approach because it enables us to observe differences in perception between the DAG representations and the code representations. This allowed us to evaluate the *fundamental value that the DAG provides over relying solely on the source code*. This method has been proven effective in evaluating software engineering artifacts [42, 158]; in our case, it reveals how DAGs supports participants' understanding and highlight areas where the they enhances or complements the information available in code. Moreover, to understand the specific usefulness of using an LLM-based approach, we added a baseline of a state-of-the-art approach for comparison, i.e., MLInspect [66].

5.3.2.1 *Collecting Experimental Objects*

We focused our analysis on the inspection of the source code of ML pipelines and their corresponding DAGs. Our initial plan was to reuse the 29 pipelines from the previous quantitative study; however, MLInspect's reliance on API matching (primarily for `pandas` and `scikit-learn`) limited its applicability. Only one of our prior pipelines could be processed by MLInspect. To ensure a fair comparison, we instead selected the four pipelines included in MLInspect's original study [66], plus the additional compatible pipeline from our previous quantitative study, yielding five scripts in total. While this choice narrows the scope, it highlights a key insight: API-based methods are restricted to known libraries, whereas CLAIRE's semantic analysis can handle diverse ecosystems (e.g., TensorFlow, PyTorch, Keras).

5.3.2.2 *Participant's Recruitment*

We prioritized recruiting participants with specific ML engineering expertise rather than aiming for broad generalizability, as finding large numbers

of fully qualified experts is difficult. Therefore, we recruited experienced ML engineers from two distinct research institutions as our study participants. The first is a public company with over 2,000 employees, comprising multiple research units, including a dedicated ML unit with more than 70 employees. The second is a national research center employing over 10,000 individuals across diverse sectors, such as aerospace, energy, transportation, and digital economy; its ML unit has approximately 60 employees. From these outreach efforts, we successfully recruited 25 ML engineers. We deemed this sample size sufficient for a qualitative evaluation, where depth of insights from knowledgeable participants is prioritized over quantity.

5.3.2.3 *Experimental Design*

The evaluation followed the process shown in Figure 5.5. We addressed two aspects: (i) the accuracy and usefulness of CLAIRE (based on our previously presented research questions) and (ii) the comparison of CLAIRE to the state-of-the-art pipeline MLInspect. For this purpose, we employed a mixed-subject design, which has been found to be an effective balancing with a moderate number of participants in other software engineering studies [147]. Specifically, this design ensures both a within-approach comparison (code vs. DAG) and a between-approach comparison (CLAIRE vs. MLInspect).

To control for ordering effects (i.e. bias due to the order in which participants are presented with either the code or the DAG), we divided participants into four groups. Groups A and B evaluated CLAIRE and ML-Inspect following two different sequences: Group A in the normal order (Code → DAG) and Group B in the reverse order (DAG → Code). Groups C and D followed the same procedure but were assigned exclusively to ML-Inspect. This design allowed us to disentangle whether participants' perceptions of clarity and accuracy were influenced by the order in which the representations were presented.

Each participant, therefore, went through two inspection rounds, providing feedback after each one. The first inspection (Figure 5.5 - Stage

2) focused on a single representation, where participants were randomly assigned to analyze either CLAIRE's DAG, ML-Inspect's DAG, or the raw code. *This stage aimed to capture their initial understanding without any prior exposure to other representations.* In the second inspection (Stage 3), participants were presented with a different representation of the same pipeline. Their feedback at this stage was therefore impacted by the knowledge and insights gained during the first inspection, thereby enabling us to observe how prior exposure shaped their evaluation.

Table 5.2: Combination of inspections with participant groups

Stage 1 Inspection	Stage 2 Inspection
Group A: Evaluate CLAIRE first, then code	
CLAIRE DAG	Raw Code
Group B: Evaluate code first, then CLAIRE	
Raw Code	CLAIRE DAG
Group C: Evaluate ML-Inspect first, then code	
ML-Inspect DAG	Raw Code
Group D: Evaluate code first, then ML-Inspect	
Raw Code	ML-Inspect DAG

Preceding the evaluation, every participant was asked to provide information about their background experience. Next, participants of groups A and C first viewed the original script and assessed how well they understood its overall objective, then reviewed the corresponding DAG and evaluated the same aspect of clarity. Participants of groups B and D completed the tasks in reverse order, first viewing the DAG and then the corresponding script.

5.3.2.4 *Results*

Our set of study participants represents a diverse group of expertise within the field. Their years of experience are balanced and spread, with 39.2% of the participants having up to two years of experience, 42.4% have between three and five years, and 17.8% have six to ten years. Moreover, we asked the participants if they had already experienced inspecting ML

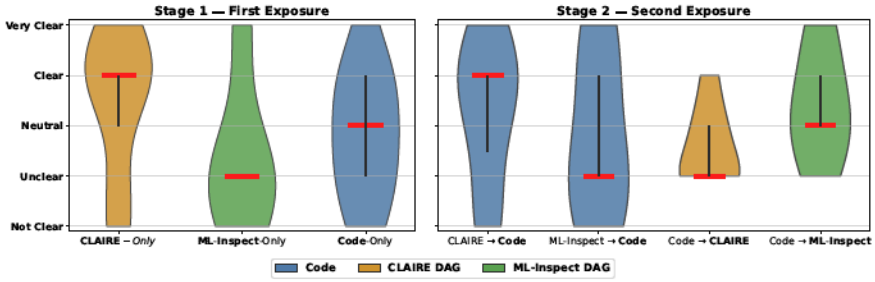


Figure 5.6: Degree of clarity of goal understanding of the pipeline through the three representations. Stage 1 represents the answers after participants had only inspected either the code or the DAG representation. Stage 2 represents the results after the participants had inspected both the code and the DAG representation. Colors and bold labels represent the representation of reference for which the clarity is reported.

pipelines through the use of graphical representations. Eight participants stated to have had experience in inspecting graphical representations of ML pipelines, particularly through AIRFLOW[4] for the design of data preprocessing operations and TENSORBOARD[5] for visualizations of neural networks and other ML models. The following sections describe the participants' perspective on the DAG representation, which helps address the two research questions of the study.

Figure 5.6 reports the participants' perceived clarity levels after viewing either (i) only the code (Stage 1), (ii) only the respective graph (Stage 1), or (iii) both code and graph in the order of their respective groups A, B, C and D (Stage 2).

In Stage 1, CLAIRE's DAG received the highest clarity ratings, with a median of 4, suggesting favorable evaluations. Its relative Inter-quartile range (IQR) is 1, stating consistent positive perceptions among the participants. By contrast, ML-Inspect's graph was rated consistently lower by participants (median: 2, IQR : 0). Interestingly, the code-only condition fell in between (median : 3, IQR : 2), though responses were more varied. These

[4]AIRFLOW: https://airflow.apache.org

[5]TENSORBOARD: https://www.tensorflow.org/tensorboard/get_started?hl=it

patterns indicate that participants tended to perceive CLAIRE's representation as clearer than the alternatives. In Stage 2 (sequential exposure), CLAIRE's clarity ratings remained comparatively higher in some conditions. Participants moving from CLAIRE to code reported little change in clarity (median : 4, IQR : 1.5), whereas those starting with ML-Inspect noted only modest increases when later viewing code (median : 2, IQR : 2). When code was presented first, subsequent exposure to CLAIRE yielded lower clarity ratings (median : 2, IQR : 1), while code followed by ML-Inspect produced slightly higher evaluations (median : 3, IQR : 1). Taken together, these observations suggest that participants generally perceived CLAIRE as clearer than both code and ML-Inspect, particularly when encountered as the first inspection.

Additionally, we asked participants to provide their perception on how useful the individual components of the DAG were (nodes, edges representing logical flow, data flow, and execution order). All results are collected in a Likert scale, as shown in Figure 5.7. Particularly, participants expressed markedly different perceptions of the two tools. For CLAIRE, the majority rated the representation positively, with 56% selecting *Very High* and a further 28% selecting *High*. In contrast, evaluations of ML-Inspect were much more evenly distributed across the scale, with only 10% rating the usefulness as *Very High* and 30% rating the clarity as *Medium*.

Therefore, we asked the participants to provide open feedback on the strengths and weaknesses of the DAGS. Participants agreed on the overall usefulness of graphs for both DAG representations as a tool for aiding comprehension, highlighting that graphical elements help to ease the understanding of complex parts of the code, especially where code comments are not provided. Particularly, looking at the set of answers related to CLAIRE's DAG inspection, seven participants specifically highlighted the adoption of logical flow edges as being very useful. Moreover, six participants highlighted the usefulness of the node titles. Participants reported that high-level sentences describing the operations represented in nodes help to catch the main activity performed in a part of the code. A subset of these participants additionally stated both elements were useful in combi-

nation, leading to a comprehensive representation that the code alone can not give, especially if comments are not present. One participant states: "*The edges were really helpful as well as the names of the node, the code does not have comments, and as such makes it harder to understand each step, with the DAG this becomes easier.*".

Regarding ML-Inspect DAGs, participants also highlighted some interesting points. Three participants noted the usefulness of the data nodes' separations. Particularly relating to the activities of training/test data splitting into two partitions, participants noted that the DAG representation is useful to understand the flow of the two partitions of the data over the executions. For example, one of the participants stated: "*The splitting of the dataset into the specific elements that were used to train the classifier and getting the labels was very intuitive.*" Moreover, four participants noted that the node and edge representations are extremely useful, in line with the representation of CLAIRE's DAGs. Regarding pitfalls and negative aspects of CLAIRE's DAG inspection, we noticed that the granularity of nodes is a commonly mentioned limitation of our representation. Five participants noted the need for more detailed information within the DAG representation to understand the details fully. They observed that a single node contains code related to more than one activity. For example, one of the participants remarks: "*The DAG in its current stage is accurate, but maybe needs to separate the code into smaller sections.*"

After examining both representations (DAG and code), participants rated the accuracy with which the DAG represented the underlying code on a five-point Likert scale ranging from *Very Low* to *Very High* (Figure 5.8). Both CLAIRE (brown) and ML-INSPECT (green) obtained a median rating of *High* (4), suggesting that both DAGs are a mostly accurate representation of the code. The spread of the ratings, however, shows important differences. For CLAIRE, values extended into the category *Very High* (IQR: 0.75), indicating that several participants judged its DAGs as highly accurate. In contrast, ML-INSPECT's ratings leaned more toward the lower end of the scale (IQR: 0.75), reflecting a tendency to evaluate its DAGs more conservatively.

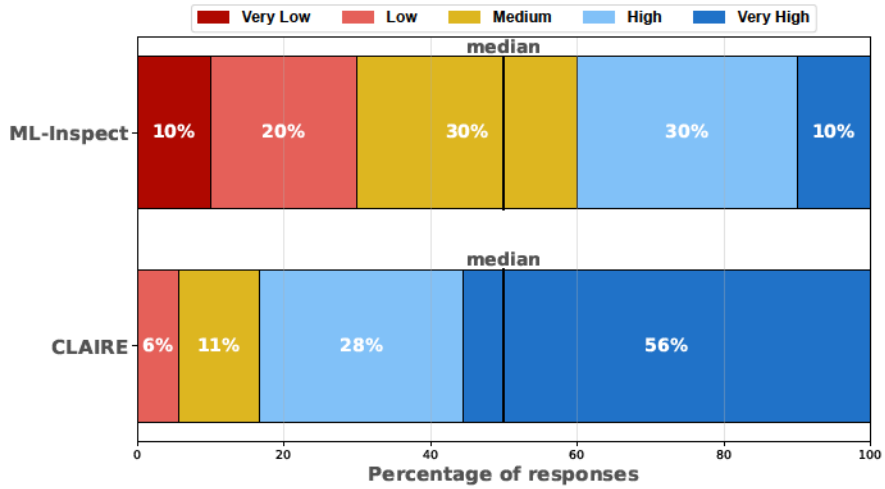


Figure 5.7: Usefulness of the DAGs elements reported by the participant during the artifact inspection.

5.3.2.5 *Implications of the Results*

The qualitative analysis of participants' perception highlighted two main insights: the trade-off between the level of detail and the clarity of DAG representations, and the usefulness of specific visual elements in DAGs.

Clarity of DAG Representations. Our clarity evaluation (Figure 5.6) revealed that participants found CLAIRE-generated DAGs useful for acquiring an initial understanding of ML pipeline characteristics. CLAIRE produced more synthetic graphs with fewer nodes than ML-Inspect, resulting in less complex representations that reduced information overload. This simplification was especially valuable when participants inspected the DAGs before reviewing the code, as it allowed them to catch the overall flow of the pipeline. However, when the code had already been inspected, the added value of CLAIRE's DAGs was more limited. In contrast, ML-Inspect fine-grained DAGs. While participants found these representations more difficult to interpret during the initial inspection, they became more useful after reviewing the code, as they already had a base knowledge of the context of the pipeline. ML-Inspect DAGs has been found useful to deepen the understanding of the code, highlighting information that is not visible

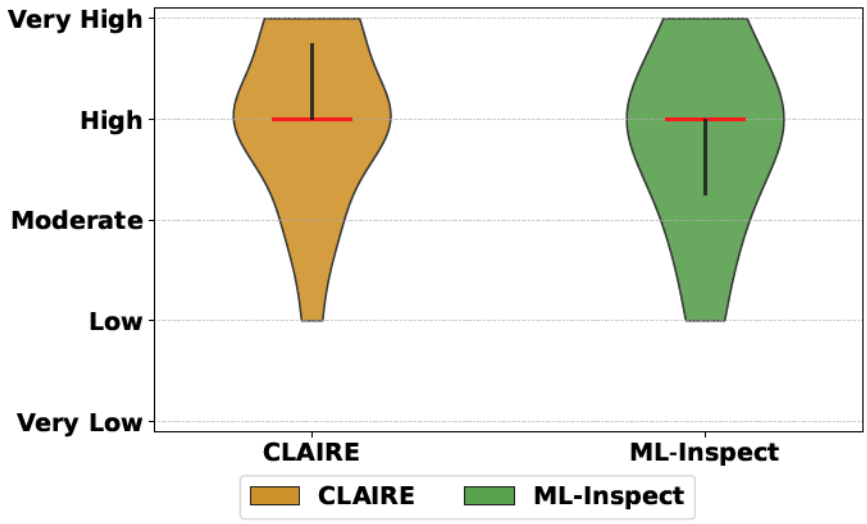


Figure 5.8: Degree of adherence of DAG representation to the respective ML pipeline Code.

in the code. These results suggest that the appropriate level of granularity depends on the inspection task: high-level DAGs are more effective for preliminary comprehension, whereas detailed DAGs are better suited for deepening understanding and revealing subtle elements in the pipeline.

> **Take Away Message 6.** CLAIRE DAGs are particular effective when presented before code inspection, while ML-INSPECT provide added value after code inspection.

Usefulness and Feedback on DAG Components. The evaluation of individual DAG components (Figure 5.7) described how participants perceived the strengths and weaknesses of the two approaches. CLAIRE's DAGs were rated positively overall, with participants emphasizing the usefulness of logical flow edges and descriptive node titles. These elements offered both structural guidance and semantic cues, compensating for missing code comments and enabling a more intuitive overview of the pipeline. At the same time, several participants noted that CLAIRE's coarser granularity sometimes grouped multiple activities within a single node, limiting the precision needed for fine-grained inspection. ML-Inspect, by contrast,

received more varied ratings. Its most appreciated feature was the explicit representation of data flow, which participants found particularly helpful for tracing training and test splits across the pipeline. Therefore, participants highlighted the importance of having a data-centric representation, increasing the awareness of data provenance.

> **Take Away Message 7.** Participants denote limited understanding of DAGs when the granularity of each node is too high or too low.

Overall, the analysis conducted suggests that the two approaches provide complementary forms of support. CLAIRE facilitates rapid orientation and lowers cognitive effort through abstraction and labeling, whereas ML-Inspect provides richer detail for tasks requiring careful analysis of data handling and execution. An effective inspection process may therefore benefit from combining both representations, aligning the choice of visualization with the goals and phase of the analysis.

5.4 LIMITATIONS AND FUTURE WORK

The findings of our study, other than highlighting specific findings on the usefulness of CLAIRE, denotes interesting limitations that are suggestions for future research.

Between different granularities of ML pipelines. The complementary nature of CLAIRE and ML-Inspect highlights distinct advantages when each approach is used. High-level representations, such as those generated by CLAIRE, provide a clear overview of the workflow without overwhelming the user. In contrast, fine-grained representations, such as those from ML-Inspect, reveal additional details that are critical for identifying hidden characteristics within the pipeline. Both types of representation are therefore essential for the effective inspection of ML pipelines. This contrast also exposes the strengths and limitations of current techniques. LLM-based approaches are particularly effective at creating explainable, lightweight representations that enhance readability and comprehension, but their high-level synthetic granularity limits to deeper analysis. Static analysis approaches, by comparison, capture detailed structural and data-centric

aspects that remain inaccessible to current LLM methods. Whether the task is to understand the overall workflow of a new experimental pipeline or to debug subtle quality issues, the level of detail required will vary, and the ideal granularity of the representation must adapt accordingly. While LLMs are useful in understanding the code, their context windows limit the amount of input they can practically reason on. In this manner, a context chunking strategy is used to face this issue. However, the promising clarity levels of CLAIRE shows the benefit on continuing work in this line of research, adopting novel strategies to increase the context knowledge of the LLM. From this perspective, we see a clear opportunity for future research: the introduction of adaptive DAG representations that can dynamically adjust their granularity. Such flexibility would allow ML engineers to start with a simplified view for orientation and then expand into progressively more detailed levels as their inspection deepens. Ultimately, this vision acknowledges the need for multiple types of ML pipeline representations, each tailored to the specific inspection goals and quality assurance tasks at hand.

Multi-agent LLM Systems. Another limitation of our approach is given by the adoption of a specific Multi-Agent system pattern: role-based multi-LLM. This design decision has been driven by the strong effectiveness given in Software Engineering Tasks, where the adoption of multiple LLMs and the addition of an additional agent checking and controlling different solution is a promising solution to avoid hallucinations and logical errors [93]. However, different patterns are published as soon as the maturity on the use of LLMs increase over time. Therefore, while this architecture gave us promising results when used with Mistral-24B, we believe that other architecture patterns can help as well. Future work will include also the use of complex architectures with different types of LLMs (e.g., reasoning models) and the adoption of other LLM roles (e.g., LLM-as-a-judge).

