The Specification, Detection, and Refactoring of Machine Learning Service Misuses

by

Hadil Ben Amor

MANUSCRIPT-BASED THESIS PRESENTED TO ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
IN PARTIAL FULFILLMENT OF A MASTER'S DEGREE
WITH THESIS
M.A.Sc.

MONTREAL, MARCH 22ND, 2026

ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

BOARD OF EXAMINERS

THIS THESIS HAS BEEN EVALUATED

BY THE FOLLOWING BOARD OF EXAMINERS

Ms. Manel Abdellatif, Thesis supervisor
Department of Software Engineering and IT at École de Technologie Supérieure

Mr. Taher A. Ghaleb, Thesis Co-Supervisor
Department of Computer Science at Trent University

Mr. Julien Gascon-Samson, Chair, Board of Examiners
Department of Software Engineering and IT at École de Technologie Supérieure

Mr. Mohammed Sayyagh, Member of the Jury
Department of Software Engineering and IT at École de Technologie Supérieure

THIS THESIS WAS PRESENTED AND DEFENDED

IN THE PRESENCE OF A BOARD OF EXAMINERS AND THE PUBLIC

ON "JANUARY 30TH, 2026"

AT ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

ACKNOWLEDGEMENTS

I would like to thank my research supervisors, Prof. Manel Abdellatif, and Prof. Taher A. Ghaleb, for their valuable support, and their guidance throughout my research work.

To my parents, Faiza and Bechir, the love of my life, your eldest daughter is about to defend her Master's degree. I still cannot believe it either. I hope I lived up to the responsibility you entrusted to me.

To my siblings, I hope I am the older sister you wished for. I hope I have inspired you, and been someone you could always count on.

To my grandparents who passed away four years ago, I would like to take this moment to honor them and cherish their everlasting presence in my heart.

To my long-distance friends: Bacem, Mouna, Sarah, and Mokhles, you are a blessing. No words can describe how grateful I am to have you.

To the beautiful souls I met here: I was lucky to have the best lab partners, the best neighbors, and the best friends. We tried new foods together, traveled, and went on crazy rides. Even on nights with minus-too-many degrees, nothing stopped us from going out and doing different activities. We shared all the ups and downs. Thank you for making this journey truly unforgettable.

To my part-time job manager, Mehdi, who marked my life and taught me so much about this community, and customer services. As an international student, I could not wish for a better manager. Your trust in me means a lot!

To my inspiration in creating impact, remaining humble, and spreading positivity, the one who connects like-minded people, Oussema Chelly, thank you! To many more successes ahead.

To the love of my life, whom I haven't met yet, another degree without you. You missed a lot of stories and suspense throughout this journey! Just be prepared with a good excuse for keeping me waiting.

I would like to express my gratitude for everything in my life: for the health, the love, and the lessons I have learned. The two turning points of my life in 2018 and 2023 were challenging, yet true blessings, and I could not have made better choices.

Finally, to myself, I am proud of you. Proud of the skills you are developing, and the discipline that lifts you back up every time. The voluntary work, the way you prioritize responsibilities, and the management of time, money, stress, and relationships, all of it matters. Keep your energy high, and continue to shine bright like a diamond!

Spécification, Détection et Refactorisation des Mauvaises Utilisations des Services de ML

Hadil Ben Amor

RÉSUMÉ

Les modèles d'apprentissage automatique sont aujourd'hui largement utilisés dans des domaines tels que le traitement d'images, le diagnostic médical et les systèmes autonomes. Les grands fournisseurs infonuagiques, comme Amazon, Google et Microsoft, proposent des services infonuagiques d'apprentissage automatique qui facilitent le développement en évitant la création de modèles à partir de zéro. Toutefois, ces services sont souvent utilisés de manière inappropriée, ce qui dégrade la qualité des systèmes et leur maintenabilité.

Malgré certains travaux portant sur des mauvaises pratiques dans la programmation orientée objet, les services infonuagiques et les systèmes basés sur l'apprentissage automatique, la littérature ne propose pas encore de cadre global pour la spécification, la détection et la correction des mauvaises pratiques liées aux services infonuagiques d'apprentissage automatique. Ce projet répond à ce manque à travers trois contributions : (1) un catalogue des mauvaises pratiques d'utilisation des services infonuagiques d'apprentissage automatique, (2) une approche automatisée de détection de ces mauvaises pratiques, et (3) une stratégie de leur correction.

Le catalogue a été construit à partir d'une étude empirique multi-focale combinant une revue de la littérature, l'analyse de 377 projets GitHub exploitant des services infonuagiques d'apprentissage automatique et une enquête auprès de 50 praticiens, conduisant à l'identification de 20 mauvaises pratiques.

Nous proposons ensuite MLmisFinder, une approche automatisée fondée sur un méta-modèle et des règles de détection. Évaluée sur 107 projets, elle atteint une précision de 96,7% et un rappel de 97%, tout en démontrant une bonne capacité de passage à l'échelle.

Enfin, la correction automatisée à l'aide des grands modèles de langage (LLMs) a été étudiée, **GPT** étant le plus performant dans 58% des cas, avec une précision allant jusqu'à 82% et un temps d'exécution moyen de 3,86 secondes.

Mots-clés: Mauvaises pratiques, Services d'apprentissage automatique, Spécification, Détection, Correction

The Specification, Detection, and Refactoring of Machine Learning Service Misuses

Hadil Ben Amor

ABSTRACT

Machine Learning (ML) models are widely used across many domains, including image processing, medical diagnostics, and autonomous systems. Major cloud providers such as Amazon, Google, and Microsoft offer ML cloud services that simplify development by eliminating the need to build models from scratch. While these services accelerate ML adoption, recurring misuses frequently arise, degrading system quality and maintainability.

Although prior work has examined specific misuse cases in areas such as object-oriented programming, cloud services, and ML-based systems, the literature still lacks a comprehensive treatment of ML cloud service misuses in terms of their specification, detection, and refactoring. This project addresses this gap through three main contributions: (1) a catalog of bad practices in ML cloud service usage, (2) a highly automated detection approach for identifying these misuses, and (3) an automated refactoring strategy for removing them.

To build the catalog, we conducted a multi-vocal empirical study combining an academic and gray literature review, a manual analysis of 377 GitHub projects using ML cloud services, and a survey of 50 industry practitioners. This study resulted in the identification of 20 distinct ML service misuses.

We propose MLmisFinder, an automated detection approach based on a metamodel and rule-based detection algorithms targeting seven misuse types. It was evaluated on 107 open-source projects, achieving an average precision of 96.7% and a recall of 97%, and demonstrated strong scalability across 817 ML service–based systems.

Finally, we explored automated refactoring using Large Language Models (LLMs), with **GPT** being the best-performing model in 58% of the cases, achieving up to 82% accuracy and the fastest average execution time of 3.86 seconds.

Keywords: Bad Practices, Machine Learning Services, Specification, Detection, Refactoring

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

Page

LIST OF ABBREVIATIONS

AI	Artificial Intelligence
AST	Abstract Syntax Tree
DL	Deep Learning
ÉTS	École de Technologie Supérieure
LLM	Large Language Models
ML	Machine Learning
SDK	Software Development Kit
SOTA	State Of The Art

INTRODUCTION

0.1 Context

Major cloud providers now offer a broad range of Machine Learning (ML) cloud services, allowing developers of diverse skill levels to integrate advanced ML functionalities into software systems (Wan, Liu, Hoffmann, Maire & Lu, 2021). This accessibility has significantly accelerated the adoption of ML technologies, simplifying model development and reducing the need for deep ML expertise. The rise of ML services, from pre-trained APIs to highly customizable AutoML solutions (Google Cloud, 2025) (Sagemaker, 2025), has enabled the rapid development, deployment, and evolution of ML service-based systems. Despite these benefits, the accelerated pace of ML service integration has also introduced new challenges. Developers often rely on ML cloud services without fully understanding best practices or the implicit and explicit constraints imposed by cloud providers. Consequently, misuse of ML services has become increasingly common, negatively affecting system accuracy, performance, cost-efficiency, and maintainability (Wan *et al.*, 2021).

0.2 Problem

While existing research has explored ML (anti)patterns, code smells, technical debt, and quality concerns in ML-based systems, misuses specific to ML cloud services remain insufficiently studied (Zhang, Cruz & Van Deursen, 2022; Washizaki, Uchida, Khomh & Guéhéneuc, 2019; Masuda, Ono, Yasue & Hosokawa, 2018). Prior work has examined isolated misuse scenarios, but the field lacks a consistent terminology, structured specification, or comprehensive characterization of ML service misuses. The absence of clear specifications affects both researchers and practitioners: developers continue to violate ML service usage constraints, often unknowingly, while existing tools fail to detect these issues effectively. Such misuses can lead to critical failures, for instance, exceeding ML API rate limits without proper handling could

increase latency. Also omitting stopping criteria during training for example could lead to higher cloud usage costs. Moreover, prior research has given little attention to automated detection or refactoring approaches for ML service misuses, leaving a significant gap in the lifecycle support of ML service-based systems.

0.3 Objectives

To address these challenges, this project targets the following research objectives (ROs):

- **RO1.** Specify and characterize recurring misuses of ML cloud services, providing a comprehensive understanding of how these services are commonly misapplied in practice.
- **RO2.** Develop an automated approach for detecting ML service misuses through static analysis of source code, enabling developers to receive timely feedback and support during both development and maintenance activities.
- **RO3.** Propose an LLM-based refactoring approach for correcting ML service misuses, with the goal of improving overall quality, maintainability, and performance of ML service-based systems.

0.4 Contributions

This project aims to provide a comprehensive contribution to the understanding, detection, and mitigation of ML service misuses through three complementary contributions:

1. **The Specification of ML Service Misuses:** We conducted a multi-vocal empirical study, including a review of academic and gray literature, analysis of open-source GitHub systems, and a survey of industry practitioners, to systematically identify and define ML service misuses. The outcome of this study was a catalog of 20 ML cloud service misuses providing a consistent terminology and a unified characterization of existing misuses across cloud providers. Through the survey with ML practitioners, we measured the level of agreement with the proposed misuse catalog as well as the frequency with which developers encounter

these misuses in practice. We also identified the impacts and challenges of using ML services, the underlying reasons behind the introduction of ML service misuses, and the mitigation solutions from an industrial perspective. This work was accepted to the ACM Transactions on Software Engineering and Methodology (TOSEM) journal in January 2026.

2. **The Detection of ML Service Misuses:** We proposed MLmisFinder, an automated approach built on a novel metamodel representing ML service-based systems in a provider-agnostic manner. MLmisFinder uses extensible static detection rules to identify seven types of ML service misuses. The evaluation of MLmisFinder on 107 systems showed a precision of 96.7% and a recall of 97%, outperforming a state-of-the-art approach and uncovering 340 misuse instances. A large-scale analysis of 817 systems further revealed the widespread nature of these misuses and demonstrated the efficiency of MLmisFinder, with an average execution time of 50.05 seconds to detect the seven misuses. The approach is both efficient and scalable, with execution time scaling linearly with project size. This work was accepted to the 33rd IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER) in December 2025.
3. **The Refactoring of ML Service Misuses:** We proposed an approach to automatically refactor ML service misuses using LLMs. We evaluated the capability of four types of LLMs to refactor 340 ML service misuses occurrences. Our evaluation showed that **GPT** (gpt-oss-20B) is the best-performing model in 58% of the cases, achieving an accuracy of 82%, and is also the fastest model with an average execution time of 3.86 seconds per refactoring.