5.5 THREATS TO VALIDITY

Various aspects might have threatened the validity of the study. This section discusses these potential limitations and how we mitigated them.

Construct Validity. One of the main threats we faced regards the sequential comparative evaluation. Adopting participant evaluation to assess the usefulness of the study can lead to a series of issues related to the ordering of the inspection performed. A participant who tries to recognize the goal after every inspection can be affected by the previous inspection. Therefore, we tried to minimize this effect by counter-balancing the groups of participants in different orders, equalizing the effect. Moreover, we added analyses that are specific to the order in which the pipeline is inspected, to monitor and observe particular patterns that are affected by the order. Indeed, we noticed particular differences for which the CLAIRE and ML-Inspect approaches are useful in specific configurations and not generally. Finally, further combinations of the inspection of the order could also be applied through inspecting both graphs. However, the number of combinations of the inspections required to split the participants set into nine groups is too small for each configuration. For this reason, we applied a mixed-subject approach, limiting the order effect and avoiding too small representations.

Internal Validity. One of the main threats regarding the Internal Validity of our study relates to the adoption of the Mojo-Distance measure and the reference ground-truth dataset we built. While Mojo-Distance is able to understand the similarity of segmentation strategies, it is limited in its ability to determine whether the multi-agent LLM solutions accurately reflect the segmentation strategy employed by ML engineers. Therefore, while the similarity is high, we can assess the performance of LLMs, a solution that is totally different from our ground-truth, can still be an excellent solution. This threat is spread among all the studies that involve this type of code understanding tasks. However, we recognize the limitations of this evaluation, and we recall research efforts in finding alternative strategies to

effectively measure the performance of LLMs in measuring the correctness of ML pipelines.

External Validity. A first threat to the external validity is related to the set of participants that are selected in the study. Alternatives evaluated were to use online services to recruit participants (e.g., Prolific) and assess their knowledge. Inspecting ML pipelines requires specific expertise in ML engineering. Therefore, assessing the background expertise of the participant to observe their eligibility requires a lot of details that definitely increase the effort to conduct the experiments. Moreover, distributing the survey to reach a higher sample of participants in an uncontrolled manner can lead the participants to be affected and tempted to use external tools during the inspection (e.g., ChatGPT). To avoid significant threats to our internal validity, our design choice limited our evaluation sample to 25 participants from our network. Conversely, through a specific questionnaire and a known organization background, we assessed that the participants are effectively ML Engineers.

5.6 CONCLUSION

We introduced CLAIRE, an automated system that utilizes multi-agent large language model (LLM) strategies to create directed acyclic graph (DAG) representations of machine learning pipelines. Our mixed-method evaluation revealed that segmentation quality is highest with powerful models like Mistral-24B, while ensemble models tend to perform at the level of their strongest member. The user study further indicated that practitioners find CLAIRE's DAGs—specifically the node labels and logical edges—clearer and more useful for initial understanding than raw code or existing tools. These findings highlight both the potential and the current limitations of LLM-based multi-agent architectures for software comprehension. Future work should focus on validating the correctness of edges, scaling to larger and more diverse pipelines, and exploring adaptive DAGs that can vary in granularity to meet different developer needs.

6

DETECTING ML-SPECIFIC CODE SMELLS IN ML-ENABLED SYSTEMS

6.1 INTRODUCTION

Machine Learning (ML) evolved through the emergence of complex software integrating ML modules, defined as ML-enabled systems [99]. 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 [34] 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 [52]. In the last decades, researchers have been studying code smells in legacy code from multiple perspectives [10, 127], identifying them as one of the main precursors of defects and code instability [57, 58, 78, 120]. More recently, Sculley *et al.* [161] 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.* [24] and Van Oort *et al.* [187] 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. [147] further examined the impact of nine AI Technical Debt (AITD) issues, revealing their sig-

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

nificant 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 [88]. 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.* [206] 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 data handling, leading to the so-called *'Unnecessary Iteration'* smell, as shown in Listing 6.1 [206]. 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 6.1: Example of Unecessary Iteration Smell.

This chapter aimed to analyze the prevalence of ML-CSs, through the use of CODESMILE. Moreover, analysis on the characteristics of ML projects are collected to inform their relationship with ML-CSs, studying difference (i) between the prevalence of smells and whether (ii) the size of the project, (iii) the adoption of continuous integration differs in terms of ML-CSs occurrences.

Based on the considerations above, this chapter, part of our study [144], aims to investigate the prevalence of ML-CSs by conducting a large-scale mixed *confirmatory* and *exploratory* study. We focus on analyzing the prevalence of ML-CSs. By leveraging a dataset of over 400k commits across

337 projects coming from the NICHE dataset [196], 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. 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, offering actionable insights into their characteristics;

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. Statistical analysis on the prevalence of ML-CSs, 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.

4. An online replication package [143], 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 chapter. Section 6.2 presents the methodology of our study, detailing the research questions, dataset preparation and ML-CS

detection. Section 6.3 reports the key findings of our investigation. Section 6.4 discusses potential threats to the validity of our study and the measures taken to mitigate them. Finally, Section 6.5 concludes the paper and outlines avenues for future work.

6.2 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 [12].

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

> **RQ1:** How prevalent are ML-specific code smells 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 [15, 120], 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.

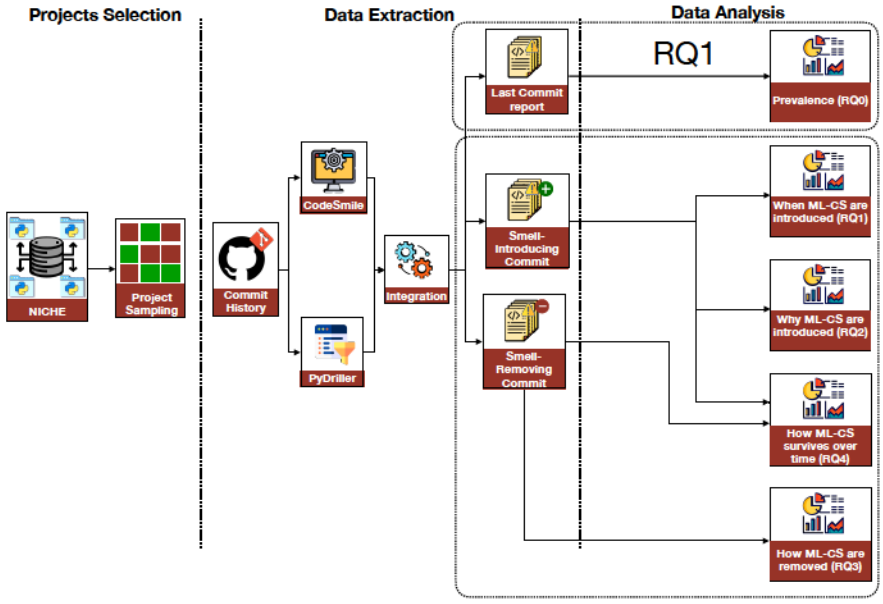


Figure 6.1: The process designed for the study.

6.2.1 *Dataset Description and Projects Selection*

We relied on the NICHE dataset [196] 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 no longer be 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

Table 6.1: 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

step, we identified 566 projects available out of the 572 and 1,110,689 commits. Table 6.1 shows the descriptive statistics on the variables "Stars", "Commits", and "Lines of Code" provided in NICHE [196]. 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.* [209], 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 173 projects, all containing less than 4,765 lines of code.

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.

Table 6.2: 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

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.[2] 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 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 6.2 shows the descriptive statistics for each size group. In addition to analyzing projects

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

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.

6.2.2 *Data Extraction*

After cloning the projects, we gathered fine-grained structural metrics using PyDriller, a framework helpful to analyze Git repositories [170]. 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.

6.2.3 *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. The tool implements rule-based criteria derived directly from the definitions provided by Zhang *et al.* [206].

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. [206]. 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 6.2 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 Tensor-



```
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

Figure 6.2: Example of How CodeSmile Performs Type Inference.

Flow. This process involves tracking variable definitions and operations throughout the code to provide accurate contextual information for smell detection.

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 6.2 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 6.2: 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 6.3. 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.

6.2.4 *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 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

Table 6.3: 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`.

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.* [206]. 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 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.

Table 6.4: 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

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 6.4. 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.

6.2.5 *Data Analysis*

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

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.* [206]. 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 Section 6.2.1, 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 [31], 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) [30].

6.3 ANALYSIS OF THE RESULTS

This section presents the results of the study for the prevalence of ML-CSs.

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 6.3, the distribution of these smells is highly uneven. 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*

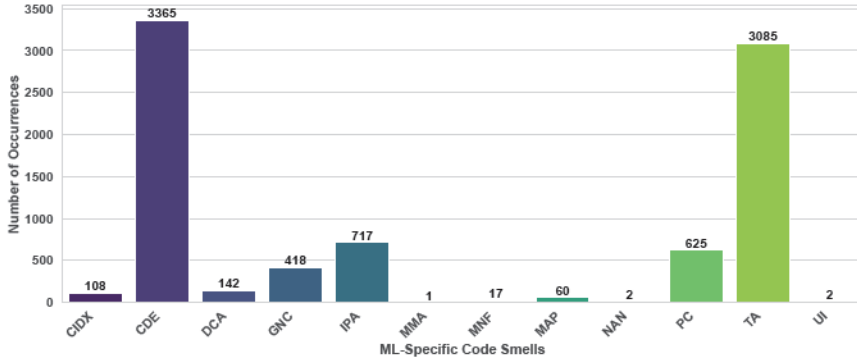


Figure 6.3: Number of occurrences of ML-CSs in the analyzed projects. The code smells considered are: *'Chain Indexing'* (CIDX), *'Columns and DataType Not Explicitly Set'* (CDE), *'Gradients Not Cleared Before Backward Propagation'* (GNC), *'In-Place APIs Misused'* (IPA), *'Matrix Multiplication API 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).

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 6.4 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

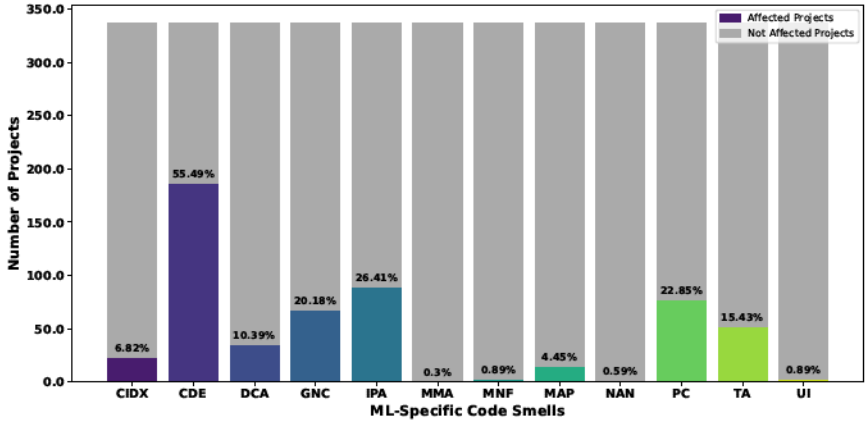


Figure 6.4: Number of projects affected by ML-CSs. The code smells considered are: *'Chain Indexing'* (CIDX), *'Columns and DataType Not Explicitly Set'* (CDE), *'Gradients Not Cleared Before Backward Propagation'* (GNC), *'In-Place APIs Misused'* (IPA), *'Matrix Multiplication API Misused'* (MMA), textsl'Memory 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).

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 6.4, 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 Mis-*

used'. 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 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 [156].

H0: DIFFERENCES AMONG THE PREVALENCE OF SMELLS. The results of the Wilcoxon tests, presented in Table 6.5, 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 (≥ 0.15, <0.33), medium (≥ 0.33, <0.47), and large (≥ 0.47) [30], 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-values. These comparisons exhibit medium to large effect sizes, underscoring the distinctiveness of this smell. For instance, the pairing of

Table 6.5: 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 *'Merge API Parameter Not Explicitly Set'* (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

'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***).

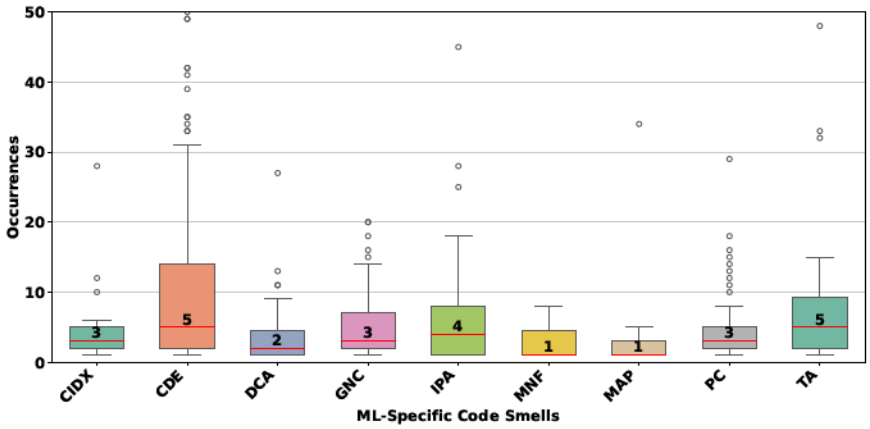


Figure 6.5: Distribution of the occurrences of ML-CSs in affected ML projects in every project analyzed. The median values of the distributions are reported in bold. The smells are: *'Chain Indexing'* (CIDX), *'Columns and DataType Not Explicitly Set'* (CDE), *'DataFrame Conversion API Misused'* (DCA), *'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).

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 in a negligible effect size (δ = 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 6.5, 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.

The results of the Friedman Test are reported in Table 6.6. 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 Explic-*

Table 6.6: 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	

itly 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 6.7 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.

Table 6.7: 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

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 nature 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$_1$.** 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.

6.4 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.* [206]. 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 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 [56, 152, 154] 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 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 [196], 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, 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.

Finally, other threats related to our subsequent analysis on evolutionary aspects of ML-CSs are reported in Section 7.4.

6.5 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.

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

PART IV

EVOLUTION OF AI-SPECIFIC TECHNICAL DEBT

7

ON THE LIFECYCLE OF ML-SPECIFIC CODE SMELLS IN ML-ENABLED SYSTEMS

7.1 INTRODUCTION

Recent research has noted the need for further investigations into the nature of ML-CSs [32, 206]. 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 [87], 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 chapter aims to provide a deeper understanding of the evolutionary characteristics of code smells, forming the basis for more effective prioritization strategies.

Based on the considerations above, this chapter aims to investigate the lifecycle of ML-CSs by conducting a large-scale mixed *confirmatory* and *exploratory* study. We focus on analyzing (i) when and why ML-CSs are introduced and removed, and (ii) how ML-CSs survive over time. By leveraging a dataset of over 400k commits across 337 projects coming from the NICHE dataset [196], 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 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 rela-

tively 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 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. An online replication package [143], 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 chapter. Section 7.2 presents the methodology of our study, detailing the research questions and the analysis pipeline. Section 7.3 reports the key findings of our investigation. Section 7.4 discusses potential threats to the validity of our study and the measures taken to mitigate them. Finally, Section 7.5 concludes the chapter and outlines avenues for future work.

7.2 RESEARCH METHOD

Our second part of the empirical study analyze 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 [12].

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

Figure 7.1 depicts the process we followed to address our research goal by addressing a preliminary and four main research questions. This study focus on the analysis on the four main research questions (from $\mathbf{RQ_1}$ to $\mathbf{RQ_4}$). The research method applied in this chapter relies on the top of the

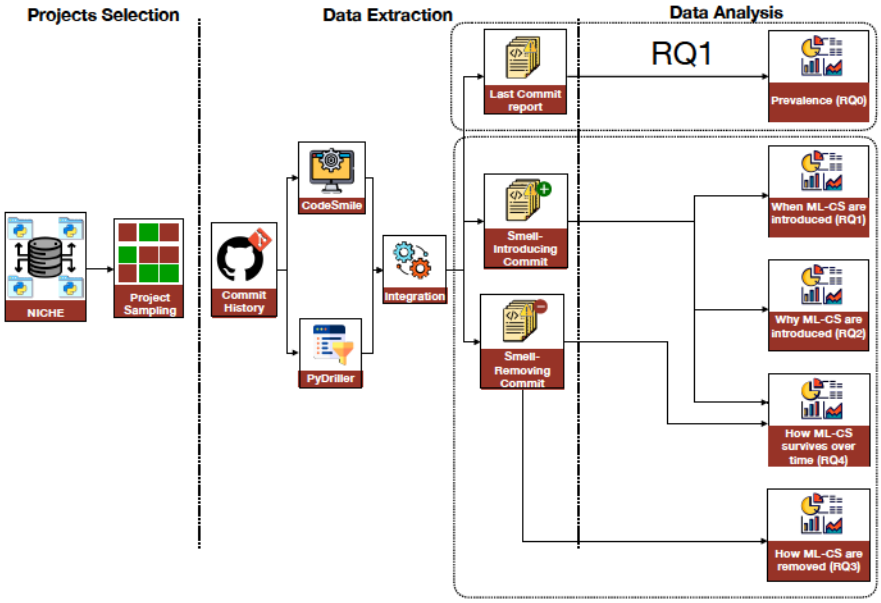


Figure 7.1: The process designed for the study.

> **RQ1:** 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 de-

termine 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.

> **RQ2:** 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.

> **RQ3:** 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.

> **RQ4:** 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.* [200] and the ACM/SIGSOFT Empirical Standards[1]. As for the latter, we used the *'General Standard'*, *'Data Science'*, and *'Repository Mining'* guidelines.

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

7.2.1 *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.

RQ$_1$: WHEN ARE ML-SPECIFIC CODE SMELLS INTRODUCED IN ML-ENABLED SYSTEMS? After the commit data extraction phase described in Section 7.2.1, 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 7.1 provides the segments and the value used for the segmentation.

Table 7.1: 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.* [182] 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 7.2. 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 performed a similar analysis from the

Table 7.2: 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.

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 7.1. 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 7.2 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.

7.2.2 *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 [143]. In addition, we

Table 7.3: 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

included the GITHUB repository link for CODESMILE: https://github.com/giammariagiordano/smell_ai/tree/main.

7.3 ANALYSIS OF THE RESULTS

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

7.3.1 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 7.3. 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 7.4, 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 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.

Table 7.4: 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

ANALYSIS ON ACTIVITY AND TIME-RELATED METRICS. Table 7.5 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, 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 7.6 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 rel-

Table 7.5: 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%**

atively 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. In-

Table 7.6: 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%

terestingly, *'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.

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.

7.3.2 *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. [182], 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 7.7 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.[2] 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.

[2] 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 7.7: 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%)**

👉 **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.

7.3.3 *RQ_3: When and how are ML-specific code smells removed in ML-enabled systems?*

Similar to the analysis performed in $\mathbf{RQ_1}$ and $\mathbf{RQ_2}$, 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 7.8 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 7.9 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 *'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

Table 7.8: 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

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 7.10. 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 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 Mis-*

Table 7.9: 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**

used' (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 7.11 summarizes the number of change-type operations (New Feature, Bug Fixing, Enhancement, and Refactoring) associated with smell-removing com-

Table 7.10: 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%**

mits 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. Enhancement 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*

Table 7.11: 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**

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.[3] 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.

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.

[3] 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

> 👉 **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.

7.3.4 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 7.12. 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 7.2). 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.

👉 **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.

7.4 THREATS TO VALIDITY

Main threats of validity related to CODESMILE and the set of projects selected are reported in Section 6.4. This section discusses possible threats to validity on the second empirical study, involving the the analysis from **RQ$_1$** to **RQ$_4$**.

THREATS TO CONSTRUCT VALIDITY A 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

Table 7.12: 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%

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. [183], Palomba et al. [125], and Ratzinger et al. [138]—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.

THREATS TO INTERNAL VALIDITY In the context of survivability analysis (**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

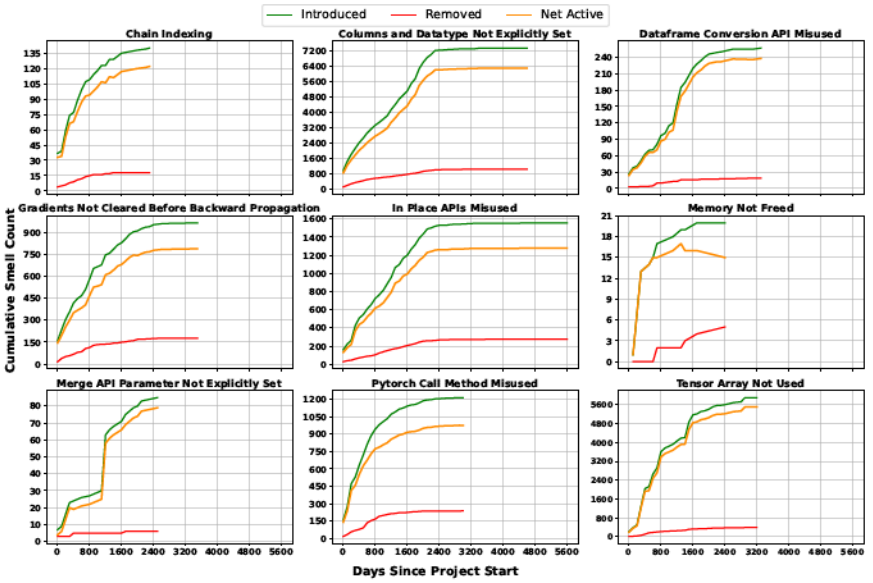


Figure 7.2: Cumulative trends of ML-CSs over the project lifetime. The figure presents the cumulative count of introduced (green), removed (red), and net active (orange) ML-specific code smells across projects over time. The code smells considered are: *'Chain Indexing'* (CIDX), *'Columns and DataType 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).

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 CONCLUSION VALIDITY A threat to the conclusion validity involves the computation of lifespan in **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.

7.5 CONCLUSION

This chapter explores ML-specific code smells (ML-CSs), detailing their lifecycle 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. 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.

A 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`[4], one commit message mentions correcting the mis-

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

use 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 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.

8

ON THE LIFECYCLE OF DL-SPECIFIC SELF-ADMITTED TECHNICAL DEBT

8.1 INTRODUCTION

Deep Learning (DL) has rapidly established itself as a transformative force across diverse industries. By leveraging advanced neural network architectures, DL provides powerful solutions to complex challenges, revolutionizing fields such as healthcare, autonomous driving, and natural language processing [1, 27, 47]. DL is a subfield of Machine Learning (ML), a broader field focused on developing algorithms that learn from data [64, 86]. ML techniques, including DL, are increasingly integrated into traditional software systems, resulting in ML-enabled systems that incorporate at least one ML component [99]. Although the adoption of these technologies in software systems has spread significantly to provide novel services, they present unique challenges that can increase the effort required to maintain and ensure the high quality of the architecture of ML-enabled systems [111]. One of the main challenges that can negatively impact the quality of software systems, particularly DL-enabled systems, is related to technical debt [16]. This term refers to the issues arising when developers opt for quick fixes or suboptimal solutions during the development of machine learning models [102, 161]. Over time, accumulating technical debt can lead to significant quality degradation, affecting not only maintainability but also other aspects of DL-specific quality such as performance and system security [147].

Developers acknowledge the presence of these quality issues explicitly, often through Self-Admitted Technical Debt (SATD) [134], which highlights potential problems in source code comments or issue trackers [89]. Various methods have been proposed to help developers manage SATD,

noting that comments usually include recognizable keywords such as *'TODO'* and *'FIXME'* [100]. An example of introducing an SATD is shown in the Listing 8.1. Developers add comments to raise an issue and to describe the suboptimal solution they employed, using the keyword (i. e., "`TODO`") to warn of the need for further rework in the future, establishing a debt.

```
def get_dummy_inputs(self, device, seed=0):
# TODO: use tensor inputs instead of PIL, this is here just to leave
    the old expected_slices untouched
    image = floats_tensor((1, 3, 32, 32),
    rng=random.Random(seed)).to(device)@@
    image=image.cpu().permute(0, 2, 3, 1)[0]
    init_image =
    Image.fromarray(np.uint8(image)).convert("RGB").resize((64, 64))
    mask_image = Image.fromarray(np.uint8(image +
    4)).convert("RGB").resize((64, 64))
```

Listing 8.1: Example of the introduction of a SATD.

Therefore, DL-specific SATD presents unique challenges compared to conventional SATD because of the distinct characteristics of DL components. Developers must consider SATD from conventional systems and those specific to ML and DL software, which are particularly vulnerable to the accumulation of technical debt [161]. One notable example is the work conducted by Pepe et al. [129], which defined a taxonomy of 41 distinct types of DL-specific SATD specific to DL-enabled systems. These types of SATD encompass various phases of DL, including data, model, training, and inference.

While previous efforts have defined DL-specific SATD in ML projects by creating a taxonomy, they have not thoroughly examined the life cycle of these technical debts. Examining when SATD instances are introduced and determining how long they persist provides critical information on the dynamics of SATD management. Furthermore, by investigating the longevity and timing of SATD instances, we can better understand their impact on software quality, maintenance efforts, and the overall lifecycle of DL-enabled systems.

Based on this gap, this chapter presents a preliminary mining repository study to investigate the life cycle of DL-specific SATD in order to understand (1) when and for how long developers acknowledge the DL-specific TD and (2) what tasks developers perform when they acknowledge the presence of DL-specific SATD in DL-enabled systems. To achieve this goal, we build on top of the work done by Pepe et al. [129] by using 41 distinct types of DL-specific SATD and 40 Python open-source projects relying on TensorFlow or PyTorch. The study reveals that DL-specific SATD is predominantly introduced during the early and middle development stages, with minimal instances acknowledged in the later stages. Additionally, the lifespan of SATD varies significantly across different components. SATD related to *Hardware* and *Training* phases tend to last longer, while those associated with *API* and *Pipeline* phases are typically resolved more quickly. Specifically, most SATD were identified during feature implementation (64 instances) and bug-fixing tasks (55 instances), underscoring the prevalence of technical debt in these development activities. These insights highlight the nuanced dynamics of DL-specific SATD, emphasizing the need for tailored strategies to manage and mitigate effectively. This preliminary work is important for software engineering researchers and developers who want to understand the life cycle of DL-specific SATD and the activities performed when the SATD is recognized in DL-enabled systems, highlighting the need for tools and models to capture, track, and address DL-specific SATD across development phases and components, enabling targeted interventions. This study highlights the importance of SATD management during feature implementation and fixes, where these debts are most common. Additionally, variations in SATD duration across components underline the need for customized solutions, such as automated tools and long-term strategies to resolve technical debts.

8.2 RESEARCH QUESTIONS

To investigate when and how developers acknowledge the presence of SATD, our study adopts the Goal-Question-Metric (GQM) approach [12].

Our Goal. Analyze the lifecycle of DL-specific TD
for the purpose of investigating when and how SATD is acknowledged
from the point of view of developer and researchers
in the context of DL-enabled system development and maintenance.

In particular, researchers are interested in the life cycle of DL-specific SATD to develop approaches for identifying, analyzing, and mitigating DL-specific SATD. Developers are interested in the analysis of the SATD to understand how to reduce risk and maintain system quality. To achieve our research goal, we focus on two main research questions **(RQs)** that guide our study. The **(RQ_1)**, seeks to identify when developers most frequently signal SATD during the software lifecycle (e.g., hardware, data, or training) and how long these issues remain unresolved. Understanding when SATD were first recognized provides insight into developers' behavior in managing technical debt and their interest in resolving it. In particular, we ask:

> **RQ1:** When and for how long do developers acknowledge DL-specific TD in DL-enabled systems?

After assessing the lifecycle of DL-specific SATD, the **(RQ_2)** aims to identify the specific actions that developers take when they recognize DL-specific SATD. These insights provide an understanding of whether SATD arises during feature development, bug fixing, or other activities.

> **RQ2:** What tasks do developers perform when acknowledging the presence of DL-specific TD in DL-enabled systems?

To design our process and analyze the results of our mining study, we followed the guidelines specified in the "Repository Mining" and "General Standard" categories of the ACM/SIGSOFT Empirical Standards [137].

Table 8.1: Definition of SATD categories.

Category	Description	Example
Hardware	This category accounts for SATD dealing with hardware components, e.g., GPUs or TPUs, used for training the DL model or during inference once the trained model is deployed in a production environment.	`TODO: do something with loras and offloading to CPU`
API	This category accounts for SATD due to the usage and integration of DL frameworks.	`TODO: support native HF versions of MusicGen.`
Data	This category groups SATD related to the format/shape of the data used to train a model.	`TODO: maybe have a cleaner way to cast the input`
Model	This category includes SATD concerning the design and setting of the DL model.	`TODO: attention mask is not used`
Training	This category encompasses SATD related to the training process, including the selection of loss functions, initialization of model parameters, learning strategies, and suboptimal implementation of training logic.	`TODO: Retrieve the seeds from the model definition instead.`
Inference	This category includes SATD related to inappropriate post-processing of the outcome generated when running the trained model, as well as using suboptimal prompting.	`TODO: add prompts to .yaml`
Pipeline	This category groups SATD dealing with the setting of the DL pipeline in terms of its design and optimization.	`TODO: enable higher-order gradients`

8.3 RESEARCH METHOD

This section describes our study research approach to answer our (**RQs**).

8.3.1 *Dataset Description*

The first step is to select appropriate projects within DL-specific SATD for our analysis.[1] For this purpose, we built our study upon the work of Pepe et al. [129] for a twofold reason.

On the one hand, the contribution of Pepe et al. [129] offers a curated dataset of 100 ML open-source projects, with 46 affected by SATD, providing a strong basis for our study. These projects are carefully selected to ensure relevance, focusing on Python-based repositories utilizing the two well-known DL frameworks, PyTorch[2] and TensorFlow[3]. This dataset was refined by excluding tutorials, toy projects, and repositories not primarily written in Python, ensuring the inclusion of active, high-quality, and representative projects. Moreover, the dataset guarantees that all projects included contain at least one DL-specific SATD.

On the other hand, Pepe et al. [129] contribute a comprehensive taxonomy of DL-specific SATD that classifies and contextualizes the technical debts found in ML projects. This taxonomy, in Table 8.1, classifies 200 SATD instances into 41 distinct types specific to DL. The types are further grouped under seven main categories, referring to aspects such as hardware configurations, training processes, data handling, and inference issues, all of which are essential in the lifecycle of DL-enabled systems.

To maintain the relevance of the SATD instances in the dataset, we conducted a verification process to confirm the data's reliability and validity. Specifically, we verified whether the GitHub repositories associated with the identified SATD instances were still accessible and actively maintained at the time of our study. Specifically, for each of the SATD instances analyzed with the validated dataset, we relied on Github API [59] to ensure the existence of the repository and the related commit for which the SATD has been identified. This step guarantees that all the SATD selected as the

[1] For the sake of clarity, we will consistently use the term SATD to denote TD issues that developers acknowledge, even if the term already includes acknowledgement by definition. Therefore, for the acknowledgement and introduction of SATD, we refer to the acknowledgement of TD issues through code comments, stating the presence of an SATD.

[2] https://pytorch.org/

[3] https://www.tensorflow.org

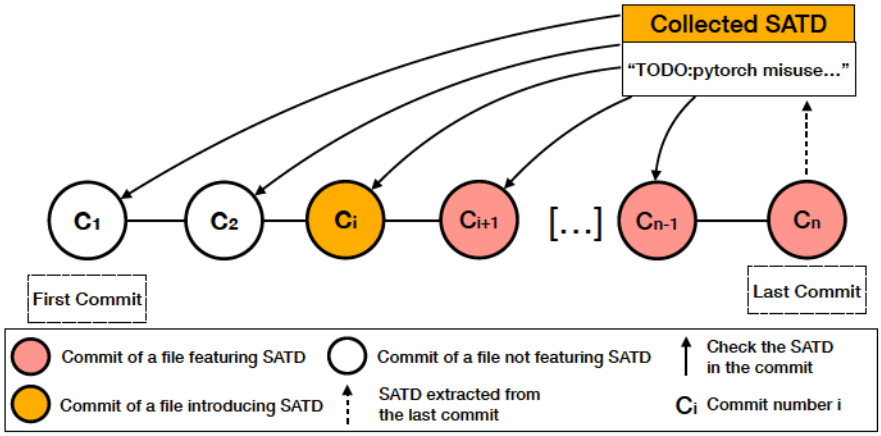


Figure 8.1: SATD introducing commit extraction

starting set of experimental objects are available to extract the commit history. Therefore, after this step, we selected a total of 185 SATD instances across 40 GitHub projects. By focusing on this refined subset, we ensured that the SATD instances and their corresponding projects were relevant to our study's objective of analyzing technical debt in DL-enabled systems.

8.3.2 *History Mining*

After selecting the project dataset and the catalog of DL-specific SATD, we analyzed the commits in Github projects. In particular, we proceeded to the commit extraction phase to identify the specific commits where each DL-specific SATD was first introduced into the codebase. To achieve this, we utilized PyDriller [4], a Python framework designed for mining Git repositories and analyzing commit histories. PyDriller allowed us to programmatically retrieve detailed information for each commit, including the commit hash, the lines modified, and the associated commit message. This simplified the data extraction process and ensured a systematic approach to managing the large volume of commits in projects. To determine the specific commit that introduced each DL-specific SATD instance, we

[4] PyDriller: https://github.com/ishepard/pydriller

employed a reverse tracing strategy similar to that performed by Tufano et al. [182], commonly used for analyzing the history of code smells. As depicted in Figure 8.1, the process began by leveraging the last commit to extract the SATD instance and to trace backward through the commit history. Leveraging the DL-specific SATD annotations, we systematically traversed the commit history in reverse chronological order, examining earlier commits to locate the exact introduction point of the DL-specific SATD instance. This backward traversal method ensured accurate mapping between DL-specific SATD instances and their origin commits.

When analyzing the commit history of a project P. Let a file F in a project P identify the introduction of a specific $s \in$ DL-specific SATD. Two scenarios can arise when encountering a commit that modifies the F file and the prior version of F does not include the specified DL-specific SATD:

1. **Last occurrence but not the first:** This commit marks the most recent occurrence of the $s \in$ DL-specific SATD in the file. However, it does not necessarily represent the first introduction of the DL-specific SATD, as it might have been added earlier in the commit history.

2. **First occurrence:** If no earlier versions of the F file, as examined through the commit history, contain the $s \in$ DL-specific SATD, then this commit is identified as the one that introduces s. In this case, it is considered the introducing commit for the DL-specific SATD.

This approach ensures that all occurrences of $s \in$ DL-specific SATD, whether it is the first occurrence or not, are accurately recorded and analyzed. Specifically, we defined the introduction commit c_i for the $s \in$ DL-specific SATD as the first commit in which the DL-specific SATD annotation appeared. Using this approach, we identified commits that introduce SATD across all file versions in the commit history, collecting a total of 5,337 commits.

8.3.3 *Data Analysis*

The following section describes our approach to analyzing the data collected to answer our research questions. To address ($\mathbf{RQ}_1$), we conducted a comprehensive analysis focusing on temporal data of the commit history associated with DL-specific SATD. Specifically, we developed a Python script that writes this information into a `csv` file. Once a $s \in$ DL-specific SATD is found in an F File, s is saved, and the commit history of F is traversed backward until the introduction of s. Additionally, in the saved dataset, we extract the current commit date and the previous commits, collecting the whole history for each file featuring the presence of a DL-specific SATD instance. Moreover, through the backward tracing of the commit history, we record the line number for every commit affected by a specific SATD instance to control the presence of the SATD also when the file changes along the history.

When SATD is introduced. To determine the introduction of any s, we measured the time and number of commits from the start of the project to the most recent commit, where it was still present. To analyze the introduction period of DL-specific SATD in their project lifetime, we examined their position in the commit history. Specifically, we calculated the difference in the number of commits between the repository's first commit and the commit that introduced DL-specific SATD. This analysis allows us to identify at what point in the project developers had reported s, measured by the number of commits made prior to its introduction. Collecting all SATD introduction positions over the project lifetime, we segmented the commit history of each project into three phases:

- **Early Stage**: The first 25% of commits in the project timeline.
- **Middle Stage**: Commits between 25% and 75% of the project timeline.
- **Last Stage**: The final 25% of commits in the project timeline.

This segmentation provided a perspective that highlights overarching insights on when DL-specific SATD-introduction commits occur within a project's lifespan.

How long SATD persists. Additionally, to examine the persistence of DL-specific SATD instances, we evaluated how long they remained unresolved by calculating the number of days between their introduction and the latest commit in the repository. It is important to note that the DL-specific SATD dataset used in this study includes only DL-specific SATD instances that are still present in the project's codebase. These instances were identified from the last active commit and traced backward through the commit history.

This analysis made it possible to determine how long developers continued to work on the project while leaving DL-specific SATD instances unresolved. Indicating the length of time DL-specific SATD instances were present in the system until the latest project version.

Combining the previous two analyses allows us to obtain information on commits that introduce SATD over the project lifetime, highlighting when they are introduced and how long they persist, answering to $\mathbf{RQ}_1$.

To address ($\mathbf{RQ}_2$), we focused on examining the commit messages associated with the introducing commits. Commit messages were analyzed using keyword-matching strategies to identify activities in which developers acknowledged DL-specific SATD. Based on the presence of keywords, the analysis categorized code modifications into four main types: feature development, bug fixes, enhancement, and refactoring.

The selection of these types has been found to be effective in classifying commit messages [182]. By using keyword-matching strategies, we systematically categorize SATD acknowledgment specific to DL in commit messages and analyze the activities that lead developers to indicate technical debt. Although this strategy helped us identify some categories, it did not allow for the identification of all categories. In detail, starting from a set of 185 DL-specific SATD commits, we extracted 62 commits under the four starting categories through keyword matching.

Therefore, the first two authors manually analyzed the remaining set of 123 commits by conducting content analysis [83]. Every inspector reviewed every comment, trying to classify it into one of the categories predefined previously. In case of disagreement, the two inspectors analyzed the results and started a discussion to reach an agreement. After this phase, 108 commits were manually categorized into the categories of Table 8.1, still leaving 15 cases of commits introducing DL-specific SATD uncategorized. Finally, a new round of inspection on the remaining part has been done to identify new categories among the meaningful commits. Therefore, a new category, named "chore", has been identified in addition.

Specifically, 11 out of the 15 commits are grouped into this category, identifying commit operations that aim to perform maintenance tasks that do not directly affect the software's functionality or features, such as updating dependencies or improving documentation.

Therefore, we analyzed the following categories:

1. **Bug fixes:** where DL-specific SATD were acknowledged during defect resolution or maintenance tasks, identified by keywords such as *fix* or *bug*;