Overall, this project is the first to provide the largest catalog of 20 ML service misuses, an automated detection approach of seven types of misuses, only one of which had been previously detected in the literature, and the first systematic evaluation of LLMs for refactoring ML service misuses. These contributions demonstrate both originality and practical value, offering a foundation for practitioners and researchers to detect, refactor, and maintain ML service-based

systems more effectively, thereby improving maintainability, reliability, and compliance across cloud providers.

0.5 Manuscript Organization

The remainder of this manuscript is structured as follows: Chapter 1 presents a comprehensive literature review on ML cloud services, software quality issues, and existing work on ML (anti)patterns, as well as on refactoring test smells and best practices for refactoring guidelines. Chapter 2 introduces the specification of ML service misuses derived from our multi-vocal empirical study. Chapter 3 describes our automated detection approach and evaluates its effectiveness across real-world ML service-based systems. Chapter 4 focuses on the evaluation of different LLMs for automated refactoring ML service misuses. Finally, we outlines the conclusion and potential future research directions.

CHAPTER 1

LITERATURE REVIEW

Introduction

To contextualize our study, this chapter examines the existing body of research across three complementary domains. First, we review prior work on the specification of ML (anti)patterns and bad practices, highlighting existing taxonomies and their limitations regarding cloud-based ML services. Second, we analyze approaches for detecting such misuses, particularly those that target ML frameworks or API usage. Finally, we explore LLM-assisted refactoring techniques that aim to correct software issues while preserving code correctness and functionality. By synthesizing insights from these domains, this review underscores the need for a focused investigation into ML service misuses and motivates our subsequent efforts in cataloging, detecting, and refactoring them.

1.1 Specification of (anti)patterns and bad practices

This section provides an overview of the key literature related to our study. We discuss previous work in three areas: (1) ML API misuses, (2) ML technical debts, since certain misuses can introduce such debts, and (3) ML quality models, as certain ML misuses can have an impact on software quality attributes.

1.1.1 ML API misuses

The use and misuse of ML cloud services remain largely underexplored, with a few exceptions. (Wan *et al.*, 2021) analyzed 360 open-source software systems that use Google and AWS ML APIs and identified eight ML API misuses that negatively impact the quality of these systems. However, the authors only analyzed ML cloud APIs that give access to pre-trained models. Unlike our work, ML service misuses related to data preprocessing, model training, and model deployment were not covered. In addition, some misuses were generic in the sense that they correspond

to any cloud API, not exclusive to ML APIs. As a result, only four misuses from their study were considered in our study. Another study was conducted by (Wei *et al.*, 2024), in which they covered Deep Learning (DL) API misuse in Python libraries, such as TensorFlow and PyTorch, exploring their characteristics and proposing an approach to detect them. However, their work did not specifically address misuse issues related to ML cloud services. Other research studies addressed the specification of ML (anti)patterns (Masuda *et al.*, 2018; Washizaki *et al.*, 2019; Bogner, Verdecchia & Gerostathopoulos, 2021). In particular, (Washizaki *et al.*, 2019) conducted a literature review and a survey with developers to collect, classify, and discuss (anti)patterns for ML-based systems. The authors found that developers have little knowledge of ML design patterns that could aid in developing their software systems. (Zhang *et al.*, 2022) explored the gap in software engineering practices for machine learning by introducing a catalog of 22 ML-specific code smells. They systematically gathered these smells from research literature, gray literature, commits, and Q&A platforms, and classified them by affected pipeline stage, long-term risks, and possible fixes to support better code maintainability in ML systems. In contrast, our work focuses specifically on ML misuses in cloud-based environments, which introduce unique challenges not addressed by Zhang *et al.*'s taxonomy. Furthermore, we complement our catalog with a practitioner survey to assess the level of agreement on these misuses and their perceived frequency in real-world cloud-based ML systems.

1.1.2 ML technical debts

(Bogner *et al.*, 2021) conducted a literature review to study technical debts in ML-based systems and antipatterns. The authors argued that there is still no comprehensive conceptual overview of technical debts in ML-based systems, despite existing research efforts. They compiled a set of 72 ML antipatterns aimed at avoiding technical debt in the literature. However, they did not specifically study ML service antipatterns. (Sculley *et al.*, 2015) discussed the concept of ML technical debt by highlighting the long-term maintenance challenges and risks associated with deploying ML systems in real-world environments. They identified several ML-specific risk factors that contribute to system-level antipatterns, emphasizing the hidden complexities that arise when ML components are integrated into production systems. Moreover, (Recupito

et al., 2024) continued the investigation of ML technical debt by focusing on the practical challenges faced by ML practitioners and identifying recurring technical issues that contribute to long-term software quality degradation. While these studies provide valuable insights into ML-related technical debt, we aim in our study to provide a comprehensive catalog of ML service misuses, which can introduce technical debt by embedding hidden complexities and long-term maintenance challenges into the system. By investigating the prevalence of these misuses in both open-source and industrial projects, we seek to understand their role in accelerating the accumulation of technical debt in ML services. In addition, we will analyze the underlying causes of these misuses and evaluate possible mitigation strategies from an industrial perspective, offering actionable guidance to potentially minimize technical debt in ML systems.

1.1.3 Quality models of ML-based systems

With respect to the software quality of ML-based systems, there has been little research in the literature addressing the need for adapted quality models (Wang, Yang, Li, Damian & Lo, 2024; Masuda *et al.*, 2018; Lewis, Ozkaya & Xu, 2021; Kuwajima, Yasuoka & Nakae, 2020). For example, (Masuda *et al.*, 2018) summarized software quality methods and techniques for ML-based systems. (Kuwajima *et al.*, 2020) studied the suitability of a traditional software quality model for ML systems and provided insights into the key differences between ML systems and conventional software systems. (Cardozo, Dusparic & Cabrera, 2023) analyzed the code quality of reinforcement learning projects by utilizing software metrics related to the definition, access, and interaction of program entities. These metrics served as a proxy of code quality, aiding in the detection of code smells. (Cabral *et al.*, 2024) investigated the impact of applying SOLID design principles to machine learning code, highlighting that adherence to these principles can significantly improve code understanding among data scientists. Their study suggests that established software engineering practices can improve maintainability and readability in ML projects, even when developed by practitioners of diverse educational backgrounds, emphasizing the importance of promoting design principles within the data science community. We leveraged the findings in these studies to specifically target ML service misuses that can negatively impact

the quality of software systems. Identifying such misuses would greatly help increase awareness of common pitfalls when building ML service-based systems and improve their overall quality.

Despite the increasing focus on antipatterns, technical debt, and quality concerns in ML-based systems, there remains a gap in identifying and characterizing misuses specific to cloud-based ML services across the ML lifecycle. Existing studies often focus on general ML design issues, local code-level antipatterns, or traditional ML frameworks, paying little attention to the unique practices and risks introduced by modern cloud-based ML service platforms. Our study addresses this gap by conducting a multi-vocal study on ML cloud services misuses, spanning from data processing to deployment and monitoring. Through a combination of research literature review, gray literature analysis, a practitioner survey, and examination of real-world projects, our aim is to raise awareness of recurring misuse patterns and support the adoption of ML services best practices in software systems.

1.2 Detection of ML (Anti)patterns and Bad Practices

Several approaches in the literature have been proposed to specify and detect ML (anti)patterns (Masuda *et al.*, 2018; Bogner *et al.*, 2021). However, only a few of them specifically focused on the specification and detection of ML service misuses. For example, (Wan *et al.*, 2021) analyzed open-source systems that leverage Google and AWS ML APIs. Their study identified eight ML API misuses that negatively impact software quality. However, their research was limited to ML cloud APIs that provide access solely to pretrained models. Unlike our detection approach, they did not explore misuses related to ML services in the context of data preprocessing, model training, and model deployment. In contrast to their work, our approach targets a different set of ML service misuses and introduces a highly automated tool for their detection. Moreover, (Wei *et al.*, 2024) conducted an analysis aimed at understanding and detecting misuses of DL APIs in frameworks such as PyTorch and TensorFlow. They developed an LLM-based API misuse detection tool, LLMAPIDet, which is designed to identify and correct these misuses. However, their focus was primarily on DL APIs within the TensorFlow and PyTorch ecosystems, and did not extend to ML service misuses in ML service-based systems. (Shivashankar & Martini,

2025) introduced MLScent, a static analysis tool that detects ML-specific code smells across frameworks such as TensorFlow and PyTorch. They highlighted the need for ML-aware code quality tools, as traditional code smell detectors often overlook patterns that can negatively impact model performance, reproducibility, and maintainability. However, their work focuses on model-level code quality and does not address challenges specific to ML service–based systems. (Washizaki *et al.*, 2019) conducted a literature review and a survey with developers to collect, classify, and discuss (anti)patterns for ML-based systems. They found that developers have little knowledge on ML design patterns that could assist in developing their ML-based systems. (Van Oort, Cruz, Loni & Van Deursen, 2022) defined project smells as a more holistic approach to code smells. They developed *mllint*, an open-source command-line utility to evaluate the software quality of Python ML projects by performing static analysis on the project's source code. Although this study aims to assist ML practitioners in developing and maintaining production-grade ML projects, it does not specifically address project smells in software systems that integrate ML services.

While previous research has explored some ML (anti)patterns, a gap remains in studying ML cloud service-specific misuses. Unlike prior work focused on general ML (anti)patterns and code smells, our focus is to specifically targets misuses in ML service-based systems. By introducing a highly automated detection method, we address key challenges in the effective integration and use of ML cloud services.

1.3 Refactoring and Best Practices for Correcting Software Antipatterns

Automated code refactoring has long been a focus of software engineering research, with traditional approaches relying on static analysis, rule-based transformations, or human-guided best practices (Mohan & Greer, 2018; Zhang, Shen, Zhang, Zheng & Zheng, 2025; Li & Zhang, 2024). More recently, LLMs have emerged as powerful tools for both detecting code smells and performing automated refactoring, enabling a new paradigm that combines machine understanding with developer expertise(Guo *et al.*, 2024; Cordeiro, Noei & Zou, 2025).