2. **Enhancement:** where DL-specific SATD were acknowledged during the improvement of existing features or functionalities, often identified by keywords such as *improve* or *enhance*;

3. **Feature development:** where DL-specific SATD were introduced alongside the implementation of new functionalities, identified by keywords such as *feature* or *add*;

4. **Refactoring tasks:** where DL-specific SATD were tied to incomplete or deferred structural changes, often marked by keywords like *refactor*.

5. **Chore:** where DL-specific SATD were tied to routine maintenance, such as dependency updates or configuration changes, identified manually.

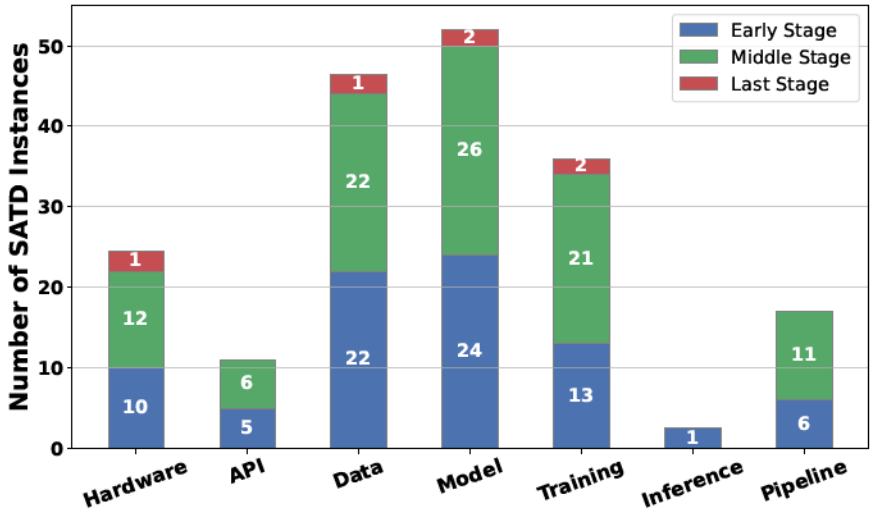


Figure 8.2: Frequency of SATD for each development stage.

Four commits introducing DL-specific SATD were not categorized and had no meaningful description; to avoid possible misclassification, we excluded these commits from our analysis. Finally, we classified 181 SATD-introducing commits into their specific five categories. Collecting the occurrences of commit categories that introduce allows us to investigate what are the activities that lead developers to recognize and denote DL-specific SATD, answering to $\mathbf{RQ_2}$. The entire process to reproduce our work is in the online appendix [142].

8.4 RESULTS

In this section, we report quantitative insights from the repository mining study to address the **RQs**:

8.4.1 RQ_1: *When and for how long do developers acknowledge DL-specific SATD in DL-enabled systems?*

Figure 8.2 presents the boxplot of the distribution of DL-specific SATD instances across the early, middle, and last stages of development for various components within DL-enabled systems across 40 projects. The analysis reveals that DL-specific SATD is most frequently acknowledged during the middle stage (98), followed by the early stage (81), and only minimally in the last stage (6). This trend underscores that developers primarily recognize and address technical debt early and during the refinement phases of the lifecycle.

Considering the totality of all projects, the *Model* phase has the highest total D-specific SATD instances (52), with 26 cases in the middle stage and 24 instances in the early. Only two are in the last stage. The *Data* phase is the second most frequent (45), with 24 in the middle stage, 22 instances in the early stage, and only one in the last stage.

The *Training* phase follows, with a total of 36 instances, distributed in the middle stage with 21 instances, while the early stage accounts for 13 instances, and only 2 are acknowledged in the last stage. The *Pipeline* phase shows 17 instances in total, with 11 occurring in the early stage and 6 in the middle stage, with no debt reported in the last stage. In contrast, the *Hardware* phase accounts for 22 instances, mainly concentrated in the middle stage (12 instances) and the early stage (10 instances), with none reported in the last stage. The *API* phase has the lowest number of instances among the major phases, with a total of 11 cases: 5 in the early stage, 6 in the middle stage, and none in the last stage. Finally, the *Inference* phase shows only one instance in the early stage.

The findings reveal a trend where most DL-specific SATD is reported in the early and middle stages, with only residual debt persisting into the last stage across most components.

The violin plot shows in Figure 8.3 the distribution of SATD lifetimes, measured in days, across various components of the DL-enabled system. The diagram reveals the variability in the persistence of DL-specific SATD

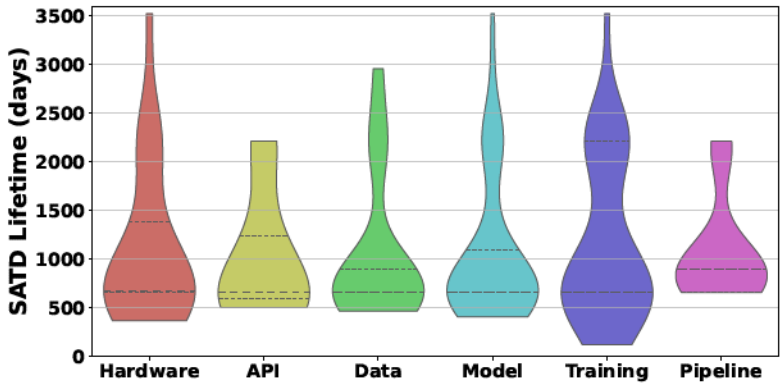


Figure 8.3: Distribution of SATD Lifetime since their introduction.

across components. For example, *Hardware* and *Training* phases exhibit the broadest distributions, with some instances persisting for extended periods, indicating that technical debt in these areas can remain unresolved for years. The median lifetimes for these components suggest that resolving such debt is often delayed, likely due to the complexity or lower prioritization of these issues. In contrast, components such as *Pipeline* and *API* phases exhibit a constrained distribution with shorter lifetimes. This indicates that DL-specific SATD in these areas is typically addressed more promptly and consistently. Meanwhile, *Model* and *Data* phases fall somewhere in between, with DL-specific SATD lifetimes showing moderate distributions and occasional instances of long-lasting debt. This reflects a mix of challenges that are resolved at varying rates. Overall, the plot highlights the differing persistence of DL-specific SATD across components, emphasizing the need for targeted strategies to tackle long-lived technical debt.

Table 8.2: Distribution of SATD Types Across Categories. Bug Fix (B), Feat (F), Enhance (E), Chore (C), Refactor (R)

Category	B	F	E	C	R	Total
Hardware	7	5	**8**	1	1	22
API	3	**4**	2	0	1	10
Data	**16**	15	8	5	0	44
Model	14	**20**	14	1	2	51
Training	6	**15**	6	2	7	36
Inference	0	**1**	0	0	0	1
Pipeline	**5**	4	3	2	3	17
Total	55	**64**	41	9	12	181

☞ **Answer to RQ$_1$.** Developers mainly identify DL-specific SATD during the early (81 instances) and middle (98 instances) stages, with few instances (6) in the final stage. SATD is most common in the *Model* and *Data* and less in the *API* and *Inference* stages. The duration of SATD varies significantly, with the *Hardware* and *Training* phases having the longest-lasting issues. In contrast, the *Pipeline* and *API* phases are resolved more quickly. Generally, developers address most SATD by the refinement phase, although some problems persist longer in certain components.

8.4.2 *RQ$_2$: What tasks do developers perform when acknowledging the presence of DL-specific SATD in DL-enabled systems?*

Table 8.2 reports the occurrences of the commit messages that introduce the DL-specific SATD, grouped in the seven categories. Analyzing the commit messages, we found a high occurrence (64 instances) of DL-specific SATD introduced during the *implementation of a new feature* operation. Still, many DL-specific SATD are consequently introduced during a *bug-*

fixing operation (55 instances) and operations related to the *enhancement* of the system (41 instances). Conversely, a lower amount of instances are introduced during a *refactoring* (12 instances) or a *chore* (9 instances).

DL-specific SATD is almost equally present across different operations categories. However, the analysis varies significantly when considering the specific types of DL-related SATD. From the set of commit messages analyzed, we found that 8 of the 22 *Hardware* related DL-specific SATD are introduced during an enhancement operation, resulting in the most recurrent in this category. Data-related DL-specific SATD usually appears when a *bug-fixing operations appear*. For instance, in the project *huggingface/diffusers*, during a bug-fixing operation related to the feature extraction component, a data-related SATD has been denoted by the developer's raising the absence of a scaling operation [43].

The most recurrent operation that introduces Model-related SATD is the implementation of the new feature with 20 instances. From the same commit operation related to the release of a new version of structured prompting, retrieved in the project *microsoft/LMOps*, we found the introduction of the developers of four Model-related SATD, raising problems related to the encoding components of transformers-based models (e. g., "*# TODO: we should reuse the pretrained model dict which already has mask*") [44].

Training-related SATD is mostly introduced during the implementation of a new feature, and 15 instances were found for this operation. Also, in this case, the addition of a new feature is denoted by developers to cause problems in the components that are involved in the training phase. Specifically, a developer denoted the introduction of a dummy encoder for the RoBERTa model to fix the non-determinism model (i. e., "*# TODO: remove after fixing the non-deterministic text encoder*") when introducing a new component in the system [45]. Among the 17 instances of pipeline-SATD, we found that every commit-operation type introduces at least one SATD of this category, with a slightly higher tendency when performing bug-fixing operations, with 5 instances. Finally, the unique SATD related

to the inference of DL models was found to be introduced during the implementation of a new feature.

In summary, there is a higher tendency to highlight DL-specific SATD when a new feature is implemented. Moreover, four of the categories are found to be introduced during this type of commit operation. At the same time, *Pipeline*, *Hardware*, and *API* SATD instances are occurrences that are distributed among different categories of commit operations.

> 👉 **Answer to RQ$_2$.** Developers more frequently introduce DL-specific SATD during feature implementation (64) and bug fixes (55). In contrast, SATD is less commonly associated with refactoring (12) or chore categories (9). DL-specific SATD introduced also varies based on the activity: *Data* SATD is often encountered during bug fixes (16), while *Model* and *Training* SATD are typically introduced during the development of new features (20 and 15 respectively).

8.5 THREATS TO VALIDITY

This section outlines the potential limitations and our strategies for addressing them.

External Validity. One of the main challenges is the representativeness of the data sample used in our investigation of DL-specific SATD. To mitigate the threat to generalizability of the dataset, we relied on the study of Pepe et al. [129] consisting of 40 open-source Python projects using TensorFlow and PyTorch and 185 DL-specific SATD types and applied a verification process to confirm the validity of the data. Although this sample provides a basis for our investigation, it may not reflect the diversity of DL-specific SATD types across software systems and contexts, since it does not include resolved SATD instances. Therefore, replications are needed to evaluate our preliminary observations, expanding the study to a larger scale. Additional investigations allow for the identification of additional DL-specific SATD types and the discovery of patterns among them. An additional threat concerns the diversity of the ML Projects affected by DL-specific SATD. To mitigate this threat, the Pepe et al. [129] dataset contains only projects

that have at least one SATD instance, ensuring the reliability of the dataset. Additionally, our study focuses on PyTorch and Tensorflow projects, given its widespread use.

Construct Validity. One potential threat arises from the classification of DL-specific SATD instances based on keyword matching and manual inspection. Misinterpretation of SATD categories or commit messages could lead to incorrect classifications. To address this, we leverage the taxonomy of DL-specific SATD [129] to ensure the use of consistent definitions, and we conducted content analysis sessions [83] to minimize bias in the manual categorization. Disagreements between inspectors were resolved through discussions to reach consensus. The data analysis was conducted by the first two authors, who have substantial expertise in AI software engineering. The first author is a Ph.D. student with over four years of experience in tech debt and software engineering for artificial intelligence (SE4AI). The second author is a Ph.D. student with over three years of experience in DL and SE4AI.

PART V

DISCUSSION, FURTHER ANALYSIS, AND CONCLUSION

9

CONCLUSION

The objective of this thesis was to investigate the characteristics, detection, and evolutionary aspects of AI-Technical Debt Issues (AITD). The research aimed to establish a solid foundation for exploring various AITD issues, beginning with the code, architecture, and data. For each type of debt, empirical studies are conducted to evaluate detection strategies and how debt issues evolve, providing a body of knowledge to increase AI practitioner awareness and guidelines to monitor accumulating debt. The contribution of this thesis is motivated by the quick shift toward adopting an AI-centered development approach. Limited knowledge on TD management tailored for AI-enabled systems highlights the need for strategies that can recognize, monitor, and address AITD. Research conducted includes several empirical methods, from ground investigation of practitioners' perception, to quantitative analysis on real-world open-source systems. In detail, every study is framed through the GQM approach for research [12], ensuring that every single research question is aligned with high-level goals. Empirical methods include literature reviews [150], survey qualitative studies [147], mining software repositories studies [141, 144], and mixed-method research [146]. All of these studies contribute a different perspective on the current perception of AITD issues in the field, examining both the state-of-the-art and the state-of-the-practice.

Key contributions of this work contribute to the field of Software Quality Assurance for AI [88], summarized as follows:

1. An extensive catalog of AITD issues, strongly connected to architecture, data, and code, enlarged with their prevalence, severity, and impact on quality aspects [144, 147, 150].

2. A comprehensive kit for detecting AITD issues, enabling the detection of one architectural AITD issue, 12 ML-CSs, and syntactic-based data smells [144, 146, 150].

3. Insights on the evolution of AITD issues, collecting when and how they are introduced and removed along the lifecycle of AI-enabled systems [141, 144].

4. Practical automated tools to enable researchers to detect ML-CSs automatically, and perform Pipeline Jungle Inspection [144, 146].

These contributions helped to increase knowledge of the AITD phenomenon, examining how it is characterized, how it can be detected, and how it evolves. In the following sections, we discuss the answers to our broad research questions, which inform the overall goal.

9.1 RQ_A: CHARACTERIZING AI-TECHNICAL DEBT

Across the several studies conducted and reported in different Chapters, it is noteworthy to state that AITD is **widespread**. Chapter 3 denoted high prevalence for part of the smells detected, such as *"Glue Code"* and *"Pipeline Jungle"*. Interestingly, more than 50% of the participants ended up with both AITD issues. Moreover, all the empirical study conducted in Chapters 6 and 7 denotes how incurring this type of debt is spread across various components and stages of machine learning (ML)-enabled systems. Out of 337 projects, we found more than 8,000 instances. Additionally, data-related analysis in Chapter 4 confirms that AITD extends beyond just code artifacts. Issues related to data quality, such as Extreme Value Smells and Missing Value Smells, are highly prevalent across the studied datasets, with 137 and 110 instances, respectively. Together, they reveal a high prevalence of suboptimal practices, including inefficient ones, as reflected in data, code, and software architecture.

Despite being widespread, AITD remains **largely under-recognized** by professionals in the field. Developers' perceptions, as collected in Chapter 3, showed varied variability strongly based on their knowledge and

skills in recognizing the issue. Participants with practical experience in AITD issues reported a higher level of severity, recognizing the critical consequences in these situations. However, the prevalence of these AITD issues is not reported by all participants. When examining their detection and management strategies, they are strongly limited in their management capabilities, primarily relying on manual inspection and ad-hoc refactoring. The absence of specialized monitoring tools and standardized terminology contributes to this problem, leading developers to normalize practices that actually undermine long-term quality. This neglect of AITD has serious implications for AI quality, as it limits awareness and remediation, allowing technical debt to accumulate unnoticed until it ultimately affects system reliability or fairness.

Finally, AITD is **multi-layered**, affecting different artifacts, activities, and quality aspects. *"Pipeline Jungle"*, as further analyzed in Chapter 3, denotes the continuous increase of the pipeline complexity, affecting all the artifacts involved in the process. Consequently, participants reported an expected impact of the AITD issue on 12 quality-related aspects, with a strong remark on maintainability, performance, and size. Data smells contribute directly to the structure of the data, but this debt is propagated among all subsequent artifacts engaged in data processing. If refactoring is not performed, it creates a cascade of debt toward data-dependent artifacts. Finally, ML-CSs introduce a challenge layer related to ML code, but spread across different activities, requiring ML engineers to be cautious with the practices adopted when loading data or defining the training cycle using gradient descent. It naturally affects several aspects of AI-specific software quality, including readability and bug proneness.

The introduction of AITD, although directly related to a single artifact, activity, or aspect within the ML building process, can have multiple and unpredictable effects. This characteristic is particularly defined by the CACE (Change Anything Changes Everything) principle [161], given the nature of AI-enabled systems. To conclude, all the studies that contributed to answering $\mathbf{RQ_A}$ in this thesis denote particular characteristics of AITD issues, resumable in **widespread**, **under-recognized**, and **multi-layered**.