For example, (Oueslati, Lamothe & Khomh, 2025) investigates the ability of relatively small, open LLMs, LLaMA-3.2-3B, Gemma-2-9B, DeepSeek-R1-14B, and Phi-4-14B, to detect and refactor test smells in Java code using multi-agent workflows. The study evaluated one, two, and four-agent setups, where agents specialized in detecting, confirming, refactoring, and validating code changes. Key findings indicate that multi-agent configurations improve detection accuracy and iterative refinement of refactorings, with Phi-4-14B achieving high-quality refactoring results comparable to proprietary LLMs while remaining cost-effective. Moreover, the approach generalized across multiple programming languages, and some refactorings were merged into open-source projects, demonstrating practical utility. This work highlights the potential of LLMs to automate localized code quality improvements but focuses specifically on test code rather than ML service usage. Moreover, (Kuang Piao *et al.*, 2025) explored how instruction strategies inspired by human best-practice refactoring guidelines (e.g., Martin Fowler's catalog) can improve LLM-based refactoring performance. The study shows that LLM performance is highly sensitive to how instructions are formulated. DeepSeek-V3 consistently outperformed GPT-4o-mini, particularly for benchmarked refactoring types, and preserved code semantics in real-world GitHub projects. Findings also suggest that objective-driven strategies, which provide high-level goals rather than explicit transformations, can lead to meaningful improvements in code quality metrics. Challenges observed include hallucinations, compilation failures, and semantic errors when the model lacks sufficient context. This work emphasizes the importance of instruction design and evaluation in LLM-assisted refactoring but primarily addresses general-purpose and localized refactorings rather than domain-specific service misuses.

While prior studies have demonstrated the utility of LLMs for refactoring test code or general-purpose software, to the best of our knowledge, no work has evaluated the use of LLMs for refactoring ML service misuses. These misuses, are highly domain-specific and require an understanding of ML cloud services (e.g., Azure ML, Amazon SageMaker, Google Vertex AI). Our work is the first to investigate whether LLMs can refactor such misuses, assessing both correctness and minimal disruption to existing code.

Conclusion

In this chapter, we reviewed existing research on the specification, detection, and refactoring of ML (anti)patterns. This review demonstrates that while prior work has focused on general code smells and test-related misuses, no study has systematically addressed ML service misuses. Building on these insights, the next chapter focuses on the specification of ML service misuses, providing a structured foundation for their detection and subsequent refactoring.

CHAPTER 2

A COMPREHENSIVE MULTI-VOCAL EMPIRICAL STUDY OF ML CLOUD SERVICE MISUSES

Hadil Ben Amor[1] , Manel Abdellatif[1] , Taher A. Ghaleb[2]

[1] Department of Software Engineering, École de Technologie Supérieure
1100 Notre-Dame West, Montreal, Quebec, Canada H3C 1K3
[2] Department of Software Engineering, Trent University
1600 West Bank Drive, Peterborough, Ontario, Canada K9L 0G2

Accepted to TOSEM in January 2026

Abstract

Machine Learning (ML) models are widely adopted across many application domains. To support this trend, major cloud providers offer ML cloud services that eliminate the need to build models from scratch. These ML services empower practitioners with varying expertise to rapidly embed ML capabilities into software systems. However, practitioners often overlook best practices and optimal design when using cloud-based services. This results in recurring misuses that can compromise system quality and may hinder long-term maintenance and evolution. Though prior work has studied some misuse cases, the field lacks consistent terminology and clear specifications of ML service misuses. In this work, we address three research questions (RQs) investigating (1) the types of ML service misuses have been identified in research literature, gray literature, and open-source software systems, (2) the state of the practice of ML cloud service misuses in industry, and (3) the prevalence of ML service misuses in open-source projects in comparison with what is observed in industry practice. To address these RQs, we conducted a comprehensive multi-vocal empirical study investigating the prevalence of ML cloud service misuses in practical contexts. Overall, our analyses incorporated a diverse range of sources, including academic research, official documentation from major ML cloud providers, and 377 GitHub projects using ML services, and were complemented by a survey of 50 ML practitioners from industry. As a result, we developed a catalog of 20 ML service misuses, cross-validated through evidence from both open-source projects and industry feedback. We observe that our

identified misuses are widely prevalent in both open-source projects and industry, often due to a lack of understanding of service capabilities, inadequate documentation, and poor awareness of best practices. This highlights the importance of ongoing education on best practices for ML services and highlights the need for tools to automatically detect and refactor ML service misuses.

2.1 Introduction

Major breakthroughs in Artificial Intelligence (AI) in general, and Machine Learning (ML) in particular, have brought huge success to many application domains such as image recognition, security, and autonomous driving (Grigorescu, Trasnea, Cocias & Macesanu, 2020; Dong, Wang & Abbas, 2021). To support this technological hype, several major cloud providers, such as Amazon, Google, and Microsoft, have been pivotal in providing ML practitioners with various ML cloud infrastructure capabilities and specialized services that support key ML development tasks (Wan *et al.*, 2021). The availability of ML cloud services has greatly facilitated the adoption and integration of ML in software systems. These services provide not only pre-trained models and ready-to-use APIs but also powerful Platform-as-a-Service (PaaS) solutions for developing ML-based systems. Vertex AI (Google Cloud, 2025) and SageMaker (Sagemaker, 2025) are examples of platforms that allow developers to build, train, and deploy custom ML models more efficiently by offering managed infrastructure, specialized tools, and integrated workflows. This support accelerates the development of ML-based systems and simplifies the integration of ML components into software systems. Existing ML cloud services cover a wide spectrum of services, from extreme simplicity and complete customizability, to build ML service-based systems (ML, 2024a).

For instance, some ML cloud services operate as black-box systems (i.e., not disclosing the ML classification models they employ) while others offer support throughout ML development stages, from data preprocessing to the selection of ML classifiers, parameter tuning, and model deployment (Microsoft, 2024a). The growing demand for ML-driven business solutions has led practitioners of varying expertise to increasingly adopt cloud-based ML services to accelerate

the development, maintenance, and evolution of ML systems (Wan *et al.*, 2021). However, this rapid pace often comes at the cost of adhering to best practices for designing and utilizing ML services (Wan *et al.*, 2021), resulting in recurring misuses that can degrade system quality, pose challenges in their maintenance and evolution, introduce bugs, and impact performance and cost-efficiency (Washizaki *et al.*, 2022).

ML service misuse is defined as poor practices in the use of services within the ML development pipeline, including data collection and preprocessing, training, testing, deployment, serving, and monitoring. These misuses are also perceived as a violation of (implicit) service usage constraints in end systems, potentially leading to inefficiencies or unintended consequences on end systems. A recent study reported that 69% of GitHub projects using Amazon and Google ML cloud services contain service misuses that directly impact system functionality and service costs (Wan *et al.*, 2021). As an example, failing to specify a stopping criterion during model training through an ML service can increase latency and cloud usage costs, as training might continue unnecessarily long without significant performance improvements (Policy, 2024). Another example is the availability of synchronous and asynchronous versions of the same ML task offered by several ML cloud providers. Figure 2.1 shows an example of such a misuse with Azure cognitive services for real-time sentiment analysis of short text messages. As illustrated in the figure, asynchronous services support larger inputs and longer processing times than synchronous services. These two API types differ in their request handling and implications for system performance. The synchronous API allows immediate processing of user requests and returns sentiment results with minimal latency, offering a responsive and cost-efficient user experience. In contrast, the asynchronous API requires the submission of a job and the repeated polling of the results, leading to longer processing times, higher resource usage, and poorer user experience when misused as a synchronous service.

ML cloud service providers usually set rate limits for their services. These limits define the maximum allowable requests within a specific time frame, helping to regulate ML service usage and prevent system overloads. However, exceeding such limits without proper handling can result in critical issues, such as increased latency and instability in ML service-based systems.

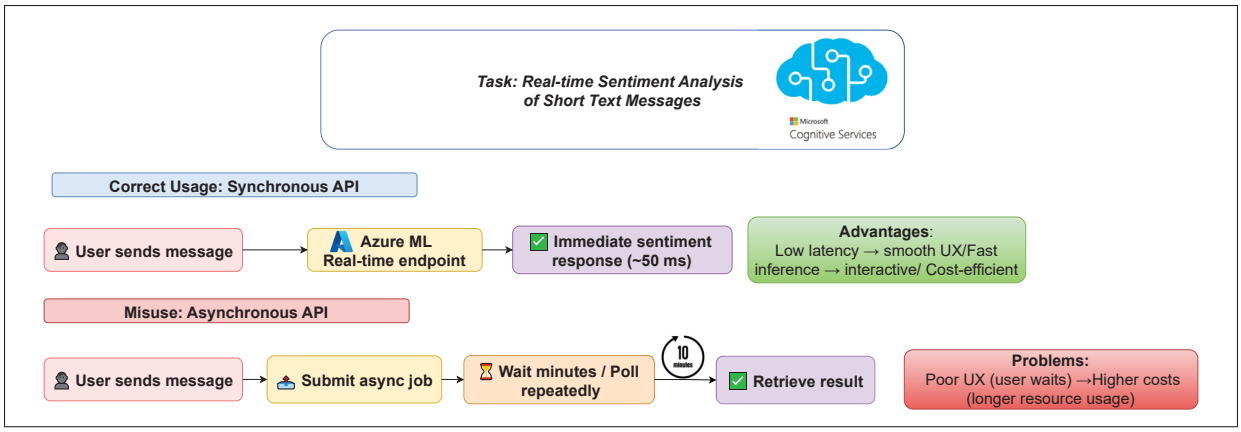


Figure 2.1
Synchronous and
Asynchronous API:
Correct Use vs. Misuse

Therefore, understanding ML service misuse in software systems is important for improving software quality, ensuring correctness, improving performance, and achieving cost efficiency.

Unlike ML or service antipatterns, which have well-established definitions in prior research (Washizaki *et al.*, 2022; Bogner *et al.*, 2021), ML service misuses still lack precise specifications and comprehensive understanding in practical contexts. In addition, existing research on ML service misuse is limited, with the highest number of misuses reported by a single study being eight (Wan *et al.*, 2021). Besides, the terminology used by prior research is often inconsistent and has vague definitions, which prevents practitioners from fully understanding and addressing any relevant issues. Recognizing these misuses can help increase awareness of common pitfalls when developing ML service-based systems, thereby potentially improving their overall quality. Therefore, in this work, we conduct a comprehensive, multi-vocal empirical study on common misuses in ML cloud services. Specifically, we address three research questions (RQs) that investigate: 1) the types of ML service misuses have been identified in research literature, gray literature, and open-source software systems, (2) the state of the practice of ML cloud service misuses in industry through an online survey with 50 ML practitioners, and (3) the prevalence of ML service misuses in open-source projects in comparison with what is observed in industry practice. We developed a comprehensive catalog of 20 misuses, categorized by the

ML development pipeline stage. We explored the prevalence of our catalog in the industry to understand the underlying causes and gain insights from practitioners on potential mitigation strategies.

Our empirical analyses revealed a strong agreement among practitioners on the validity and relevance of our catalog. The results show that ML service misuses are widely spread in industry, with "*Improper handling of ML API limits*" being the most frequent misuse. Practitioners indicated that these misuses typically arise from a lack of understanding of ML service capabilities, insufficient cloud documentation, and limited awareness of best practices for ML services. To address these challenges, practitioners emphasized the importance of regular performance reviews to maintain efficient ML service-based systems. They also advocated for a thorough validation of data quality, combined with continuous education on best practices, to ensure the correct use of ML services.