> 👉 **RQ$_A$:** *What types of AI-Specific Technical Debt (AITD) occur, and how do they manifest in AI-enabled systems?*
> AITD is highly prevalent, difficult to recognize, and affects multiple layers of AI-enabled systems, spanning data, code, and architecture.

9.2 RQ$_B$: DETECTING AI-TECHNICAL DEBT

Chapter 3 denoted the limited capabilities of AI practitioners to have a predefined set of strategies or automated tools to identify and refactor AITD issues. Particularly, participants mainly relied on manual review and ad-hoc refactoring, hoping that these strategies could successfully resolve the problems encountered. This thesis contributes to adding a maturity layer to AITD management. Chapter 5 contributes to the definition of an LLM-based automated tool helping with the inspection of ML pipelines, i.e., CLAIRE. Results from this tool revealed high fidelity in representing the workflow process from ML code with an 83% median Mojo-Similarity with respect to the ground-truth representation. Moreover, participants who evaluated the adoption of CLAIRE for pipeline comprehension denoted a high level of clarity in auto-generated DAGs. This contribution serves as a preliminary basis for detecting pipeline-related AITD issues, particularly *"Pipeline Jungle"*. Regarding ML-CSs, this thesis contributes CODESMILE, a static analysis tool that can detect 12 ML-CSs. In this case, the tool demonstrates an interesting performance in detecting this type of AITD issue, achieving an 83% F1-score. Finally, open-source tools enable the detection of data smells related to the distribution and encoding of data, such as DSD [50]. Even if these tools altogether cover only partially the set of AITD issues in code, architecture, and data artifacts, they are a promising starting point for further research. Future studies can use these tools as a baseline to further improve detection performance.

☞ **RQ$_B$:** *How can AITD issues be automatically detected in real-world AI-enabled systems?*
Together, these tools demonstrate that AITD detection is feasible and can be approached at different levels, thus laying the groundwork for automated debt management in AI-enabled systems.

9.3 RQ$_C$: EVOLUTION OF AI-TECHNICAL DEBT

The analysis included in this thesis explores the evolutionary aspects of ML-CSs and SATDs, enabling the collection of dynamic information that is useful for understanding how management practices should be applied over time. Chapter 7 explored the lifespan of ML-CSs, from their introduction to their removal. Results indicate that certain ML-specific code smells, such as *In-Place APIs Misused* and *TensorArray Not Used*, are highly persistent within ML-enabled systems. These smells frequently persist across multiple releases, suggesting that maintenance activities often fail to remove them effectively. Instead, the results show that maintenance commits progressively introduce ML-CSs during the early stage of file modifications, and they remain unresolved over time. Similar findings are shown when analyzing the set of AITD issues acknowledged by developers (SATDs). Chapter 8 highlights how SATD are introduced among the lifetime of DL-enabled systems, for which two similar outcomes can be aligned to ML-CSs. Firstly, most of the SATD analyzed are introduced during the first 25% of commits, and additional SATD are acknowledged during the middle stages. More interestingly, even when developers explicitly acknowledge debt through SATD comments, they rarely act upon it, revealing a gap between awareness and remediation.

Therefore, both ML-CSs and SATDs are predominantly introduced during the early development stages and remain unresolved over time, reflecting a pattern of long-lived and cumulative debt. These two outcomes, when analyzed together and confirmed to reveal two types of AITD issues, leave several discussions open regarding the evolutionary nature. First, AITD issues continue to increase silently, with a high trend of increase

during the early stage of AI development. Moreover, even if recognized, ML developers leave unresolved issues, possibly due to their limited skills in resolving the debts. This particular detail is naturally connected to the finding of AITD management practices presented in Chapter 3, where AI practitioners identify ad-hoc refactoring as the most commonly used solution to address AITD. Therefore, the rapid increase in AITD issues highlights the need for a TD management kit, as its absence leaves an uncontrollable amount of debt among AI-enabled systems.

> 👉 **RQ_C:** *What are the lifecycle characteristics and persistence of AI-Specific Technical Debt (AITD) issues?*
> AITD follows a long-lived and latent lifecycle, where debts are introduced early, persist silently, and are seldom resolved—reinforcing the importance of continuous, automated management strategies.

9.4 LESSON LEARNT

All the studies that contributed to this thesis enlighten a specific aspect of studying AITD, where each of the issues presented has been a compelling challenge. Therefore, the following lessons learnt are described based on the experience and the outcomes obtained in the studies.

- 💼 **AI integration on software systems adds an important layer of debt.** As noted by Sculley et al. [161] and observed by Menzies [103], AI-enabled systems, although containing a single small component related to the AI model, exhibit peculiar differences in other elements as well. Many elements are related to data management and preparation for training models, while others are associated with Feature Engineering and Model Serving Infrastructure. All these components, although apparently similar to traditional code, are driven by the primary business service they provide, namely AI. This shift introduces numerous specific patterns and development practices that differ from traditional software systems and require specialized knowledge. The findings encapsulated in this thesis highlight a par-

ticular skill set that software developers need, combining AI-specific expertise with software engineering.

AI-Technical Debt issues have a different nature from Traditional technical debt. This thesis captures AITD, covering three types of debt: architectural, code, and data debt. Among the types of debt discussed in this thesis, observations are made on the high but subtle prevalence. Moreover, their comparison to traditional debt reveals notable differences. One of the most prevalent symptoms of debt is the presence of code smells [36, 122, 183], along with design issues [132] and architectural debt [201]. Most of these smells demonstrated the use of code, text, and historical information that informs the presence of a traditional debt. When facing AITD, instead, the set of artifacts extends over the simple code and architecture. It encompasses data, model-specific settings, and activities that are visible at a higher level of granularity, reaching the system level. Issues investigated in, for instance, Chapter 3 require a picture showing environmental information to obtain information about the control access data and observe the presence of *"Undeclared Consumers"*. We observed that our findings have both similarities and differences from those reported in studies of traditional code smells. Our analysis in Chapter **??**, 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. [182], 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. However, ML-CSs exhibit unique characteristics tied to the domain-specific nature of machine learning development. For instance, ML-CSs often arise from operations specific to ML workflows, such as data handling, model training, and pipeline construction, which differ significantly from the contexts in which traditional code smells emerge. Moreover, the granularity of

ML-CSs included in this study is lower than traditional code smells, reaching at most the method level. This particular difference can be reflected in the findings of the commit modification type when ML-CSs are introduced. Indeed, while traditional code smells are introduced from their creation [182], we noticed that ML-CSs are also introduced in the late stage of development.

Additionally, the shorter lifespan of ML-CSs, as highlighted in Chapter 7 regarding 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.

Analyzing SATD in traditional software systems, Potdar and Shihab [134] discovered that even after many releases, only less than 64% percent of SATD instances are resolved. On the same line, Bavota and Russo [13] also analyzed the lifespan of traditional SATD instances, discovering that these self-acknowledged issues remained unresolved for more than 1000 commits since the start of the project. In the context of SATD when adopting DL technologies (Chapter 8), we found that many of the DL-specific SATD instances considered in this study, especially for the *training-related* category, remained unresolved for longer periods.

These observations underscore that managing technical debt in AI-enabled systems requires strategies that go beyond those used for traditional software. The interplay between ML-specific smells, SATD, and the complexity of ML pipelines suggests that future research

should be aware on the complexities and the challenges arising in AI-enabled systems, with novel paradigms and prompt refactorings that run at the same speed of the continuous evolution of these systems.

- **AI can be the solution to maintain AI.** Studies included in the chapters of this thesis show high performance through the use of static analysis when analyzing the code, enabling us to build CODE SMILE. However, more complex AITD issues required the extraction and understanding of semantic information. For instance, observing *"Pipeline Jungle"* issues required extracting logical steps from the code to form an ML pipeline. Applying semantic analysis through static analysis or rule-based evaluation can be quite challenging, and, as with ML-INSPECT, it has significant limitations in the API that the rules cover. More generalizable approaches require the use of AI to extract and catch information based on the context of the code, providing structural information based on the semantics of the code. Moreover, even for ML-CSs, the limits on the precision of the rules are naturally evident from the approach's performance. AI can help further boost the performance of ML-CS detectors.

10

OPEN ISSUES AND NEXT DIRECTIONS

While this thesis provides a solid foundation for contributions, various challenges remain to be addressed in order to manage TD in AI completely. Specifically, the thesis contributed to exploring different types of debt horizontally, examining vertical aspects within a single debt type, and assessing the level of maturity required to manage each AITD issue. Therefore, open issues are explored under these three perspectives, examining the **breadth**, **depth**, and **maturity** of Technical Debt Management. In the following, we highlight the open challenges and define the next directions.

⚠ **Current State of AITD has a limited breadth on different debt types.** The contribution of this thesis has narrowed the knowledge on three types of debts: data, architecture, and code. However, novel types can also be observed with the acquisition of novel knowledge. Ronanki et al. [155] defined the concept of prompt smells as "semantic or syntactic characteristics of a prompt instance resulting from imprecise prompt engineering". Simon et al. [168] began investigating internal model characteristics, exploring the concept of *algorithm debt*. Moreover, as already explored by Bogner et al. [18] and Petrozzino [130], the concept of debt goes beyond technical issues. Adopting AI as a software service implies socio-technical concerns that can create more serious issues than economic factors. This issue is particularly emphasized when examining *fairness debt* [169]. Suboptimal design choices can introduce bias into critical activities of the AI development process, resulting in severe consequences and unethical outputs. These novel types of debts come from the continuous evolution of Artificial Intelligence, which is also spread in software engineering. This is strictly connected to the scope of defining Software Engineering for AI [185]. Regarding the research

field, technical debt management in AI-enabled systems should be expanded to encompass socio-technical concerns and provide strategies for addressing ethical concerns.

In conclusion, TD management should provide knowledge and support on novel technologies and novel aspects as AI continues to evolve, increasing its **breadth**.

⚠ **Current State of AITD has a limited depth on specific AITD issues.** The contribution of this thesis expands knowledge on several issues that contribute to debt accumulation, covering one AITD issue, 12 ML-CSs, and syntactic data smells. However, several AITD issues that we have explored remain in a preliminary state of definition. For instance, data debt is not limited to how data are represented but also in how they are maintained and processed over the AI-enabled system lifecycle. Several issues can occur when data are acquired, processed, and handled through the process. Akgül et al. explored this issue [2], denoting that even with additional details of data management activities, several issues covered by the definition of data debt remain unexplored. One of the challenging aspects to explore in data debt is related to data lineage and provenance, which requires proper documentation to track the flow of data throughout the development process of AI-enabled systems.

Therefore, AITD requires continuous research to investigate unexplored AITD issues, increasing the **depth** for each type of debt.

⚠ **Limited Maturity on Technical Debt Management.** If we compare AITD management at the current state to traditional technical debt, several challenges are still open. A recent literature review conducted by Moreschini et al. [108] highlights the current state of the evolution of technical debt, denoting a critical challenge: most research efforts stop at the level of TD identification. Limited knowledge is found across the different stages of maturity for TD management. One of the strong limitations remains in finding practices that address AITD issues, i.e., AITD refactoring. As also denoted in Chapter 3, AI

practitioners struggle to define generalizable practices that guide the resolution of AITD issues. Moreover, other levels of **maturity** in AITD management are still key points that need to be addressed, such as prevention, monitoring, and measurement [108].

The open issues found in this thesis allowed to define therefore the following next directions:

» **Toward Quality Assurance for GenAI.** Enlarging the scope of Technical Debt Management requires studies exploring novel debt that increase the breadth and the depth of AITD issues. Novel studies observe and characterize debt that occurs in LLM-enabled systems. Analog to existing studies, prompt engineering patterns can be further explored to define prompt smells based on readability metrics [155]. Moreover, recent trends start to examine the role of software architecture in Agentic AI systems [186]. Multi-agent systems may require additional considerations when using collaboration and communication patterns within them [93]. Poor communication between agents can lead to performance issues or excessive inference time. The role of TD Management is to define quality assurance and monitoring practices tailored for GenAI-enabled systems.

» **Toward Continuous Quality Assurance for AI.** Shifting from TD identification toward continuous TD monitoring under is therefore still an open challenge. Moreover, when shifting to a dynamic environment, other AITD issues arise, such as the continuous change in the data relationship over time (i.e., data drift). Current MLOps strategies prioritize the observation of models through continuous training and validation over a comprehensive set of quality assurance practices [204]. Future studies should integrate AITD management in MLOps monitoring toolkits, enabling continuous quality assurance for AI.

LIST OF PUBLICATIONS

The complete list of publications is reported below. The ∗ symbol highlights the publications discussed in this dissertation.

INTERNATIONAL JOURNAL PAPERS

J01 - **Recupito, G.**, Pecorelli, F., Catolino, G., Lenarduzzi, V., Taibi, D., Di Nucci, D., and Palomba, F. (2024). 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 ∗

J02 - **Recupito, G.**, Giordano, G., Ferrucci, F., Di Nucci, D., and Palomba, F. (2025). When code smells meet ML: on the lifecycle of ML-specific code smells in ML-enabled systems. Empirical Software Engineering, 30(5), 1-49. ∗

J03 - De Martino, V., **Recupito, G.**, Giordano, G., Ferrucci, F., Di Nucci, D., and Palomba, F. (2025). Into the ML-universe: An improved classification and characterization of machine-learning projects. Journal of Systems and Software, 230.

J04 - **Recupito, G.**, Opitz, D., Hecking, T., Di Nucci D., Palomba, F., and Felderer, M. "LLM-based Architecture to Create Graph Representations from Machine Learning Pipelines." In: Submitting to IEEE Transactions on Software Engineering (2025) ∗

J05 - **Recupito, G.**, Mazloomzadeh, I., Da Silva, L., Khomh, F., Di Nucci D., and Palomba, F. "AI Inherits New Debts: Exploring, Identifying, and Classifying AI-Specific Self-Admitted Technical Debt" In: Submitting to ACM Transactions on Software Engineering and Methodology (2025)

INTERNATIONAL CONFERENCE PAPERS

C01 - **Recupito, G.**, Pecorelli, F., Catolino, G., Moreschini, S., Di Nucci, D., Palomba, F., and Tamburri, D. A. (2022, August). A multivocal literature review of MLOps tools and features. The 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA) (pp. 84-91). IEEE.