The contributions of this work are summarized as follows.

- We conducted a thorough review of the research literature related to ML systems, cloud services, and (anti)patterns, which resulted in 31 relevant papers, revealing a limited number of ML service misuse scenarios.
- We conducted a gray literature review to identify a more comprehensive set of ML service misuses that have not been previously studied in the context of ML service-based systems.
- We conducted an in-depth empirical analysis of a curated set of 377 GitHub repositories, employing a combination of manual and static code analysis to identify misuses of ML services, resulting in a validated dataset of such misuses.
- We developed a comprehensive catalog of 20 ML service misuses, categorized by the ML development pipeline stage. This catalog is the largest of its kind and provides a comprehensive overview of potential issues.
- We conducted a survey with 50 practitioners experienced in ML cloud services, who validated the catalog and provided insights into the use of ML services in the industry. The survey revealed the prevalence of ML service misuses, highlighted their potential causes, and their possible mitigation solution.

The remainder of this chapter is structured as follows. Section 2.2 highlights the motivation of our work. Section 2.3 describes our multi-vocal study. The catalog proposal and the analysis of the survey results are presented in Section 2.4. Section 2.5 discusses the implications of our findings. Section 2.6 discusses the potential validity threats to our findings. Finally, Section 2.7 concludes our work and suggests possible future research.

2.2 Motivation

Major cloud providers now offer a broad range of ML cloud services, enabling developers at all skill levels to integrate advanced ML capabilities into software systems. This accessibility has greatly accelerated the adoption and integration of ML technologies. However, with increasing business demands and the growing prevalence of ML services, developers are building, maintaining, and evolving ML-based systems at a rapid pace, often without adhering to established best practices. Such misuse of ML cloud services can significantly degrade system quality, making them more difficult to maintain and evolve (Wan *et al.*, 2021). While prior research has mainly examined (anti)patterns, quality concerns, and technical debts in ML-based systems, there has been no systematic identification and characterization of misuses specific to ML cloud services. Although prior work has studied some misuse cases, there is still a lack of consistent terminology and clear specifications of ML service misuses in both academia and industry. In this work, we aim to address this gap through a comprehensive multi-vocal empirical study on ML cloud service misuses. Our investigation combines (1) a review of both research and gray literature, (2) an empirical analysis of open-source GitHub projects, and (3) a survey of industrial practitioners to gain a deeper understanding of the current state of ML cloud service misuses.

Our multi-vocal study is guided by the following research questions (RQs):

- **RQ1. What types of ML service misuses have been identified in research literature, gray literature, and open-source software systems?** We aim to identify and categorize ML cloud service misuses described in both research and gray literature, as well as those observed in open-source software projects. Our objective is to build a comprehensive catalog

that details the definition, characteristics, and concrete implementation examples of each misuse.

- **RQ2. What is the state of the practice of ML cloud service misuses in industry?** Our goal is to explore practitioners' perceptions of ML cloud service misuses that we identify in RQ1. Our goal is to investigate their agreement on such misuses as well as their prevalence in industry. Besides, we aim to investigate the causes practitioners attribute to such misuses and the mitigation strategies they employ.
- **RQ3. How comparable is the prevalence of ML service misuses in open-source projects to what is observed in industry practice?** Our objective is to compare the prevalence of ML cloud service misuses in open-source systems with their occurrence in industry practice, to understand differences and similarities in misuse patterns across these environments.

2.3 Methodology

Figure 2.2 depicts the design of our multi-vocal study to build and analyze our catalog of ML service misuses. The study consists of five main steps, which we describe in detail in the following subsections.

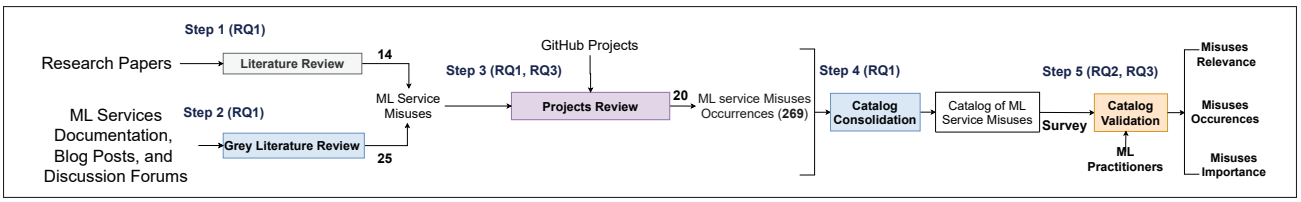


Figure 2.2 Overview of our methodology

2.3.1 Research Literature Review (RQ1)

To address our first research question, we conducted a structured review of the literature on the use and quality of machine learning systems in cloud-based environments. To ensure comprehensiveness and avoid missing relevant studies, we followed the approach guided by

established principles and the guidelines of (Kitchenham, 2004). Figure 2.3 depicts our process for paper selection.

Initially, we defined a set of keywords related to our study, including *machine Learning*, *misuse*, and *cloud services*. Then, we identified a corresponding set of synonyms for each keyword and formulated the following search query:

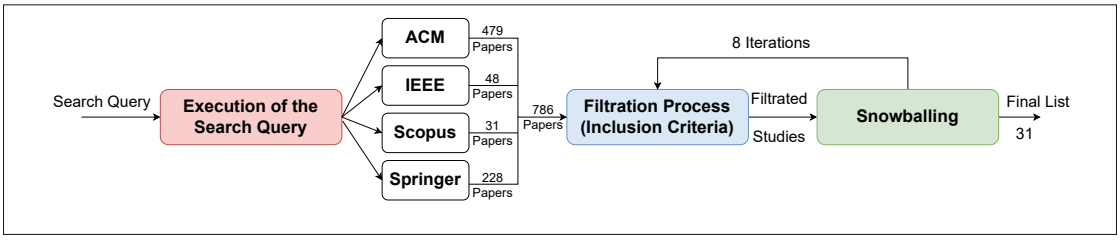


Figure 2.3 Papers selection process of our research literature review

We executed this query on popular digital libraries for research papers, including ACM, IEEE Xplore, Scopus, and Springer, leading to an initial list of 786 papers. The search has been refined by applying filters based on relevance and a publication date range limited to the past ten years. We then filtered these collected papers according to their titles and abstracts. Two of the co-authors (with experience in cloud-based ML services research) independently conducted a manual review of the papers, resolving disagreements through discussion to reach consensus. We calculated the Cohen's kappa coefficient (Banerjee, Capozzoli, McSweeney & Sinha, 1999), a statistic commonly used to measure inter-rater reliability. The resulting value was 89.79%, indicating a strong level of agreement between the reviewers. We included in our review those papers that met at least two of the following criteria: (1) studies related to ML; (2) studies related to cloud computing services; and (3) studies related to (anti)patterns. Finally, to minimize the risk of missing relevant studies, we applied forward and backward snowballing techniques (Wohlin, 2014), conducting eight iterations that resulted in the inclusion of six additional studies.

2.3.2 Gray Literature Review (RQ1)

In addition to our research literature review, we conducted a review of the gray literature on ML cloud service misuses, following the guidelines outlined by (Garousi, Felderer & Mäntylä, 2019). We started by searching for arXiv papers using the same search query terms and inclusion criteria as in our research literature review to ensure consistency in our scope. We then expanded our search to include official documentation and practical developer insights from various online sources, executing the same query on Google Search to capture a broad spectrum of gray literature.

Our search query led to the identification of 3,116 online references. To select only relevant resources, we applied the following selection criteria:

(1) We filtered out non-English content to ensure clarity and consistency.

(2) We selected resources from only highly credible platforms known for their rich, practical insights and widespread developer usage. These platforms include (1) Official Documentation from major ML cloud providers: Microsoft, Amazon, and Google; and (2) Official Cloud Technical Blogs and Developer Platforms: Microsoft Tech Community[1], Google Developers Blog[2], AWS Developer Blogs[3], Medium Blogs[4]; and (3) Specialized books that cover the design and usage of ML cloud services (Tranquillin, Lakshmanan & Tekiner, 2023; Ping, 2022). These sources provide diverse perspectives, ranging from official best-practice recommendations to real-world challenges and common misuses reported by developers.

(3) We selected online references that were relevant to the context of our study and aligned with the latest developments in ML cloud services.

(4) We removed duplicates and outdated materials, specifically those published before the most recent stable release of each cloud or ML service at the time of our research, to ensure up-to-date, quality sources.

---

[1] https://techcommunity.microsoft.com

[2] https://developers.googleblog.com/en

[3] https://aws.amazon.com/blogs/developer

[4] https://medium.com

We performed five iterations of backward and forward snowballing, which yielded 12 additional relevant sources on ML cloud service misuses. We should note that to ensure consistency with our research literature review, we applied a publication date filter from 2017 to 2024, as this period reflects the most relevant and recent retained papers. Moreover, to ensure a focused and manageable review, we reviewed references until we found mostly non-relevant links. Specifically, after reviewing the first 10% of references in each source and encountering ten consecutive non-relevant links, we stopped further analysis. This approach allowed us to retain the most relevant resources while keeping the review practically feasible and representative.

2.3.3 Analysis of GitHub Projects (RQ1 & RQ3)

To collect concrete examples of ML service misuses (RQ1) and study their prevalence (RQ3) in open-source software, we conducted an in-depth analysis of various repositories hosted on GitHub, while relying on the preliminary catalog of the misuses identified in the previous steps. We initially started with the dataset provided by (Wan *et al.*, 2021) and observed that it contains 307 Python projects covering eight ML service misuses. Moreover, their dataset focused exclusively on ML APIs from Google and Amazon, which limited the scope of ML applications. Considering the lack of similar datasets that satisfy our needs, we proceeded to collect additional GitHub projects, as described below.

a) **Projects Collection:** We collected ML service-based GitHub projects, a rich source of open-source ML-based software systems. The data collection and analysis were carried out during the period between May 2024 and July 2024, without a specific time frame for the creation date of such projects. We used the *GitHub API*[5] to automatically retrieve projects, enabling a more efficient and structured retrieval of information. We used specific search keywords listed by the cloud service providers under study. In addition, we employed GitHub's search functionality to identify the presence of those keywords within Python files. For example, to collect repositories using Microsoft Azure ML, we considered the following keywords: *"ML cloud", "Azure cognitive service", "API", "Azure AI", and "AutoML"*. The

[5] https://docs.GitHub.com/en/rest

collected information about the projects includes the repositories' URLs, number of stars, number of forks, contributors, and descriptions. All this data is available in our replication package (Ben Amor, Abdellatif & Ghaleb, 2024).

b) **Projects Filtration:** We filtered the collected repositories by automatically analyzing the project description and source code. This involved inspecting the development language, identifying the presence of Jupyter notebooks, and locating Python files. This allowed us to filter repositories based on their relevance to our study. The project collection process resulted in 1,900 Python GitHub repositories related to ML services. We randomly selected a statistically significant sample of 320 projects, calculated based on a 95% confidence level and a 5% margin of error for a population of 1,900[6], for further manual analysis. However, we analyzed 57 more projects (leading to a total of 377 projects) to strengthen the reliability and depth of our analysis. Our goal was to ensure broad representation by including repositories related to major cloud providers (i.e., Microsoft, Google, and Amazon) and diverse types of ML services (e.g., text-to-speech, image recognition, and video processing).

However, we found that not all repositories that mention cloud services or ML-related terms really use ML services. Hence, to maintain the focus on projects that use cloud-based ML services, we excluded repositories that met at least one of the following criteria:

- **Toy or tutorial projects:** Repositories created solely for educational purposes (e.g., short workshop demos) were excluded, as they do not represent real-world development practices or deployed systems. For example, a repository[7]demonstrating how to call an ML service once using a minimal script was excluded even though it mentioned "*Azure ML*".

[6] https://www.calculator.net/sample-size-calculator.html?type=1&cl=95&ci=5&pp=50&ps=1900&x=Calculate

[7] https://github.com/gowhich501/customVisionApi

- **Official Toolkits from Cloud Providers:** Such repositories[8,9,10], often provided by cloud providers (e.g., "*AzureML examples*" or "*aws-sagemaker-notebook-samples*"), serve as starter templates and do not reflect actual usage scenarios authored by practitioners. Although they are technically related to ML services, their primary purpose is to present product features rather than complete projects with context-specific implementations.
- **Misleading mentions of ML services without actual use:** Some repositories mention cloud providers or ML concepts in their *ReadMe* files or project descriptions, but do not make use of any cloud-based ML service in their source code. For example, we excluded a project that deployed a microservice on Google Cloud Platform and mentioned "*Google AI*" in its documentation but did not interact with any ML services (e.g., Vertex AI) in the actual codebase.

Two of the co-authors (with significant experience in machine learning and software engineering) played the role of evaluators and manually applied the above exclusion criteria independently, yielding a set of 87 ML service-based projects for further analysis. Their inter-rater agreement for this filtering stage, measured using Cohen's kappa, was 100%, indicating a high level of consistency.

c) **Projects Analyses:** The same co-authors manually and independently analyzed the projects to detect the presence of different ML service misuses. We used Understand[11], a static code analysis tool, to perform a thorough code review and control flow graph generation in complex repositories containing numerous directories. This tool provides different views of the architecture, thus enabling a comprehensive understanding and analysis of the systems.

d) **Misuses Isolation:** As part of the detection process, the two co-authors independently and systematically identified and highlighted instances of ML service misuse, categorizing them by their stage of the ML development pipeline. Each detected misuse is accompanied by a detailed explanation, which provides reasons as to why the identified instance is problematic

[8] https://github.com/aws-samples/reinvent2021-tsa-bot

[9] https://github.com/aws-samples/amazon-sagemaker-groundtruth-and-amazon-comprehend-ner-examples

[10] https://github.com/Azure-Samples/ai-rag-chat-evaluator

[11] https://scitools.com

and a link to the exact line of code within the repository, facilitating quick access for further inspection. For example, to detect "*Non specification of early stopping criteria*" misuse, the two co-authors independently analyzed the ML service-based system, identified the ML service cloud provider used, verified the training component of the code, and reviewed the recommendations for specifying early stopping criteria based on the official documentation of the cloud provider. If the training component did not adhere to best practices for cost-efficiency and resource optimization, a misuse was detected.

e) **Detection Agreement:** The two co-authors tried to resolve any disagreement during the process through discussion meetings. In cases where a consensus could not be reached, a third coauthor was involved to provide additional insights and resolve disagreements. To assess the level of agreement between the evaluators, we calculated Cohen's kappa coefficient (Banerjee *et al.*, 1999). We achieved Cohen's kappa of 0.85, indicating a high level of agreement among the evaluators and highlighting the reliability of the manual identification process of the occurrences of ML service misuses.

2.3.4 Consolidation of the Catalog of ML Service Misuses (RQ1)

We curated a preliminary catalog of ML service misuses by combining insights from the research literature, the gray literature, and real-world examples from open-source projects (RQ1). The identification process involved iterative refinement and pre-validation of the identified misuses from each dimension of our multi-vocal study. The two co-authors collaboratively reviewed and evaluated in regular meetings the relevance of the identified misuses, applying the following inclusion criteria: (1) misuses specific to ML service–based systems, excluding those applicable to general service systems; (2) misuses applicable to all cloud providers; (3) misuses with occurrences identified within GitHub projects; and (4) misuses that are potentially detectable through source code analysis. Consensus was reached through a consultation with the third co-author. We also categorized the misuses according to their stage within the ML development pipeline. The final curated set served as the basis for a practitioner survey in which we asked developers about the relevance and occurrence of these misuses in their experience.

2.3.5 Online Survey with Practitioners (RQ2 & RQ3)

We conducted an online survey with practitioners to validate our catalog and gain better insight into the usage of ML cloud services in the industry (RQ2). This also enabled us to examine and compare the prevalence of ML cloud services misuses across both industry and open-source projects (RQ3). To build the survey, we followed established guidelines in the literature (Molléri, Petersen & Mendes, 2020; Fowler Jr, 2013) that provide recommendations for different stages of the survey, such as question design, sampling, data collection, and analysis. These recommendations helped us identify and structure the different parts of the survey, craft precise questions, and set suitable response options. The survey consisted of three main phases, explained below.

(1) **Preparation of the Online Survey.** We created a web-based survey[12] using Google Forms. We designed our survey questions based on the research literature review, the gray literature review, and the analysis of ML service-based systems. Before publishing the survey, we conducted a pilot interview with three professionals and four researchers to identify any duplicated, inconsistent, or unclear questions in the survey before distribution. This also helped to assess the survey length, which might discourage participants from responding, and to estimate the approximate time required to complete the survey. The seven participants went through the survey and suggested minor modifications. To facilitate participants' understanding of ML service misuses, we included a link to our online catalog[13] in the survey to further explain each misuse. The final survey consisted of seven sections, primarily about the demographics of the participants, the frequency of usage of ML services, the introduction of ML misuses, the agreement level with the catalog, and the possible reasons for misuses.

(2) **Selection of Participants.** We targeted practitioners with experience in ML cloud services. Identifying and inviting such practitioners was challenging due to the key requirement for familiarity with both ML and cloud computing, as well as practical experience with ML services in their work. To select the participants, we relied on (1) information about companies

[12] https://forms.gle/Na3sCzyWt6WmTA7e9

[13] https://ml-service-misuses.GitHub.io/Catalog-Proposal

that offer ML cloud services and their clients/partners, (2) the authors' networks (e.g., former colleagues), and (3) search queries on LinkedIn profiles (e.g., `AI Engineer OR Cloud Architect OR Azure certified OR Cloud for AI`). For those who were identified as potential practitioners, we sent them invitations via EMail and LinkedIn messages. We chose to limit our invitations to a maximum of five practitioners per organization to ensure a wide representation and mitigate the risk of bias. We invited a total of 421 practitioners to complete the survey and encouraged them to share our invitations within their network. The survey was completed by 50 practitioners, which corresponds to a 12% completion rate, which is somewhat reasonable for practitioner-oriented surveys in software engineering research, as achieving high response rates is challenging in this kind of studies (Ghazi, Petersen, Reddy & Nekkanti, 2018).

(3) **Data Validation.** To ensure the reliability and accuracy of the survey responses, we employed a multi-step validation process. First, we performed a preliminary screening of the responses to identify incomplete or inconsistent answers. Responses that contained missing critical information or exhibited contradictions were flagged for further verification. To ensure that participants had experience with ML cloud services, we verified their eligibility based on multiple self-reported indicators collected in the survey, including the number of ML-service based projects they had worked on, the specific cloud-based ML technologies used (e.g., SDKs, APIs, and UI), the frequency of using ML services, and their years of experience in AI, software development, and cloud computing. Participants whose responses revealed no relevant background or whose responses were contradictory would have been excluded from further analysis. However, no participants met these exclusion criteria. After that, we extended invitations for follow-up interviews with participants to confirm their responses, validate key findings, clarify ambiguities, and ensure that the survey data accurately reflects the experiences of the participants. By following these validation steps, we aimed to improve the robustness of our findings and minimize potential biases in the survey results.

Statistical Correlation. We measured Spearman's correlations (Ali Abd Al-Hameed, 2022) between (a) participants' experience levels in AI development and the number of ML service

misuses they agreed upon, as well as (b) participants' experience levels and the frequency of encountering each ML service misuse. We used Spearman's correlation as it does not assume a specific distribution of the data. The goal was to explore whether more experienced practitioners report being aware of, or exposed to, a higher number of ML service misuses in practice. Understanding such trends helps us to assess whether awareness of misuses increases with expertise or, conversely, if misuses are prevalent regardless of experience level. These insights provide useful implications for targeting awareness tools and educational materials toward specific practitioner groups.

Additional Qualitative Insights: To complement the survey data and provide richer context to our findings, we conducted a qualitative analysis of practitioner-oriented online resources. These included blog posts, technical documentation, and Q&A forums such as Stack Overflow, which we used solely for qualitative analysis rather than for catalog construction and consolidation. We also examined research papers discussing challenges and impacts related to cloud-based ML services. Insights from this analysis helped us confirm the reported misuses, understand their introduction reasons and mitigation strategies, and strengthen the interpretation of our survey results.

2.4 Results

This section presents the results of our study with respect to our research questions. We first describe the catalog of ML service misuses identified through our reviews of the research literature and gray literature, followed by our analysis of GitHub projects (RQ1). We then report the findings derived from our survey (RQ2) and compare the prevalence of the misuses identified in industry settings and open-source projects (RQ3).

2.4.1 RQ1. What types of ML service misuses have been identified in research literature, gray literature, and open-source software systems?

To address RQ1, we analyzed findings from both research and gray literature, complemented by empirical evidence from open-source software systems. Our goal was to identify and consolidate

the different types of ML service misuses reported across these sources. In the following sections, we first present the findings of our research literature review, then proceed with the findings of our gray literature review, and subsequently discuss how these insights, combined with the mining of open-source software systems, are integrated into a consolidated catalog of ML service misuses.

2.4.1.1 Results of the Research Literature Review

As shown in Figure 2.3, our search query and filtration process led to the inclusion of 31 studies published between 2018 and 2024. Table I-1 presents the selected studies, along with their publication years and a brief summary of each study. We observe that the highest number of reported ML service misuses in a single study was eight, as identified by (Wan *et al.*, 2021). This underscores the need for a comprehensive and systematic catalog of ML service misuses to better understand and mitigate common pitfalls in using such services in practice. The distribution of the studies collected over the years is also depicted in Figure 2.4, illustrating the trends in the research output from 2018 to 2024. The number of publications has generally increased over time, with noticeable fluctuations. A steady rise is observed from 2018 to 2021, reaching a peak in 2022, where the highest number of publications (approximately 12) was recorded. This is followed by a decline in 2023, before rising again in 2024, suggesting a continued interest in the intersection of ML, cloud computing, and software quality. The observed trend indicates growing awareness and research focus on ML misuses and best practices within ML and service-based systems. In addition, we identified 14 distinct misuses across different stages of the ML lifecycle. Most of these misuses occur during data collection and preprocessing, including inefficient data transmission and failure to use batch APIs, which often degrade data quality before training. In the training stage, we found one misuse related to the avoidance of parallel training experiments, which can pose delays in iteration and limit model optimization. The testing stage revealed three misuses, including ignoring fairness evaluation, ignoring testing schema mismatches and misinterpreting output. In deployment, we identified three misuses, including neglecting automatic rollback mechanisms for production models. Finally, we observed one misuse in the

serving stage related to calling the wrong ML service API, and one misuse in monitoring related to ignoring data drift monitoring. The full list of misuses is presented and described in more detail in Section 2.4.1.4. We also report the misuses in our replication package (Ben Amor *et al.*, 2024).