C02 - **Recupito, G.,** Rapacciuolo, R., Di Nucci, D., and Palomba, F. (2024, April). Unmasking data secrets: An empirical investigation into data smells and their impact on data quality. In Proceedings of the IEEE/ACM 3rd International Conference on AI Engineering-Software Engineering for AI (pp. 53-63). *

C03 - Novielli N., Oliveto R., Palomba F., Calefato F., Colavito G., De Martino V., Della Porta A., Giordano G., Guglielmi E., Lanubile F., Quaranta L., G. **Recupito G.**, Scalabrino S. , Spina A., and Vitale A. (2024)" QualAI: Continuous Quality Improvement of AI-based Systems." In Proceedings of the RCIS 2024 Workshops (CEUR Workshop Proceedings, Vol. 3801). CEUR-WS.org.

C04 - Della Porta, A., De Martino, V., **Recupito, G.**, Iemmino, C., Catolino, G., Di Nucci, D., and Palomba, F. (2024). Using Large Language Models to Support Software Engineering Documentation in Waterfall Life Cycles: Are We There Yet?. ITAL-IA - Quarto Convegno Nazionale CINI sull'Intelligenza Artificiale, 2024.

C05 - La Gamba, D., Iuliano, G., **Recupito, G.**, Giordano, G., Ferrucci, F., Di Nucci, D., and Palomba, F. (2024, June). Toward a Search-Based Approach to Support the Design of Security Tests for Malicious Network Traffic. In Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering (pp. 624-628).

C06 - Novielli N., Oliveto R., Palomba F., Calefato F., Colavito G., De Martino V., Della Porta A., Giordano G., Guglielmi E., Lanubile F., Quaranta L., **Recupito G.**, Scalabrino S. , Spina A., and Vitale A. (2024).

Continuous Quality Improvement of AI-based Systems: the QualAI Project. In Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (pp. 603-607).

C08 - Palomba F., Voria G., Parziale A., Pentangelo V., Della Porta A., De Martino V., **Recupito G.** and Giordano G. (2025). Teaching Software Quality Assurance for Artificial Intelligence: An Experience Report. 51st Euromicro Conference Series on Software Engineering and Advanced Applications (SEAA).

C09 - **Recupito, G.**, De Martino, V., Di Nucci, D., and Palomba, F. A First Look at the Lifecycle of DL-Specific Self-Admitted Technical Debt. Accepted at the International Workshop of Software Quality Assurance for AI 2025 (SQA4AI) *.

C10 - Della Porta, A., **Recupito, G.**, Lambiase, S., Di Nucci, D., and Palomba, F. (2025). Unlocking Code Simplicity: The Role of Prompt Patterns in Managing LLM Code Complexity. Accepted at the International Workshop of Software Quality Assurance for AI 2025 (SQA4AI).

C11 - **Recupito, G.**, Giordano, G., Di Nucci, D., and Palomba, F. Detecting Semantic Data Smells with BERT: A Transformer-Based Approach to Data Quality. 1st ECAI Workshop on MLOps (MLOps), 2025.

INTERNATIONAL BOOK CHAPTER

B01 - Codabux, Z., Fard, F., Verdecchia, R., Palomba, F., Di Nucci, D., and **Recupito, G.** (2024). Teaching Mining Software Repositories. In Handbook on Teaching Empirical Software Engineering (pp. 325-362). Cham: Springer Nature Switzerland.

BIBLIOGRAPHY

[1] Ahsan Ahmad, Aftab Tariq, Hafiz Khawar Hussain, and Ahmad Yousaf Gill. “Revolutionizing Healthcare: How Deep Learning is poised to Change the Landscape of Medical Diagnosis and Treatment.” In: *Journal of Computer Networks, Architecture and High Performance Computing* (2023).

[2] Nilay Yorgancılar Akgül, Tuğba Taşkaya Temizel, Özden Özcan Top, and Pelin Dayan Akman. “Aligning Data Debt with AI-Integrated Software Project Lifecycle Processes: A Standard-Based Mapping Approach.” In: *2025 IEEE/ACM International Conference on Technical Debt (TechDebt)*. IEEE. 2025, pp. 1–11.

[3] Pekka Ala-Pietilä, Yann Bonnet, Urs Bergmann, Maria Bielikova, Cecilia Bonefeld-Dahl, Wilhelm Bauer, Loubna Bouarfa, Raja Chatila, Mark Coeckelbergh, Virginia Dignum, et al. *The assessment list for trustworthy artificial intelligence (ALTAI)*. European Commission, 2020.

[4] Mohannad Alahdab and Gül Çalıklı. “Empirical analysis of hidden technical debt patterns in machine learning software.” In: *Product-Focused Software Process Improvement: 20th International Conference, PROFES 2019, Barcelona, Spain, November 27–29, 2019, Proceedings 20*. Springer. 2019, pp. 195–202.

[5] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. “Software engineering for machine learning: A case study.” In: *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE. 2019, pp. 291–300.

[6] Areti Ampatzoglou, Apostolos Ampatzoglou, Alexander Chatzigeorgiou, and Paris Avgeriou. "The financial aspect of managing technical debt: A systematic literature review." In: *Information and Software Technology* 64 (2015), pp. 52–73. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2015.04.001. URL: https://www.sciencedirect.com/science/article/pii/S0950584915000762.

[7] Josu Diaz-de Arcaya, Ana I. Torre-Bastida, Gorka Zárate, Raúl Miñón, and Aitor Almeida. "A Joint Study of the Challenges, Opportunities, and Roadmap of MLOps and AIOps: A Systematic Survey." In: *ACM Computing Surveys* 56.4 (Apr. 30, 2024), pp. 1–30. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3625289. URL: https://dl.acm.org/doi/10.1145/3625289 (visited on 08/11/2025).

[8] Gebremariam Assres, Guru Bhandari, Andrii Shalaginov, Tor-Morten Gronli, and Gheorghita Ghinea. "State-of-the-Art and Challenges of Engineering ML- Enabled Software Systems in the Deep Learning Era." In: *ACM Computing Surveys* 57.10 (Oct. 31, 2025), pp. 1–35. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3731597. URL: https://dl.acm.org/doi/10.1145/3731597 (visited on 08/11/2025).

[9] Christiane Atzmüller and Peter Steiner. "Experimental vignette designs for factorial surveys." In: *Kolner Zeitschrift fur Soziologie und Sozialpsychologie* 58 (Mar. 2006), pp. 117–146+200.

[10] Muhammad Ilyas Azeem, Fabio Palomba, Lin Shi, and Qing Wang. "Machine learning techniques for code smell detection: A systematic literature review and meta-analysis." In: *Information and Software Technology* 108 (2019), pp. 115–138.

[11] Sebastian Baltes, Florian Angermeir, Chetan Arora, Marvin Muñoz Barón, Chunyang Chen, Lukas Böhme, Fabio Calefato, Neil Ernst, Davide Falessi, Brian Fitzgerald, et al. "Evaluation Guidelines for Empirical Studies in Software Engineering involving LLMs." In: *arXiv preprint arXiv:2508.15503* (2025).

[12] Victor R. Basili, Gianluigi Caldiera, and H Dieter Rombach. "The goal question metric approach." In: *Encyclopedia of software engineering* (1994), pp. 528–532.

[13] Gabriele Bavota and Barbara Russo. "A large-scale empirical study on self-admitted technical debt." In: *Proceedings of the 13th international conference on mining software repositories*. 2016, pp. 315–326.

[14] Benjamin Benni, Mireille Blay-Fornarino, Sébastien Mosser, Frédéric Précisio, and Günther Jungbluth. "When DevOps Meets Meta-Learning: A Portfolio to Rule them all." In: *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. 2019, pp. 605–612. DOI: 10.1109/MODELS-C.2019.00092.

[15] Narjes Bessghaier, Ali Ouni, and Mohamed Wiem Mkaouer. "On the Diffusion and Impact of Code Smells in Web Applications." In: *Services Computing – SCC 2020*. Ed. by Qingyang Wang, Yunni Xia, Sangeetha Seshadri, and Liang-Jie Zhang. Cham: Springer International Publishing, 2020, pp. 67–84. ISBN: 978-3-030-59592-0.

[16] Aaditya Bhatia, Foutse Khomh, Bram Adams, and Ahmed E Hassan. "An Empirical Study of Self-Admitted Technical Debt in Machine Learning Software." In: *arXiv preprint arXiv:2311.12019* (2023).

[17] Sumon Biswas, Mohammad Wardat, and Hridesh Rajan. "The Art and Practice of Data Science Pipelines: A Comprehensive Study of Data Science Pipelines in Theory, in-the-Small, and in-the-Large." In: *Proceedings of the 44th International Conference on Software Engineering*. Pittsburgh Pennsylvania: ACM, May 2022, pp. 2091–2103. ISBN: 978-1-4503-9221-1. DOI: 10.1145/3510003.3510057. (Visited on 08/11/2025).

[18] Justus Bogner, Roberto Verdecchia, and Ilias Gerostathopoulos. "Characterizing technical debt and antipatterns in AI-based systems: A systematic mapping study." In: *2021 IEEE/ACM Interna-*

tional Conference on Technical Debt (TechDebt). IEEE. 2021, pp. 64–73.

[19] Jan Bosch, Helena Holmström Olsson, Björn Brinne, and Ivica Crnkovic. "AI Engineering: Realizing the Potential of AI." In: *IEEE Software* 39.6 (2022), pp. 23–27.

[20] Jan Bosch, Helena Holmström Olsson, and Ivica Crnkovic. "Engineering ai systems: A research agenda." In: *Artificial Intelligence Paradigms for Smart Cyber-Physical Systems* (2021), pp. 1–19.

[21] Eric Breck, Shanqing Cai, Eric Nielsen, Michael Salib, and D Sculley. "The ML test score: A rubric for ML production readiness and technical debt reduction." In: *2017 IEEE International Conference on Big Data (Big Data)*. IEEE. 2017, pp. 1123–1132.

[22] Eric Breck, Neoklis Polyzotis, Sudip Roy, Steven Whang, and Martin Zinkevich. "Data Validation for Machine Learning." In: *MLSys*. 2019.

[23] Lionel Briand, Khaled El Emam, and Sandro Morasca. "On the application of measurement theory in software engineering." In: *Empirical Software Engineering* 1 (1996), pp. 61–88.

[24] Nicolás Cardozo, Ivana Dusparic, and Christian Cabrera. *Prevalence of Code Smells in Reinforcement Learning Projects*. 2023. arXiv: 2303.10236 [cs.SE].

[25] Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. "Detecting Performance Anti-Patterns for Applications Developed Using Object-Relational Mapping." In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: Association for Computing Machinery, 2014, 1001–1012. ISBN: 9781450327565. DOI: 10.1145/2568225.2568259. URL: https://doi.org/10.1145/2568225.2568259.

[26] Zhifei Chen, Lin Chen, Wanwangying Ma, and Baowen Xu. "Detecting code smells in python programs." In: *2016 international conference on Software Analysis, Testing and Evolution (SATE)*. IEEE. 2016, pp. 18–23.

[27] Pranav Singh Chib and Pravendra Singh. "Recent advancements in end-to-end autonomous driving using deep learning: A survey." In: *IEEE Transactions on Intelligent Vehicles* (2023).

[28] Shaiful Chowdhury, Gias Uddin, Hadi Hemmati, and Reid Holmes. "Method-level bug prediction: Problems and promises." In: *ACM Transactions on Software Engineering and Methodology* 33.4 (2024), pp. 1–31.

[29] Konrad Cinkusz and Jaroslaw A. Chudziak. "Towards LLM-augmented Multiagent Systems for Agile Software Engineering." In: *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. ASE '24. New York, NY, USA: Association for Computing Machinery, Oct. 27, 2024, pp. 2476–2477. ISBN: 979-8-4007-1248-7. DOI: 10.1145/3691620.3695336. URL: https://dl.acm.org/doi/10.1145/3691620.3695336 (visited on 08/12/2025).

[30] Norman Cliff. "Dominance statistics: Ordinal analyses to answer ordinal questions." In: *Psychological Bulletin* 114 (1993), pp. 494–509. URL: https://api.semanticscholar.org/CorpusID:120113824.

[31] William Jay Conover. *Practical nonparametric statistics*. Vol. 350. john wiley & sons, 1999.

[32] Dolors Costal, Cristina Gómez, and Silverio Martínez-Fernández. "Metrics for Code Smells of ML Pipelines." In: *Product-Focused Software Process Improvement*. Ed. by Regine Kadgien, Andreas Jedlitschka, Andrea Janes, Valentina Lenarduzzi, and Xiaozhou Li. Cham: Springer Nature Switzerland, 2024, pp. 3–9. ISBN: 978-3-031-49269-3.

[33] Pierre-Olivier Côté, Amin Nikanjam, Rached Bouchoucha, Ilan Basta, Mouna Abidi, and Foutse Khomh. "Quality issues in machine learning software systems." In: *Empirical Software Engineering* 29.6 (2024), p. 149.

[34] Ward Cunningham. "The WyCash portfolio management system." In: *ACM Sigplan Oops Messenger* 4.2 (1992), pp. 29–30.

[35] Manuel De Stefano, Fabiano Pecorelli, Fabio Palomba, and Andrea De Lucia. "Comparing within-and cross-project machine learning algorithms for code smell detection." In: *Proceedings of the 5th international workshop on machine learning techniques for software quality evolution*. 2021, pp. 1–6.

[36] Dario Di Nucci, Fabio Palomba, Damian A Tamburri, Alexander Serebrenik, and Andrea De Lucia. "Detecting code smells using machine learning techniques: Are we there yet?" In: *2018 ieee 25th international conference on software analysis, evolution and reengineering (saner)*. IEEE. 2018, pp. 612–621.

[37] Yilun Du, Shuang Li, Antonio Torralba, Joshua B. Tenenbaum, and Igor Mordatch. *Improving Factuality and Reasoning in Language Models through Multiagent Debate*. 2023. DOI: 10.48550/arXiv.2305.14325.

[38] Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. "Selecting empirical methods for software engineering research." In: *Guide to advanced empirical software engineering* (2008), pp. 285–311.

[39] Lisa Ehrlinger, Thomas Grubinger, Bence Varga, Mario Pichler, Thomas Natschläger, and Jürgen Zeindl. "Treating missing data in industrial data analytics." In: *2018 Thirteenth International Conference on Digital Information Management (ICDIM)*. IEEE. 2018, pp. 148–155.

[40] Lisa Ehrlinger and Wolfram Wöß. "A survey of data quality measurement and monitoring tools." In: *Frontiers in big data* 5 (2022), p. 850611.

[41] Widad Elouataoui, Imane El Alaoui, Saida El Mendili, and Youssef Gahi. "An Advanced Big Data Quality Framework Based on Weighted Metrics." In: *Big Data and Cognitive Computing* 6.4 (2022), p. 153.

[42] Sergio España, Nelly Condori-Fernandez, Arturo González, and Óscar Pastor. "An empirical comparative evaluation of requirements engineering methods." In: *Journal of the Brazilian Computer Society* 16 (2010), pp. 3–19.

[43] *Example of Data-related SATD introduced during bug-fixing in huggingface-diffusers project.* https://github.com/huggingface/diffusers/blob/3dc97bd1482fb099aa41a15aae55ff45c8f2b042/examples/community/pipeline_zero1to3.py#L667. Accessed: 2024-10-30.

[44] *Example of Model-related SATD introduced during the implementation of a new feature in microsoft-LMOps project.* https://github.com/microsoft/LMOps/blob/00b5846b6e9c20b088c39fa50412fac46f7bd056/structured_prompting/fairseq-version/fairseq/fairseq/tasks/online_backtranslation.py#L641. Accessed: 2024-10-30.

[45] *Example of Training-related SATD introduced during the implementation of a new feature in huggingface-diffusers project.* https://github.com/huggingface/diffusers/blob/ba352aea29df9a7f086bf0815fe9fe479218f801/tests/pipelines/altdiffusion/test_alt_diffusion.py#L187. Accessed: 2024-10-30.

[46] Carmine Ferrara, Francesco Casillo, Carmine Gravino, Andrea De Lucia, and Fabio Palomba. "ReFAIR: Toward a Context-Aware Recommender for Fairness Requirements Engineering." In: *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. 2024, pp. 2631–2642. DOI: 10.1145/3597503.3639185.

[47] Maya Fisher. *Companies Using Machine Learning: Transforming Industries with AI Solutions — modelsai.org.* https://modelsa

i.org/blog/companies-using-machine-learning. [Accessed 10-11-2024].

[48] Harald Foidl and Michael Felderer. "Risk-based data validation in machine learning-based software systems." In: *proceedings of the 3rd ACM SIGSOFT international workshop on machine learning techniques for software quality evaluation*. 2019, pp. 13–18.

[49] Harald Foidl, Michael Felderer, and Stefan Biffl. "Technical Debt in Data-Intensive Software Systems." In: *2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 2019, pp. 338–341. DOI: 10.1109/SEAA.2019.00058.

[50] Harald Foidl, Michael Felderer, and Rudolf Ramler. "Data smells: categories, causes and consequences, and detection of suspicious data in AI-based systems." In: *Proceedings of the 1st International Conference on AI Engineering: Software Engineering for AI*. 2022, pp. 229–239.

[51] Francesca Arcelli Fontana, Marco Zanoni, Alessandro Marino, and Mika V Mäntylä. "Code smell detection: Towards a machine learning-based approach." In: *2013 IEEE international conference on software maintenance*. IEEE. 2013, pp. 396–399.

[52] Martin Fowler and Kent Beck. "Refactoring: Improving the design of existing code." In: *11th European Conference. Jyväskylä, Finland*. 1997.

[53] Gartner. *Gartner Data & Analytics Summit London 2024 – Day 1 Highlights*. Accessed: 2025-10-03. Gartner. May 2024. URL: https://www.gartner.com/en/newsroom/press-releases/2024-05-13-gartner-data-and-analytics-summit-london-2024-day-1-highlights.

[54] Jiri Gesi, Siqi Liu, Jiawei Li, Iftekhar Ahmed, Nachiappan Nagappan, David Lo, Eduardo Santana de Almeida, Pavneet Singh Kochhar, and Lingfeng Bao. *Code Smells in Machine Learning Systems*. 2022. arXiv: 2203.00803 [cs.SE].

[55] Bahar Gezici and Ayça Kolukısa Tarhan. "Systematic literature review on software quality for AI-based software." In: *Empirical Software Engineering* 27.3 (2022), p. 66.

[56] Giammaria Giordano, Giusy Annunziata, Andrea De Lucia, Fabio Palomba, et al. "Understanding Developer Practices and Code Smells Diffusion in AI-Enabled Software: A Preliminary Study." In: *IWSM-Mensura*. 2023.

[57] Giammaria Giordano, Antonio Fasulo, Gemma Catolino, Fabio Palomba, Filomena Ferrucci, and Carmine Gravino. "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)*. IEEE. 2022, pp. 947–958.

[58] Giammaria Giordano, Giulia Sellitto, Aurelio Sepe, Fabio Palomba, and Filomena Ferrucci. "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)*. IEEE. 2023, pp. 227–234.

[59] *GitHub REST API documentation - GitHub Docs — docs.github.com*. https://docs.github.com/en/rest. 2022.

[60] Valentina Golendukhina, Harald Foidl, Michael Felderer, and Rudolf Ramler. "Preliminary findings on the occurrence and causes of data smells in a real-world business travel data processing pipeline." In: cited By 0. 2022, pp. 18–21. DOI: 10.1145/3549037.3561275. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85143197103&doi=10.1145\%2f3549037.3561275&partnerID=40&md5=a50135f57b790372a91ec909e7f4aae9.