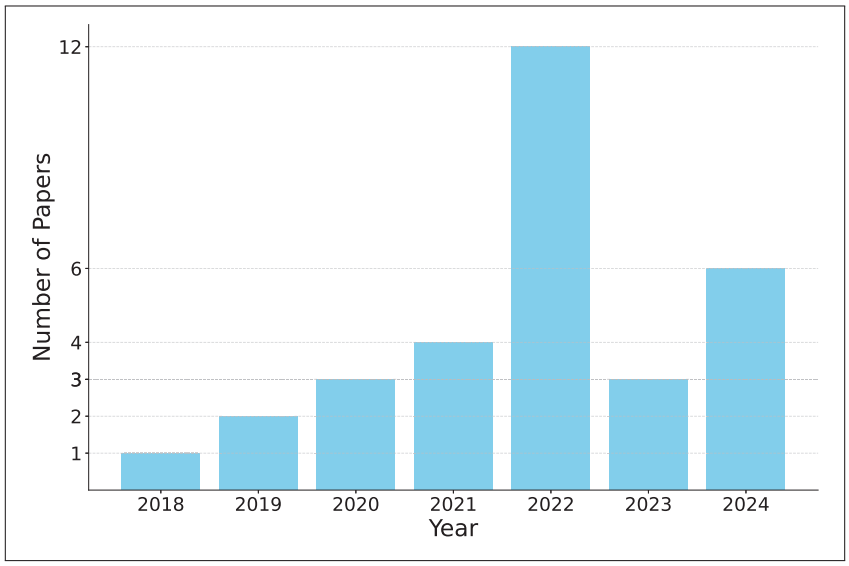


Figure 2.4 Number of Studies Published Per Year

2.4.1.2 Results of the Gray Literature Review

We provide the results of our search query from each source of the gray literature review in Table 2.1, along with a detailed summary of the arXiv studies in Table II-1. We observe eight arXiv publications spanning the period from 2020 to 2024, which reflects recent discussions in the gray literature, with more than half of the studies published in 2023 and 2024. More importantly, we found that none of the identified studies specifically address ML service-based systems. Though these studies cover relevant topics, such as technical debt, antipatterns, and general software engineering challenges, they do not consider the unique characteristics, practices, or misuses associated with ML service-based systems. This indicates a notable gap in the existing gray literature and highlights the need for targeted research in this emerging area. We identified 25 misuses spanning the various stages of the ML lifecycle. During data collection and preprocessing, notable misuses include data leakage, storing data in block

storage, and not using batch API for data processing. In the training stage, we observed misuses such as non-specification of early stopping criteria, over-reliance on default training settings/service configurations, not using automatic hyperparameter tuning, and excluding algorithms in automated ML. The testing stage revealed misuses such as ignoring testing schema mismatch, ignoring fairness evaluation, and using suboptimal evaluation metrics. In deployment, we identified misuses such as overwriting existing ML APIs without versioning and choosing the wrong deployment endpoint. During the serving stage, common misuses included Improper Handling of API Limits and Calling the Wrong ML Cloud Service API. Finally, in the monitoring stage, we found a misuse related to ignoring monitoring data drift. Detailed resources are provided in our replication package, allowing readers to further explore the sources.

Table 2.1 Summary of results from each source after executing the search query

Source	Results from Search Query	# of Retained References
ArXiv	327	8
Official Documentation	1,740	25
Official cloud tech blogs	78	10
Medium	971	8

2.4.1.3 Results of the GitHub Projects Analysis

Our manual evaluation of GitHub projects led to the identification of 20 distinct ML service misuses, with a total of 290 occurrences. Notably, while our literature review identified a broader set of potential misuses, only those 20 misuses were actually observed in the projects we analyzed. To the best of our knowledge, this constitutes the largest catalog of ML service-based misuses compiled to date, drawing from multiple sources, including research literature, gray literature, and the analysis of open-source projects (through both automated and manual methods). This work establishes a new benchmark for research in this area, as we not only describe and categorize the misuses but also report their prevalence in real-world open-source projects. We provide a detailed report of the detected misuses for each project, along with explanations and direct links to the corresponding resources in our replication package for further reference (Ben Amor *et al.*,

2024). We should note that all the identified misuses in GitHub projects have been retained in our final catalog, as detailed in the following.

2.4.1.4 Resulting Catalog of ML Service Misuses

The research literature review and the gray literature review provided valuable insights into the practical use of ML services within software systems. By integrating these data, we were able to derive an initial set of ML service misuses, which formed a preliminary catalog of 34 misuses. Of those misuses, 14 were identified through our review of the research literature and 25 through our examination of the gray literature, with an overlap of five misuses. Then, our analysis of GitHub projects uncovered 20 of those misuses, six of them from the research literature and 14 of them from the gray literature. After applying the selection criteria (see Section 2.3.4), we retained 20 ML service misuses, which we describe in detail in Section 2.4.1.4. We should note that such a reduction from 34 to 20 misuses was further motivated by practical considerations in the design of our online survey with practitioners. Instead of incorporating all 34 misuses identified in the research and gray literature, we chose to focus on 20 misuses more commonly found in real-world GitHub projects. Including the entire set could have overwhelmed participants, potentially leading to survey fatigue and impacting response quality and completion rates. Further, to keep the survey focused and gather meaningful feedback, we selected a diverse and representative subset of the 20 most prevalent misuses, covering the different types and stages of the ML development pipeline. Furthermore, the survey was designed to be completed in less than 15 minutes, an important constraint given the limited availability of time of practitioners.

We categorized the retained ML service misuses according to the stages of the ML development pipeline and published our catalog proposal online[13]. Below, we provide a detailed description of each misuse, along with illustrative examples of some misuses derived from our analysis of GitHub repositories.

1) **Data Collection & Preprocessing Stage.**

   Misuse 1 - ***Not using batch API for data processing:*** Cloud providers offer batch processing APIs to optimize data loading performance by handling data in batches. However, developers

sometimes fail to use these batch APIs, choosing instead to load data items individually or implement their own batch processing solutions (Cao *et al.*, 2022). Not using a batch API can cause Out-Of-Memory (OOM) issues, excessive network traffic, and significant delays in data loading, slowing down model training and increasing operational costs. This misuse is applicable in contexts where batch processing is required or beneficial. More specifically, in scenarios where real-time or streaming data processing is necessary, ML cloud services often provide specialized real-time APIs, and in such cases, not using a batch API would not be considered a misuse. The code below[14] shows two occurrences of this misuse, as the Azure Text Analytics batch API was not used for language detection and sentiment analysis across multiple documents (API, 2024), leading to slower data processing. Below, we present a refactored version that uses batch processing, reducing redundant calls and network traffic.

Misuse 1: Not using Batch API for Data Processing (Example)

```
from azure.ai.textanalytics import TextAnalyticsClient
def main():
    ...
    cog_client = TextAnalyticsClient(endpoint=cog_endpoint, credential=credential)
        # Analyze each text file in the reviews folder
        for file_name in os.listdir(reviews_folder):
            # Read the file contents
            ...
            detectedLanguage = cog_client.detect_language(documents=[text])[0]
            ...
            sentimentAnalysis = cog_client.analyze_sentiment(documents=[text])[0]
            print("\nSentiment: {}".format(sentimentAnalysis.sentiment))
        ...
```

[14] https://GitHub.com/ovokpus/Python-Azure-AI-REST-APIs/blob/main/text-analytics-sdk/text-analysis.py

Misuse 1: Not using Batch API for Data Processing (Recommendation)

```
from azure.ai.textanalytics import TextAnalyticsClient
def main():
    ...
    # Load documents directly from files for batch calls
    documents = [open(os.path.join(reviews_folder, file_name), encoding='utf8').read()
        for file_name in os.listdir(reviews_folder)]
    # Batch process for language and sentiment detection
    detected_languages = cog_client.detect_language(documents=documents)
    sentiments = cog_client.analyze_sentiment(documents=documents)
    ...
```

Misuse 2 - ***Inefficient data transmission:*** Refers to inefficient data transmission between components within an ML service-based system, such as storage services, compute nodes, and other cloud resources. This inefficiency leads to increased latency and higher cloud usage costs (Cao *et al.*, 2022). For example, the repeated transfer of training data from cloud storage to compute nodes for each job, instead of caching it locally, can cause significant delays in training time, increased network traffic, and higher data transfer costs.

2) **Training Stage.**

Misuse 3 - ***Non specification of early stopping criteria:*** ML services typically offer options to set early stopping criteria, helping to avoid overfitting and unnecessary computational expenses (Hyperparameters, 2024). However, developers sometimes fail to specify these criteria, which allows the training to proceed for more epochs than necessary. This oversight can result in wasted computational resources, increased training duration, higher costs, and potential overfitting (Ying, 2019).

Misuse 4 - ***Avoiding parallel training experiments:*** Cloud providers offer the capability to run parallel training experiments to accelerate the model training process and enhance the efficiency of ML service-based systems (ML, 2024b). Avoiding parallel experiments can slow down model development and prevent the full utilization of available computational resources. In the following example[15], the Azure `ScriptRunConfig` class is used without specifying a distribution configuration, which effectively disables parallel training, resulting in a less efficient process. To improve efficiency, it is recommended to either (1) include a

[15] https://GitHub.com/praneet22/DevOpsForAI/blob/master/aml_service/10-TrainOnLocal.py

distribution configuration within `ScriptRunConfig` (as shown in the listing below) or (2) use Azure `DistributedRunConfig`, which automatically distributes computation across multiple nodes, thereby enabling parallel training experiments and significantly accelerating the training process (ScriptRunConfig, 2024).

Misuse 4: Avoiding Parallel Training Experiments (Example)

```
from azureml.core import Experiment
from azureml.core import ScriptRunConfig
...
src = ScriptRunConfig(
source_directory="./code",
script="training/train.py",
run_config=run_config_user_managed,)
run = exp.submit(src)
```

Misuse 4: Avoiding Parallel Training Experiments (Recommendation)

```
from azureml.core import Experiment
from azureml.core import ScriptRunConfig
...
# Create a distributed run configuration
distributed_run_config = DistributedRunConfig(
    node_count=2,  # Number of nodes
    virtual_environment='env1',
    instance_type='Standard_NC6',  #VM size)
src = ScriptRunConfig(source_directory="./code",
script="training/train.py",
run_config=run_config_user_managed,
distributed_job_config=distributed_run_config)
```

Misuse 5 - ***Not using automatic hyperparameter tuning:*** ML cloud providers offer the capability to define the search space and automatically optimize ML model hyperparameters (Hyperparameters, 2024). However, developers might not leverage this feature, choosing instead to manually set hyperparameters, which can lead to suboptimal model performance and increased training time (Victoria & Maragatham, 2021; Weerts, Mueller & Vanschoren, 2020).