[61] Valentina Golendukhina, Valentina Lenarduzzi, and Michael Felderer. "What is software quality for AI engineers? Towards a thinning of the fog." In: *Proceedings of the 1st International Conference on AI Engineering: Software Engineering for AI*. 2022, pp. 1–9.

[62] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D. Sculley. "Google Vizier: A Service for Black-Box Optimization." In: *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. New York, NY, USA: Association for Computing Machinery, 2017. ISBN: 9781450348874.

[63] Lina Gong, Shujuan Jiang, and Li Jiang. "Tackling Class Imbalance Problem in Software Defect Prediction Through Cluster-Based Over-Sampling With Filtering." In: *IEEE Access* 7 (2019), pp. 145725–145737. DOI: 10.1109/ACCESS.2019.2945858.

[64] Ian Goodfellow. *Deep learning*. 2016.

[65] Ulrike M Graetsch, Hourieh Khalajzadeh, Mojtaba Shahin, Rashina Hoda, and John Grundy. "Dealing with data challenges when delivering data-intensive software solutions." In: *IEEE Transactions on Software Engineering* (2023).

[66] Stefan Grafberger, Shubha Guha, Julia Stoyanovich, and Sebastian Schelter. "MLINSPECT: A Data Distribution Debugger for Machine Learning Pipelines." In: *Proceedings of the 2021 International Conference on Management of Data*. SIGMOD/PODS '21: International Conference on Management of Data. Virtual Event China: ACM, June 9, 2021, pp. 2736–2739. ISBN: 978-1-4503-8343-1. DOI: 10.1145/3448016.3452759. URL: https://dl.acm.org/doi/10.1145/3448016.3452759 (visited on 08/11/2025).

[67] David Gray, David Bowes, Neil Davey, Yi Sun, and Bruce Christianson. "The misuse of the NASA metrics data program data sets for automated software defect prediction." In: *15th Annual Conference on Evaluation and Assessment in Software Engineering (EASE 2011)*. 2011, pp. 96–103. DOI: 10.1049/ic.2011.0012.

[68] Hannes Hapke and Catherine Nelson. *Building Machine Learning Pipelines*. "O'Reilly Media, Inc.", July 13, 2020. 367 pp. ISBN: 978-1-4920-5316-3. Google Books: H6_wDwAAQBAJ.

[69] Junda He, Christoph Treude, and David Lo. "LLM-Based Multi-Agent Systems for Software Engineering: Literature Review, Vision, and the Road Ahead." In: *ACM Transactions on Software Engineering and Methodology* 34.5 (June 2025), pp. 1–30. ISSN: 1049-331X, 1557-7392. DOI: 10.1145/3712003. (Visited on 08/12/2025).

[70] Martin Hirzel and Michael Feffer. "A Suite of Fairness Datasets for Tabular Classification." In: *arXiv preprint arXiv:2308.00133* (2023).

[71] Johannes Holvitie, Sherlock A. Licorish, Rodrigo O. Spínola, Sami Hyrynsalmi, Stephen G. MacDonell, Thiago S. Mendes, Jim Buchan, and Ville Leppänen. "Technical debt and agile software development practices and processes: An industry practitioner survey." In: *Information and Software Technology* 96 (2018), pp. 141–160. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2017.11.015. URL: https://www.sciencedirect.com/science/article/pii/S0950584917305098.

[72] Nick Hynes, D Sculley, and Michael Terry. "The data linter: Lightweight, automated sanity checking for ml data sets." In: *NIPS MLSys Workshop*. Vol. 1. 5. 2017.

[73] ISO/IEC. *Software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Quality model for AI systems*. International Standard ISO/IEC 25059:2023. International Organization for Standardization and International Electrotechnical Commission, 2023. URL: https://www.iso.org/obp/ui#iso:std:iso-iec:25059.

[74] ISO/IEC. *Software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Quality model*. International Standard ISO/IEC 25010:2011. International Organization for Standardization and International Electrotechnical Commission, 2023. URL: https://www.iso.org/obp/ui#iso:std:iso-iec:25010.

[75] Emanuele Iannone, Roberta Guadagni, Filomena Ferrucci, Andrea De Lucia, and Fabio Palomba. "The secret life of software vulner-

abilities: A large-scale empirical study." In: *IEEE Transactions on Software Engineering* 49.1 (2022), pp. 44–63.

[76] Fehmi Jaafar, Yann-Gaël Guéhéneuc, Sylvie Hamel, and Foutse Khomh. "Mining the relationship between anti-patterns dependencies and fault-proneness." In: *2013 20th Working Conference on Reverse Engineering (WCRE)*. 2013, pp. 351–360. DOI: 10.1109/WCRE.2013.6671310.

[77] Hadhemi Jebnoun, Houssem Ben Braiek, Mohammad Masudur Rahman, and Foutse Khomh. "The scent of deep learning code: An empirical study." In: *Proceedings of the 17th International Conference on Mining Software Repositories*. 2020, pp. 420–430.

[78] Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. "An exploratory study of the impact of antipatterns on class change-and fault-proneness." In: *Empirical Software Engineering* 17 (2012), pp. 243–275.

[79] Barbara A Kitchenham and Shari Lawrence Pfleeger. "Principles of survey research part 2: designing a survey." In: *ACM SIGSOFT Software Engineering Notes* 27.1 (2002), pp. 18–20.

[80] Barbara Kitchenham. "Procedures for performing systematic reviews." In: *Keele, UK, Keele University* 33.2004 (2004), pp. 1–26.

[81] Barbara Kitchenham and Shari Pfleeger. "Principles of survey research part 3: constructing a survey instrument." In: *ACM SIGSOFT Software Engineering Notes* 27 (Jan. 2002), pp. 20–24.

[82] Dominik Kreuzberger, Niklas Kühl, and Sebastian Hirschl. "Machine Learning Operations (Mlops): Overview, Definition, and Architecture." In: *IEEE Access* 11 (2023), pp. 31866–31879. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2023.3262138. (Visited on 08/11/2025).

[83] Klaus Krippendorff. *Content analysis: An introduction to its methodology*. Sage publications, 2018.

[84] Hiroshi Kuwajima, Hirotoshi Yasuoka, and Toshihiro Nakae. "Engineering problems in machine learning systems." In: *Machine Learning* 109.5 (2020), pp. 1103–1126.

[85] Tai Le Quy, Arjun Roy, Vasileios Iosifidis, Wenbin Zhang, and Eirini Ntoutsi. "A survey on datasets for fairness-aware machine learning." In: *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 12.3 (2022), e1452.

[86] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning." In: *nature* 521.7553 (2015), pp. 436–444.

[87] Meir M Lehman, Juan F Ramil, Paul D Wernick, Dewayne E Perry, and Wladyslaw M Turski. "Metrics and laws of software evolution-the nineties view." In: *Proceedings Fourth International Software Metrics Symposium*. IEEE. 1997, pp. 20–32.

[88] Valentina Lenarduzzi, Francesco Lomio, Sergio Moreschini, Davide Taibi, and Damian Andrew Tamburri. "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. 2021, pp. 43–53.

[89] Yikun Li, Mohamed Soliman, and Paris Avgeriou. "Automatic identification of self-admitted technical debt from four different sources." In: *Empirical Software Engineering* 28.3 (2023), p. 65.

[90] Zengyang Li, Paris Avgeriou, and Peng Liang. "A systematic mapping study on technical debt and its management." In: *Journal of Systems and Software* 101 (2015), pp. 193–220. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2014.12.027. URL: https://www.sciencedirect.com/science/article/pii/S0164121214002854.

[91] Jiakun Liu, Qiao Huang, Xin Xia, Emad Shihab, David Lo, and Shanping Li. "Is using deep learning frameworks free? characterizing technical debt in deep learning frameworks." In: *Proceedings of the*

ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Society. 2020, pp. 1–10.

[92] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. "Lost in the Middle: How Language Models Use Long Contexts." In: *Transactions of the Association for Computational Linguistics* 12 (Feb. 2024), pp. 157–173. ISSN: 2307-387X. DOI: 10.1162/tacl_a_00638. eprint: https://direct.mit.edu/tacl/article-pdf/doi/10.1162/tacl_a_00638/2336043/tacl_a_00638.pdf. URL: https://doi.org/10.1162/tacl_a_00638.

[93] Yue Liu, Sin Kit Lo, Qinghua Lu, Liming Zhu, Dehai Zhao, Xiwei Xu, Stefan Harrer, and Jon Whittle. "Agent Design Pattern Catalogue: A Collection of Architectural Patterns for Foundation Model Based Agents." In: *Journal of Systems and Software* 220 (Feb. 1, 2025), p. 112278. ISSN: 0164-1212. DOI: 10.1016/j.jss.2024.112278. URL: https://www.sciencedirect.com/science/article/pii/S0164121224003224 (visited on 08/12/2025).

[94] Zijun Liu, Yanzhe Zhang, Peng Li, Yang Liu, and Diyi Yang. *Dynamic LLM-Agent Network: An LLM-agent Collaboration Framework with Agent Team Optimization*. 2023. DOI: 10.48550/arXiv.2310.02170{\#}.

[95] Lucy Ellen Lwakatare, Aiswarya Raj, Jan Bosch, Helena Holmström Olsson, and Ivica Crnkovic. "A taxonomy of software engineering challenges for machine learning systems: An empirical investigation." In: *Agile Processes in Software Engineering and Extreme Programming: 20th International Conference, XP 2019, Montréal, QC, Canada, May 21–25, 2019, Proceedings 20*. Springer International Publishing. 2019, pp. 227–243.

[96] Abdou Maiga, Nasir Ali, Neelesh Bhattacharya, Aminata Sabané, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Esma Aïmeur. "Support Vector Machines for Anti-Pattern Detection." In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Soft-*

ware Engineering. ASE '12. Essen, Germany: Association for Computing Machinery, 2012, 278–281. ISBN: 9781450312042. DOI: 10.1145/2351676.2351723. URL: https://doi.org/10.1145/2351676.2351723.

[97] Taha Mansouri, Mohammad Reza Sadeghi Moghadam, Fatemeh Monshizadeh, and Ahad Zareravasan. "IoT data quality issues and potential solutions: a literature review." In: *The Computer Journal* 66.3 (2023), pp. 615–625.

[98] R. Marinescu. "Detection strategies: metrics-based rules for detecting design flaws." In: *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.* 2004, pp. 350–359. DOI: 10.1109/ICSM.2004.1357820.

[99] Silverio Martínez-Fernández, Justus Bogner, Xavier Franch, Marc Oriol, Julien Siebert, Adam Trendowicz, Anna Maria Vollmer, and Stefan Wagner. "Software engineering for AI-based systems: a survey." In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31.2 (2022), pp. 1–59.

[100] Antonio Mastropaolo, Massimiliano Di Penta, and Gabriele Bavota. "Towards Automatically Addressing Self-Admitted Technical Debt: How Far Are We?" In: *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2023, pp. 585–597.

[101] Jeanna Matthews. "Patterns and anti-patterns, principles and pitfalls: accountability and transparency in AI." In: *AI Magazine* 41.1 (2020), pp. 82–89.

[102] Sean McGuire, Erin Schultz, Bimpe Ayoola, and Paul Ralph. "Sustainability is stratified: Toward a better theory of sustainable software engineering." In: *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE. 2023, pp. 1996–2008.

[103] Tim Menzies. "The Five Laws of SE for AI." In: *IEEE Software* 37.1 (2020), pp. 81–85. DOI: 10.1109/MS.2019.2954841.

[104] Tommi Mikkonen, Jukka K Nurminen, Mikko Raatikainen, Ilenia Fronza, Niko Mäkitalo, and Tomi Männistö. "Is machine learning software just software: A maintainability view." In: *Software Quality: Future Perspectives on Software Engineering Quality: 13th International Conference, SWQD 2021, Vienna, Austria, January 19–21, 2021, Proceedings 13*. Springer. 2021, pp. 94–105.

[105] Claire Cain Miller. "Can an algorithm hire better than a human." In: *The New York Times* 25 (2015).

[106] Moran Mizrahi, Guy Kaplan, Dan Malkin, Rotem Dror, Dafna Shahaf, and Gabriel Stanovsky. "State of What Art? A Call for Multi-Prompt LLM Evaluation." In: *Transactions of the Association for Computational Linguistics* 12 (2024), pp. 933–949. DOI: 10.1162/tacl{\textunderscore}a{\textunderscore}00681.

[107] Naouel Moha, Yann-Gael Gueheneuc, Laurence Duchien, and Anne-Francoise Le Meur. "DECOR: A Method for the Specification and Detection of Code and Design Smells." In: *IEEE Transactions on Software Engineering* 36.1 (2010), pp. 20–36. DOI: 10.1109/TSE.2009.50.

[108] Sergio Moreschini, Elvira Arvanitou, Elisavet-Persefoni Kanidou, Nikolaos Nikolaidis, Ruoyu Su, Apostolos Ampatzoglou, Alexander Chatzigeorgiou, and Valentina Lenarduzzi. "The Evolution of Technical Debt from DevOps to Generative AI: A multivocal literature review." In: *Journal of Systems and Software* (2025), p. 112599.

[109] Aiswarya Munappy, Jan Bosch, Helena Holmström Olsson, Anders Arpteg, and Björn Brinne. "Data management challenges for deep learning." In: *2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE. 2019, pp. 140–147.

[110] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. "Using an LLM to Help With Code Understanding." In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ICSE '24: IEEE/ACM 46th In-

ternational Conference on Software Engineering. Lisbon Portugal: ACM, Apr. 12, 2024, pp. 1–13. ISBN: 979-8-4007-0217-4. DOI: 10.1145/3597503.3639187. URL: https://dl.acm.org/doi/10.1145/3597503.3639187 (visited on 08/11/2025).

[111] Roger Nazir, Alessio Bucaioni, and Patrizio Pelliccione. "Architecting ML-enabled systems: Challenges, best practices, and design decisions." In: *Journal of Systems and Software* 207 (2024), p. 111860.

[112] Tomoko Nemoto and David Beglar. "Likert-scale questionnaires." In: *JALT 2013 conference proceedings*. 2014, pp. 1–8.

[113] Morteza Noferesti and Naser Ezzati-Jivan. "Enhancing empirical software performance engineering research with kernel-level events: A comprehensive system tracing approach." In: *Journal of Systems and Software* 216 (2024), p. 112117.

[114] Nicole Novielli, Daniela Girardi, and Filippo Lanubile. "A Benchmark Study on Sentiment Analysis for Software Engineering Research." In: MSR '18. Gothenburg, Sweden: Association for Computing Machinery, 2018. ISBN: 9781450357166. DOI: 10.1145/3196398.3196403. URL: https://doi.org/10.1145/3196398.3196403.

[115] David OBrien, Sumon Biswas, Sayem Imtiaz, Rabe Abdalkareem, Emad Shihab, and Hridesh Rajan. "23 shades of self-admitted technical debt: An empirical study on machine learning software." In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2022, pp. 734–746.

[116] Bentley James Oakes, Michalis Famelis, and Houari Sahraoui. "Building Domain-Specific Machine Learning Workflows: A Conceptual Framework for the State of the Practice." In: *ACM Transactions on Software Engineering and Methodology* 33.4 (May 31, 2024), pp. 1–50. ISSN: 1049-331X, 1557-7392. DOI: 10.1145/3638243. URL: https://dl.acm.org/doi/10.1145/3638243 (visited on 08/11/2025).

[117] Steffen Olbrich, Daniela S. Cruzes, Victor Basili, and Nico Zazworka. "The evolution and impact of code smells: A case study of two open source systems." In: *The evolution and impact of code smells: A case study of two open source systems*. Cited by: 179; All Open Access, Green Open Access. 2009, 390 – 400. DOI: 10.1109/ESEM.2009.5314231. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-72449192177&doi=10.1109%2fESEM.2009.5314231&partnerID=40&md5=43ebe3211c8a106a5877364d941bf97e.

[118] Parmy Olson. "The algorithm that beats your bank manager." In: *CNN Money March* 15 (2011).

[119] Ipek Ozkaya. "What is really different in engineering AI-enabled systems?" In: *IEEE software* 37.4 (2020), pp. 3–6.

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

[121] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrea De Lucia. "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. 2014, pp. 101–110.

[122] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. "On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation." In: *Empirical Software Engineering* 23.3 (June 2018), pp. 1188–1221. ISSN: 1573-7616. DOI: 10.1007/s10664-017-9535-z. URL: https://doi.org/10.1007/s10664-017-9535-z.

[123] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. "Mining Version Histories for Detecting Code Smells." In: *IEEE Transactions on Soft-*

ware Engineering 41.5 (2015), pp. 462–489. DOI: 10.1109/TSE.2014.2372760.

[124] Fabio Palomba, Annibale Panichella, Andrea De Lucia, Rocco Oliveto, and Andy Zaidman. "A textual-based technique for Smell Detection." In: *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. 2016, pp. 1–10. DOI: 10.1109/ICPC.2016.7503704.

[125] Fabio Palomba, Annibale Panichella, Andy Zaidman, Rocco Oliveto, and Andrea De Lucia. "The Scent of a Smell: An Extensive Comparison Between Textual and Structural Smells." In: *IEEE Transactions on Software Engineering* 44.10 (2018), pp. 977–1000. DOI: 10.1109/TSE.2017.2752171.

[126] Zhenyu Pan, Xuefeng Song, Yunkun Wang, Rongyu Cao, Binhua Li, Yongbin Li, and Han Liu. "Do Code LLMs Understand Design Patterns?" In: *2025 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code)*. 2025 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code). May 2025, pp. 209–212. DOI: 10.1109/LLM4Code66737.2025.00031. URL: https://ieeexplore.ieee.org/document/11028452/ (visited on 08/11/2025).

[127] Elder Vicente de Paulo Sobrinho, Andrea De Lucia, and Marcelo de Almeida Maia. "A systematic literature review on bad smells–5 w's: which, when, what, who, where." In: *IEEE Transactions on Software Engineering* 47.1 (2018), pp. 17–66.

[128] Fabiano Pecorelli, Fabio Palomba, Dario Di Nucci, and Andrea De Lucia. "Comparing heuristic and machine learning approaches for metric-based code smell detection." In: *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE. 2019, pp. 93–104.

[129] Federica Pepe, Fiorella Zampetti, Antonio Mastropaolo, Gabriele Bavota, and Massimiliano Di Penta. "A Taxonomy of Self-Admitted Technical Debt in Deep Learning Systems." In: *IEEE Interna-*

tional Conference on Software Maintenance and Evolution (ICSME) (2024).

[130] Catherine Petrozzino. "Who pays for ethical debt in AI?" In: *AI and Ethics* 1.3 (2021), pp. 205–208.

[131] Xavier Pleimling, Vedant Shah, and Ismini Lourentzou. "[Data] Quality Lies In The Eyes Of The Beholder." In: *Proceedings of the 15th International Conference on PErvasive Technologies Related to Assistive Environments*. 2022, pp. 118–124.

[132] Reinhold Plösch, Johannes Bräuer, Matthias Saft, and Christian Körner. "Design debt prioritization: A design best practice-based approach." In: *Proceedings of the 2018 International Conference on Technical Debt*. 2018, pp. 95–104.

[133] Neoklis Polyzotis, Sudip Roy, Steven Euijong Whang, and Martin Zinkevich. "Data Management Challenges in Production Machine Learning." In: *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD '17. Chicago, Illinois, USA: Association for Computing Machinery, 2017, 1723–1726. ISBN: 9781450341974.

[134] Aniket Potdar and Emad Shihab. "An exploratory study on self-admitted technical debt." In: *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE. 2014, pp. 91–100.

[135] Mikko Raatikainen, Charalampos Souris, Jukka Remes, and Vlad Stirbu. "Machine Learning Lineage for Trustworthy Machine Learning Systems: Information Framework for MLOps Pipelines." In: *IEEE Software* 42.1 (Jan. 2025), pp. 51–58. ISSN: 1937-4194. DOI: 10.1109/MS.2024.3414317. (Visited on 08/11/2025).

[136] Emmanuel Raj. *Engineering MLOps: Rapidly build, test, and manage production-ready machine learning life cycles at scale*. Packt Publishing Ltd, 2021.

[137] Paul Ralph, Nauman bin Ali, Sebastian Baltes, Domenico Bianculli, Jessica Diaz, Yvonne Dittrich, Neil Ernst, Michael Felderer, Robert Feldt, Antonio Filieri, et al. "Empirical standards for software engineering research." In: *arXiv preprint arXiv:2010.03525* (2020).

[138] Jacek Ratzinger, Thomas Sigmund, and Harald C Gall. "On the relation of refactorings and software defect prediction." In: *Proceedings of the 2008 international working conference on Mining software repositories*. 2008, pp. 35–38.

[139] Daniele Ravì, Charence Wong, Fani Deligianni, Melissa Berthelot, Javier Andreu-Perez, Benny Lo, and Guang-Zhong Yang. "Deep learning for health informatics." In: *IEEE journal of biomedical and health informatics* 21.1 (2016), pp. 4–21.

[140] Jörg Rech and Klaus-Dieter Althoff. "Artificial intelligence and software engineering: Status and future trends." In: *KI* 18.3 (2004), pp. 5–11.

[141] Gilberto Recupito, Vincenzo De Martino, Dario Di Nucci, and Fabio Palomba. "A First Look at the Lifecycle of DL-Specific Self-Admitted Technical Debt." In: *Accepted at 2025 IEEE International Conference on Software Analysis, Evolution and Reengineering - Companion (SANER-C)* ().

[142] Gilberto Recupito, Vincenzo De Martino, Dario Di Nucci, and Fabio Palomba. "A First Look at the Lifecycle of DL-Specific Self-Admitted Technical Debt - Replication package." In: (Feb. 2024). URL: https://figshare.com/s/5d50b381dfd29719bab0.

[143] Gilberto Recupito, Giammaria Giordano, Filomena Ferrucci, Dario Di Nucci, and Fabio Palomba. *When Code Smells Meet ML: On the Lifecycle of ML-specific Code Smells in ML-enabled Systems - Appendix*. Feb. 2024. DOI: 10.6084/m9.figshare.28167065. URL: https://doi.org/10.6084/m9.figshare.28167065.

[144] Gilberto Recupito, Giammaria Giordano, Filomena Ferrucci, Dario Di Nucci, and Fabio Palomba. "When code smells meet ML: on the

lifecycle of ML-specific code smells in ML-enabled systems." In: *Empirical Software Engineering* 30.5 (2025), pp. 1–49.

[145] Gilberto Recupito, Dominik Opitz, Tobias Hecking, Dario Di Nucci, Fabio Palomba, and Michael Felderer. "LLM-based Architecture to Create Graph Representations from Machine Learning Pipelines - Replication Package." In: (). URL: https://doi.org/10.6084/m9.figshare.27400710.

[146] Gilberto Recupito, Dominik Opitz, Tobias Hecking, Dario Di Nucci, Fabio Palomba, and Michael Felderer. "LLM-based Architecture to Create Graph Representations from Machine Learning Pipelines." In: *submitted to IEEE Transactions on Software Engineering* (2025).

[147] Gilberto Recupito, Fabiano Pecorelli, Gemma Catolino, Valentina Lenarduzzi, Davide Taibi, Dario Di Nucci, and Fabio Palomba. "Technical debt in AI-enabled systems: On the prevalence, severity, impact, and management strategies for code and architecture." In: *Journal of Systems and Software* 216 (2024), p. 112151.

[148] Gilberto Recupito, Fabiano Pecorelli, Gemma Catolino, Sergio Moreschini, Dario Di Nucci, Fabio Palomba, and Damian A. Tamburri. "A Multivocal Literature Review of MLOps Tools and Features." In: *2022 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 2022, pp. 84–91. DOI: 10.1109/SEAA56994.2022.00021.

[149] Gilberto Recupito, Raimondo Rapacciuolo, Dario Di Nucci, and Fabio Palomba. *Unmasking Data Secrets: An Empirical Investigation into Data Smells and Their Impact on Data Quality. Online Appendix*. 2023. URL: https://figshare.com/s/31a3742977548f96f506.

[150] Gilberto Recupito, Raimondo Rapacciuolo, Dario Di Nucci, and Fabio Palomba. "Unmasking data secrets: An empirical investigation into data smells and their impact on data quality." In: *Proceedings of the IEEE/ACM 3rd International Conference on AI Engineering-Software Engineering for AI*. 2024, pp. 53–63.

[151] Brittany Reid, Markus Wagner, Marcelo d'Amorim, and Christoph Treude. "Software Engineering User Study Recruitment on Prolific: An Experience Report." In: *arXiv preprint arXiv:2201.05348* (2022).

[152] Wasiur Rhmann. "Quantitative Software Change Prediction in Open Source Web Projects Using Time Series Forecasting." In: *International Journal of Open Source Software and Processes (IJOSSP)* 12.2 (2021), pp. 36–51.

[153] Jonan Richards and Mairieli Wessel. "What You Need Is What You Get: Theory of Mind for an LLM-Based Code Understanding Assistant." In: *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2024 IEEE International Conference on Software Maintenance and Evolution (ICSME). Oct. 2024, pp. 666–671. DOI: 10.1109/ICSME58944.2024.00070. URL: https://ieeexplore.ieee.org/document/10795053/ (visited on 08/11/2025).

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

[155] Krishna Ronanki, Beatriz Cabrero-Daniel, and Christian Berger. "Prompt smells: An omen for undesirable generative AI outputs." In: *Proceedings of the IEEE/ACM 3rd International Conference on AI Engineering-Software Engineering for AI*. 2024, pp. 286–287.

[156] B. Rosner and R. J. Glynn. "Power and Sample Size Estimation for the Wilcoxon Rank Sum Test with Application to Comparisons of C Statistics from Alternative Prediction Models." In: *Biometrics* 65.1 (Mar. 2009), pp. 188–197. ISSN: 0006-341X. DOI: 10.1111/j.1541-0420.2008.01062.x. eprint: https://academic.oup.com/biometrics/article-pdf/65/1/188/52732920/biometrics_65_1_188.pdf. URL: https://doi.org/10.1111/j.1541-0420.2008.01062.x.

[157] Riccardo Roveda, Francesca Arcelli Fontana, Ilaria Pigazzini, and Marco Zanoni. "Towards an architectural debt index." In: *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE. 2018, pp. 408–416.

[158] Hossein Saiedian, Prabha Kumarakulasingam, and Muhammad Anan. "Scenario-based requirements analysis techniques for real-time software systems: a comparative evaluation." In: *Requirements Engineering* 10 (2005), pp. 22–33.

[159] Joseph L Schafer and Maren K Olsen. "Multiple imputation for multivariate missing-data problems: A data analyst's perspective." In: *Multivariate behavioral research* 33.4 (1998), pp. 545–571.

[160] David Sculley. *Machine Learning, Technical Debt and You*. 2017. URL: https://www.youtube.com/watch?v=V18AsBIHlWs.

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

[162] Greg Serapio-García, Mustafa Safdari, Clément Crepy, Luning Sun, Stephen Fitz, Peter Romero, Marwa Abdulhai, Aleksandra Faust, and Maja Matarić. *Personality Traits in Large Language Models*. URL: http://arxiv.org/pdf/2307.00184v3.

[163] Alex Serban, Koen Van der Blom, Holger Hoos, and Joost Visser. "Adoption and effects of software engineering best practices in machine learning." In: *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 2020, pp. 1–12.

[164] Karthik Shivashankar and Antonio Martini. "Maintainability Challenges in ML: A Systematic Literature Review." In: *2022 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 2022, pp. 60–67. DOI: 10.1109/SEAA56994.2022.00018.

[165] Arumoy Shome, Luis Cruz, and Arie Van Deursen. "Data smells in public datasets." In: *Proceedings of the 1st International Conference on AI Engineering: Software Engineering for AI*. 2022, pp. 205–216.

[166] Julien Siebert, Lisa Joeckel, Jens Heidrich, Koji Nakamichi, Kyoko Ohashi, Isao Namba, Rieko Yamamoto, and Mikio Aoyama. "Towards Guidelines for Assessing Qualities of Machine Learning Systems." In: *Quality of Information and Communications Technology*. Ed. by Martin Shepperd, Fernando Brito e Abreu, Alberto Rodrigues da Silva, and Ricardo Pérez-Castillo. Cham: Springer International Publishing, 2020, pp. 17–31. ISBN: 978-3-030-58793-2.

[167] Andrew J Simmons, Scott Barnett, Jessica Rivera-Villicana, Akshat Bajaj, and Rajesh Vasa. "A large-scale comparative analysis of coding standard conformance in open-source data science projects." In: *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 2020, pp. 1–11.

[168] Emmanuel Iko-Ojo Simon. "Characterising Algorithm Debt in Machine and Deep Learning Systems." In: *2025 IEEE/ACM 47th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE. 2025, pp. 216–217.

[169] Ronnie de Souza Santos, Felipe Fronchetti, Sávio Freire, and Rodrigo Spinola. "Software fairness debt: Building a research agenda for addressing bias in AI systems." In: *ACM Transactions on Software Engineering and Methodology* 34.5 (2025), pp. 1–21.

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

[171] Monika Steidl, Michael Felderer, and Rudolf Ramler. "The pipeline for the continuous development of artificial intelligence models—Current state of research and practice." In: *Journal of Systems*

and Software 199 (2023), p. 111615. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2023.111615. URL: https://www.sciencedirect.com/science/article/pii/S0164121223000109.

[172] Aviel J. Stein, Daniel Schwartz, Yiwen Shi, and Spiros Mancoridis. " Linguistic Approach to Segmenting Source Code." In: *2022 IEEE 16th International Conference on Semantic Computing (ICSC)*. Los Alamitos, CA, USA: IEEE Computer Society, Jan. 2022, pp. 177–178. DOI: 10.1109/ICSC52841.2022.00037. URL: https://doi.ieeecomputersociety.org/10.1109/ICSC52841.2022.00037.

[173] Margaret-Anne Storey, Rashina Hoda, Alessandra Maciel Paz Milani, and Maria Teresa Baldassarre. "Guiding principles for mixed methods research in software engineering." In: *Empirical Software Engineering* 30.5 (2025), p. 138.

[174] Dina Sukhobok, Nikolay Nikolov, and Dumitru Roman. "Tabular data anomaly patterns." In: *2017 International Conference on Big Data Innovations and Applications (Innovate-Data)*. IEEE. 2017, pp. 25–34.

[175] Xiaobing Sun, Tianchi Zhou, Gengjie Li, Jiajun Hu, Hui Yang, and Bin Li. "An Empirical Study on Real Bugs for Machine Learning Programs." In: *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. 2017, pp. 348–357. DOI: 10.1109/APSEC.2017.41.

[176] Girish Suryanarayana, Ganesh Samarthyam, and Tushar Sharma. *Refactoring for Software Design Smells: Managing Technical Debt*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2014. ISBN: 0128013974.

[177] Georgios Symeonidis, Evangelos Nerantzis, Apostolos Kazakis, and George A. Papakostas. "MLOps - Definitions, Tools and Challenges." In: *2022 IEEE 12th Annual Computing and Communication Workshop and Conference (CCWC)*. 2022 IEEE 12th Annual Computing and Communication Workshop and Conference (CCWC). Jan. 2022, pp. 0453–0460. DOI: 10.1109/CCWC54503.2022.97209

02. URL: https://ieeexplore.ieee.org/document/9720902/ (visited on 08/11/2025).

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

[179] Henri Theil. "A multinomial extension of the linear logit model." In: *International economic review* 10.3 (1969), pp. 251–259.

[180] Ferdian Thung, Shaowei Wang, David Lo, and Lingxiao Jiang. "An Empirical Study of Bugs in Machine Learning Systems." In: *2012 IEEE 23rd International Symposium on Software Reliability Engineering*. 2012, pp. 271–280. DOI: 10.1109/ISSRE.2012.22.

[181] Edith Tom, Aybüke Aurum, and Richard Vidgen. "An exploration of technical debt." In: *Journal of Systems and Software* 86.6 (2013), pp. 1498–1516. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2012.12.052. URL: https://www.sciencedirect.com/science/article/pii/S0164121213000022.

[182] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. "When and Why Your Code Starts to Smell Bad." In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 1. 2015, pp. 403–414. DOI: 10.1109/ICSE.2015.59.

[183] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. "When and why your code starts to smell bad (and whether the smells go away)." In: *IEEE Transactions on Software Engineering* 43.11 (2017), pp. 1063–1088.

[184] Vassilios Tzerpos and Richard C Holt. "MoJo: A distance metric for software clusterings." In: *Sixth working conference on reverse engineering (cat. no. PR00303)*. IEEE. 1999, pp. 187–193.

[185] Sebastian Uchitel, Marsha Chechik, Massimiliano Di Penta, Bram Adams, Nazareno Aguirre, Gabriele Bavota, Domenico Bianculli, Kelly Blincoe, Ana Cavalcanti, Yvonne Dittrich, et al. "Scoping software engineering for AI: The TSE perspective." In: Institute of Electrical and Electronics Engineers. 2024.

[186] Karthik Vaidhyanathan and Henry Muccini. "Software Architecture in the Age of Agentic AI." In: *European Conference on Software Architecture*. Springer. 2025, pp. 41–49.

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

[188] Aimee Van Wynsberghe. "Sustainable AI: AI for sustainability and the sustainability of AI." In: *AI and Ethics* 1.3 (2021), pp. 213–218.

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

[190] Heng Wang and Zubin Abraham. "Concept drift detection for streaming data." In: *2015 International Joint Conference on Neural Networks (IJCNN)*. 2015, pp. 1–9. DOI: 10.1109/IJCNN.2015.7280398.

[191] Xiaofei Wang, Herbert Schuster, Reuben Borrison, and Benjamin Klöpper. "Technical debt management in industrial ml-state of practice and management model proposal." In: *2023 IEEE 21st International Conference on Industrial Informatics (INDIN)*. IEEE. 2023, pp. 1–9.

[192] Hironori Washizaki, Hiromu Uchida, Foutse Khomh, and Yann-Gaël Guéhéneuc. "Studying software engineering patterns for designing machine learning systems." In: *2019 10th International Workshop on Empirical Software Engineering in Practice (IWESEP)*. IEEE. 2019, pp. 49–495.

[193] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. "Chain-of-thought prompting elicits reasoning in large language models." In: *Proceedings of the 36th International Conference on Neural Information Processing Systems*. NIPS '22. New Orleans, LA, USA: Curran Associates Inc., 2024. ISBN: 9781713871088.

[194] Zhihua Wen and Vassilios Tzerpos. "An effectiveness measure for software clustering algorithms." In: *Proceedings. 12th IEEE International Workshop on Program Comprehension, 2004.* IEEE. 2004, pp. 194–203.

[195] Marilyn Domas White and Emily E Marsh. "Content analysis: A flexible methodology." In: *Library trends* 55.1 (2006), pp. 22–45.

[196] Ratnadira Widyasari et al. "NICHE: A Curated Dataset of Engineered Machine Learning Projects in Python." In: *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. 2023, pp. 62–66.

[197] R. Wirth and Jochen Hipp. "CRISP-DM: Towards a standard process model for data mining." In: *Proceedings of the 4th International Conference on the Practical Applications of Knowledge Discovery and Data Mining* (Jan. 2000).

[198] Claes Wohlin. "Guidelines for snowballing in systematic literature studies and a replication in software engineering." In: *Proceedings of the 18th international conference on evaluation and assessment in software engineering*. 2014, pp. 1–10.

[199] Claes Wohlin, Emilia Mendes, Katia Romero Felizardo, and Marcos Kalinowski. "Guidelines for the search strategy to update systematic literature reviews in software engineering." In: *Information and Software Technology* 127 (2020), p. 106366. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2020.106366. URL: https://www.sciencedirect.com/science/article/pii/S0950584919302233X.

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

[201] Lu Xiao, Yuanfang Cai, Rick Kazman, Ran Mo, and Qiong Feng. "Identifying and quantifying architectural debt." In: *Proceedings of the 38th international conference on software engineering*. 2016, pp. 488–498.

[202] Zhenran Xu, Senbao Shi, Baotian Hu, Jindi Yu, Dongfang Li, Min Zhang, and Yuxiang Wu. *Towards Reasoning in Large Language Models via Multi-Agent Peer Review Collaboration*. 2023. DOI: 10.48550/arXiv.2311.08152.

[203] Han Yu, Shanhe Jiang, and Kenneth C. Land. "Multicollinearity in hierarchical linear models." In: *Social Science Research* 53 (2015), pp. 118–136. ISSN: 0049-089X. DOI: https://doi.org/10.1016/j.ssresearch.2015.04.008. URL: https://www.sciencedirect.com/science/article/pii/S0049089X15000885.

[204] Mohammad Zarour, Hamza Alzabut, and Khalid T Al-Sarayreh. "MLOps best practices, challenges and maturity models: A systematic literature review." In: *Information and Software Technology* 183 (2025), p. 107733.

[205] Pengwei Zhan, Zhen Xu, Qian Tan, Jie Song, and Ru Xie. *Unveiling the Lexical Sensitivity of LLMs: Combinatorial Optimization for Prompt Enhancement*. 2024. arXiv: 2405.20701 [cs.CL]. URL: https://arxiv.org/abs/2405.20701.

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

[207] Shuai Zhang, Lina Yao, Aixin Sun, and Yi Tay. "Deep learning based recommender system: A survey and new perspectives." In: *ACM computing surveys (CSUR)* 52.1 (2019), pp. 1–38.

[208] Jianlong Zhou and Fang Chen. *Human and Machine Learning*. Springer, 2018.

[209] Yuming Zhou, Hareton Leung, and Baowen Xu. "Examining the potentially confounding effect of class size on the associations between object-oriented metrics and change-proneness." In: *IEEE Transactions on Software Engineering* 35.5 (2009), pp. 607–623.