Misuse 6 - ***Not using training checkpoints:*** Cloud providers offer the ability to resume training from the most recent checkpoint, saving the current state of the experiment rather

than starting from scratch. This can save significant time and computational resources, especially when training large and complex models (Xu, Liu, Tao, Xuan & Zhang, 2022; Rojas, Kahira, Meneses, Gomez & Badia, 2020). However, developers may neglect to save training checkpoints in cloud storage. If a model fails and checkpoints have not been saved, the entire training job or pipeline will terminate, resulting in a loss of data since the model's state is not preserved in memory (Practices, 2024b). In the following example[16], the training process in Google Vertex AI commences without implementing checkpoints to save the model's state. To prevent potential data loss and improve efficiency, a `save_checkpoint` function can be defined (as shown in the recommendation listing) to store the model's state in Google Cloud Storage during training.

Misuse 6: Not Using Training Checkpoints (Example)

```
from detectors.vertex_ai_job import Trainer
...
 # Run Trainer
    if args.train:
        print('Initialising training')
        trainer = Trainer(config=config)
        trainer.train()
        Trainer.clean_up()
```

[16] https://GitHub.com/Subrahmanyajoshi/Cancer-Detection-using-GCP/blob/master/detectors/detector.py

Misuse 6: Not Using Training Checkpoints (Recommendation)

```
from detectors.vertex_ai_job import Trainer
from google.cloud import storage
...
#Save checkpoints in Google Cloud storage
def save_checkpoint(checkpoint_path)
    storage_client = storage.Client()
    # Define a Google Cloud Storage container
    bucket = storage_client.bucket("MyBucket")
    # Create a blob from the bucket
    blob = bucket.blob(checkpoint_path.split("/"))
    # Upload the file to Google Cloud Storage
    blob.upload_from_filename(checkpoint_path)
    print(f"Checkpoint saved to {blob.name}")
...
# Run Trainer
    if args.train:
        print('Initialising training')
        trainer = Trainer(config=config)
        trainer.train()
        trainer.save_checkpoint("path/to/save")
        Trainer.clean_up()
```

Misuse 7 - ***Bad choice of training compute targets:*** Refers to selecting non-optimal hardware and compute resources for training ML models. Cloud providers offer various types of training compute targets, yet not all resources are adequate for every ML task. For example, Azure Machine Learning Kubernetes is ideal for ML pipelines and Azure ML Designer, which require robust support for parallel processing and resource allocation. However, it is not recommended for automated ML, which requires different computing capabilities (Targets, 2024). Choosing the right compute target ensures optimal performance and cost-effectiveness for the specific training task.

Misuse 8 - ***Excluding algorithms in automated ML:*** Cloud providers offer automated machine learning services that perform experiments with various ML algorithms to generate an optimized model for a given prediction task ready for deployment (Microsoft, 2024b). However, when configuring these services, developers may mistakenly exclude promising algorithms, thereby limiting the model's effectiveness and performance.

3) **Testing Stage.**

MISUSE 9 - *Misinterpreting output:* ML cloud services offer pre-built models that operate on high-dimensional continuous representations but often produce a small discrete set of outputs. Consequently, the output of these services may contain complex and easily misinterpretable semantics, potentially leading to bugs (Wan *et al.*, 2021). For example, Google's NLP Sentiment Detection API returns two metrics: score and magnitude, which together assess the sentiment of a text input. Developers may misinterpret the sentiment if they do not correctly combine both metrics. The score reflects the polarity of the sentiment (positive or negative), while the magnitude indicates the strength of the sentiment (Wan *et al.*, 2021). The code example below, from a journal application[17], judges the sentiment in a user's journal. However, it only considers the score value, potentially marking the journal as emotionally negative incorrectly. As mentioned in Google's service documentation, it is recommended to extend the sentiment assessment with a rule-based evaluation using both score and magnitude to predict the final sentiment, as shown below.

MISUSE 9: Misinterpreting Output (Example)

```
from google.cloud import language_v1
...
response=client.analyze_sentiment(document^document,
encoding_type=encoding_type)
...
sentiment=response.document_sentiment.score
...
if avg_sentiment < 0:
    message= 'Your posts show that you might not be
    going through the best of time.'
        elif avg_sentiment<0.5:
                message = 'Your posts show that you are having some decent times! Here s
                    to many more happy days'
```

[17] https://GitHub.com/beekarthik-ai/JournalBot/blob/master/webapp/app_folder/routes.py

Misuse 9: Misinterpreting Output (Recommendation)

```
from google.cloud import language_v1

# Assuming ‘client‘, ‘document‘, and ‘encoding_type‘ are already defined
response = client.analyze_sentiment(document=document, encoding_type=encoding_type)

score = response.document_sentiment.score
magnitude = response.document_sentiment.magnitude

if abs(score) < 0.15 or abs(magnitude) < 0.15:
    message = "Neutral"
elif score > 0:
    message = "Here's to many more happy days"
else:
    message = "You might not be going through the best of times."

print(message)
```

Misuse 10 - *Ignoring fairness evaluation:* ML cloud providers offer mechanisms for fairness evaluation, which is important to ensure unbiased and equitable models (Mehrabi, Morstatter, Saxena, Lerman & Galstyan, 2021; Antunes *et al.*, 2018). Neglecting fairness evaluation can result in predictions that have potential biases and unfair towards specific demographic groups (Framework, 2024), which could negatively impact model's generalizability and performance. While fairness evaluation is most directly applicable when protected attributes are present in the data, it is a recommended best practice even when such attributes are not explicitly available, as it can help detect indirect biases and ensure equitable prediction across different groups (Ashurst & Weller, 2023; Bothmann, Peters & Bischl, 2022). Fairness issues can also arise from complex feature interactions or evolving user populations. Therefore, integrating fairness assessments as a continuous part of the ML lifecycle helps identify and mitigate unintended biases, contributing to more robust and responsible ML service deployments.

Misuse 11 - *Ignoring testing schema mismatch:* Cloud providers offer ML services to detect unmatched data schemas, which include feature or data distribution mismatches between training, testing, and production data, often by raising alerts. However, developers may ignore setting up these alerts or disable them. For example, Amazon ML displays

alerts if the schemas for the training and evaluation data sources are not consistent (Amazon Web Services, 2024b). Disabling these alerts can result in missing discrepancies, such as features present in the training data but absent in the evaluation data, or detecting unexpected additional features. This oversight may weaken the model's accuracy and performance in the production environment (Polyzotis, Zinkevich, Roy, Breck & Whang, 2019).

Misuse 12 - ***Using suboptimal evaluation metrics:*** Some ML services optimize and evaluate models based on specified evaluation metrics. These metrics determine how model performance is measured during training and evaluated during testing (Naser & Alavi, 2020). However, developers sometimes choose suboptimal evaluation metrics, leading to less effective models that do not align well with business needs or dataset characteristics. For example, according to Microsoft ML's official documentation (Microsoft, 2024c), in classification tasks, metrics such as accuracy and `norm_macro_recall` may not perform well with small datasets, datasets with significant class imbalance, or when metric values are close to zero or one. In such cases, `AUC_weighted` could be a better option (Microsoft, 2024c).

4) **Deployment Stage.**

Misuse 13 - ***Overwriting existing ML APIs without versioning:*** Cloud providers offer ML API versioning through tools such as Azure API Management and AWS Management Console. Without version control, it becomes challenging to track changes, revert to previous versions, or understand the evolution of the deployed model. Developers may, however, overlook these tools and unintentionally overwrite existing ML APIs without proper versioning, which can lead to issues in the production environment. As discussed in a Stack Overflow post (Overflow, 2024), when publishing an updated model using Azure ML Studio, overwriting an existing API without versioning can make it difficult to roll back to a previous stable version if the new model introduces unexpected behavior or lower accuracy, potentially causing disruptions in the production system.

Misuse 14 - ***Choosing the wrong deployment endpoint:*** Cloud providers offer online and batch endpoints for deployment. Online endpoints are mainly used to operationalize models for real-time inference in synchronous, low-latency requests. In contrast, batch endpoints

are used primarily to operationalize models or pipelines for long-term, asynchronous inference (Endpoints, 2024a). However, developers may select the inappropriate endpoint for deployment. For example, using a batch endpoint for real-time tasks can delay inferences since batch processing is not optimized for speed.

Misuse 15 - ***Disabling automatic rollbacks:*** ML cloud service providers offer features that automatically rollback to a previously stable version of a model if errors or performance issues arise with a newly deployed version. However, developers may disable this feature, allowing poorly performing models to remain in production, which can negatively impact system performance (Serban, Van der Blom, Hoos & Visser, 2020).

Misuse 16 - ***Disabling automatic scaling for online prediction service:*** Automatic scaling allows resources to be dynamically adjusted based on demand to maintain sufficient capacity for online prediction services. This feature helps manage varying prediction request rates while minimizing cloud usage costs (Google Cloud, 2024). However, developers may disable automatic scaling when deploying ML models, potentially increasing inference latency during peak demand periods. The example below[18] highlights this issue since automatic scaling is disabled by default in Amazon SageMaker. As a result, developers need to explicitly configure auto-scaling policies. It is recommended to use SageMaker's "*AutoScalingPolicy*" to automatically adjust resources according to demand, ensuring optimal performance during traffic spikes and minimizing costs during low usage periods (as shown in the recommendation snippet) (Amazon Web Services, 2024a; Endpoints, 2024b).

Misuse 16: Disabling automatic scaling for online prediction service (Example)

```
import sagemaker
...
endpoint_config_response = client.create_endpoint_config(
   EndpointConfigName=epc_name,
   ProductionVariants=[ {"VariantName": "sklearnvariant",
       "ModelName": model_name,
       "InstanceType": "ml.c5.large",
       "InitialInstanceCount": 1 },],)
```

Misuse 16: Disabling automatic scaling for online prediction service (Recommendation)

[18] https://GitHub.com/Ewen2015/BatCat/blob/master/batcat/SageMaker.py

```
import sagemaker
...
endpoint_config_response = client.create_endpoint_config(
EndpointConfigName=epc_name,
ProductionVariants = [{"VariantName": "sklearnvariant",
    "ModelName": model_name,
    "InstanceType": "ml.c5.large",
    "InitialInstanceCount": 1,
    "AutoScalingPolicy": {
    "ScaleInCooldown": 60, # waiting time before next scale
    "MinCount": 1, # Minimum number of instances
    "MaxCount": 3 }}]
```

5) **Serving Stage.**

Misuse 17 - *Improper handling of ML API limits:* Failure to adhere to API request rate limits can compromise the stability and performance of the ML service (OpenAI, 2024). Developers may not adequately manage these limits, causing predictions to abruptly halt when the rate is exceeded. For instance, surpassing the maximum number of API calls within a set timeframe, such as requests per second defined by the Azure OpenAI service (Quotas & limits, 2024), can result in delayed or rejected requests until they conform to the permitted rate.

Misuse 18 - *Misusing Synchronous/Asynchronous ML APIs:* ML cloud providers typically offer both synchronous and asynchronous versions of the same tasks. Asynchronous services are designed to handle larger inputs and longer processing times, as opposed to synchronous services. However, calling asynchronous ML services in a synchronous manner can degrade system performance significantly, as it increases latency due to blocked service invocations (Wan *et al.*, 2021).

Misuse 19 - *Calling the wrong ML service API:* Cloud providers often offer multiple ML APIs for similar tasks. Without a thorough understanding of these APIs, developers might use the wrong one, leading to significantly reduced prediction accuracy, incorrect prediction results, or software failures. For example, image classification and object detection are vision APIs that provide description tags for input images. The former offers a single tag for the entire image, while the latter provides a tag for each object. Using image classification

instead of object detection can cause software to miss important objects, and using object detection instead of image classification can result in incorrect image tags (Wan *et al.*, 2021).

6) **Monitoring Stage.**

Misuse 20 - ***Ignoring monitoring for data drift:*** This refers to neglecting the continuous assessment of changes in statistical characteristics or data distributions, which is crucial for maintaining model performance (Bova *et al.*, 2017). Data drift occurs when the incoming data distribution differs from the training data, leading to degraded model accuracy over time (Mallick, Hsieh, Arzani & Joshi, 2022). Cloud providers recommend implementing skew and drift detection mechanisms to monitor these changes and alert developers when significant changes occur (Practices, 2024a). By detecting data drift early, models can be retrained or adjusted to ensure they continue performing as expected in production environments.

2.4.2 RQ2. What is the state of the practice of ML cloud service misuses in industry?

In this section, we present the results of our practitioner survey, with data collected from responses between July and December 2024.

2.4.2.1 Survey Participants:

Our survey reached a diverse group of 50 practitioners who actively participated in ML service-based projects, representing a wide range of roles within the field. Specifically, as shown in Figure 2.5, 34% of the participants identified as ML engineers, another 34% as data scientists, and 16% as software engineers. The remaining 14% of the participants held various other roles, including data engineers (6%), software architects (6%), and cloud engineers (4%), highlighting the breadth of expertise and perspectives involved in the survey. In terms of experience, our participants also reflected considerable diversity. The majority reported having between one and five years of experience across various fields: AI (56%), software development (52%), and cloud computing (42%). Moreover, 42% had less than one year of experience in cloud computing, while 16% of the practitioners had less than one year of experience in both AI and

software development. This variety in experience levels further underscores the diversity of the participants, encompassing both emerging professionals and those with more established expertise in ML service-based systems.

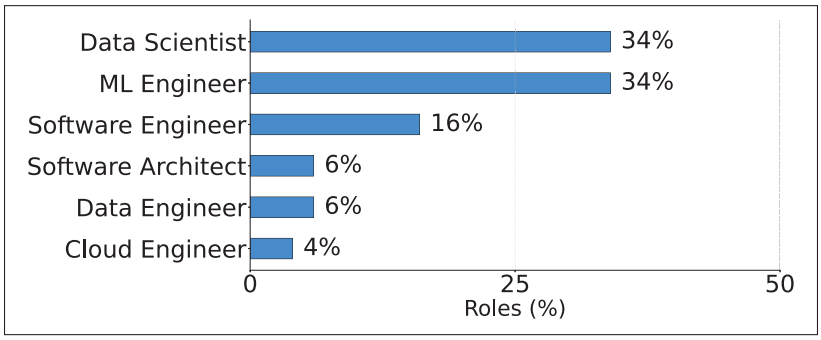


Figure 2.5 Roles of Survey Participants

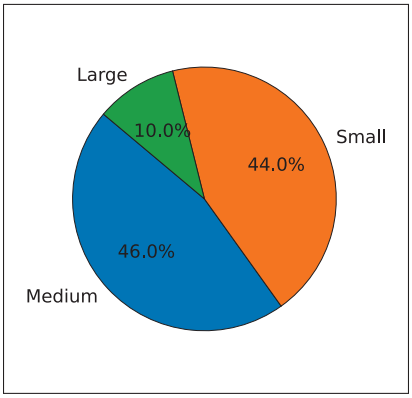


Figure 2.6 Projects Size

2.4.2.2 Size of Projects:

We asked practitioners regarding the scale of ML service-based projects they had participated in. We found that more than half (52%) had worked on more than five such projects, while 48% had worked on one to five projects. Among these projects, as shown in Figure 2.6, 44% were considered small ($\leq 5,000$ LOC), 46% medium-sized, and only 10% large ($\geq 30,000$ LOC). We also asked practitioners about the frequency with which they used ML services. Figure 2.7 indicates that these services are commonly used in the industry, with 52% of respondents reporting usage Very Often (daily), Often (weekly), or Sometimes (monthly), while only 16% reported using them Rarely (once a year or less). This frequent use highlights the need for best practices. We further asked practitioners about which services are used most often, and found that Azure ML services, Amazon ML services, and Google ML services are among the top. Surprisingly, only a small proportion of practitioners indicated the use of IBM Watson (10%) and Oracle or others (6%). Practitioners indicated that they interact with ML services using various methods: 70% use APIs, 66% use Command Line Interface (CLI) and User Interfaces

(UI), and 62% use SDKs, as shown in Figure 2.8. A preference for APIs is noted due to their integration ease, while the high usage of CLIs and UIs suggests a balance between technical and non-technical engagement.

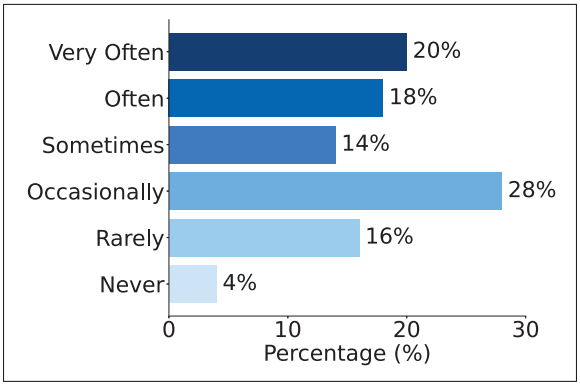


Figure 2.7 Frequency of ML Service Utilization

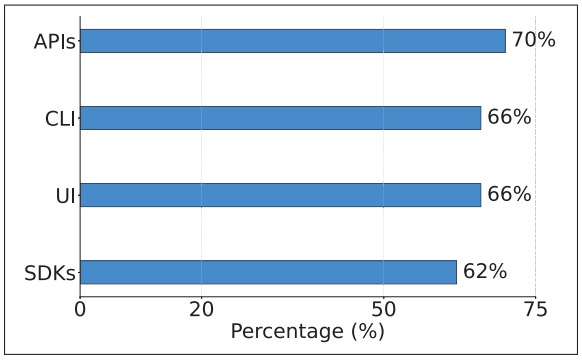


Figure 2.8 Methods of ML Service Utilization

We should note that the diversity in project sizes based on ML services, the degree of ML services adoption, interaction methods, and different cloud providers all enhance the generalizability of our findings, despite the survey's number of participants.

2.4.2.3 ML Services Impact and Challenges:

Practitioners highlighted both positive and negative impacts of ML services on their project development process. On the positive side, 82% of practitioners reported that adopting ML services in their industrial projects helped them save development time. In addition, 68% found that their projects were more scalable when using ML services, and 56% noted an increase in the efficiency of the software development process, as noted in Figure 2.9. These findings underscore the critical role of ML services in improving system scalability and efficiency while reducing the time required for development. Practitioners also identified several challenges associated with ML services, which we report in Figure 2.10. More than half (54%) reported integration difficulties, indicating that ML services can complicate the incorporation of new components into existing systems. The resource management issues were highlighted by 46% of

the practitioners, who experienced problems with memory and computational resource allocation. Security concerns were also prevalent, with 42% of the participants expressing concerns about data security and privacy. These challenges emphasize the need for effective practices to mitigate the negative impacts of ML services. In addition, 77% of practitioners faced challenges at various stages of their projects. As shown in Figure 2.11, specific difficulties included data collection and processing (42%), training (38%), testing (35%), deployment (50%), serving (38%), and monitoring (40%). These findings show that the widespread challenges related to ML services can contribute to or result in their misuse, underscoring the need for comprehensive strategies to mitigate such risks.

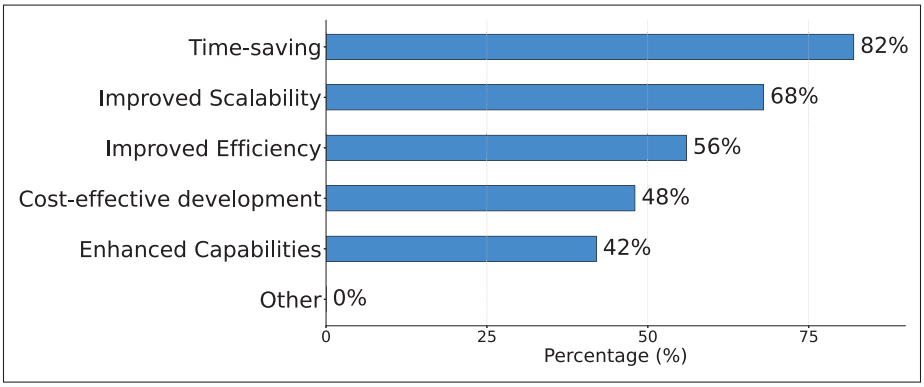


Figure 2.9 Impact of ML Services

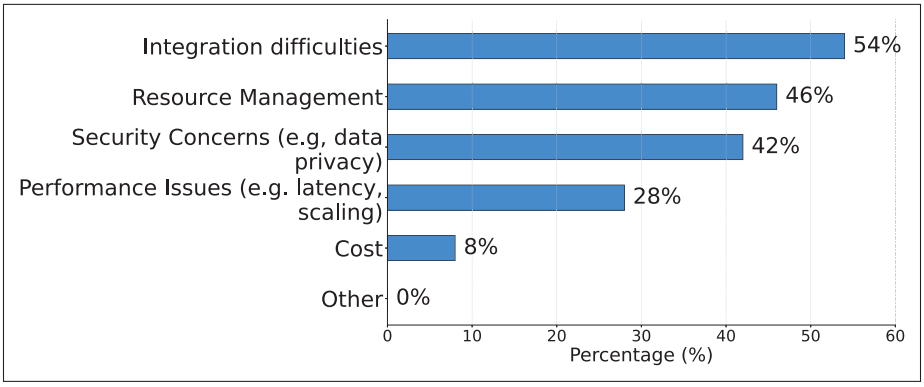


Figure 2.10 ML Service-Related Challenges

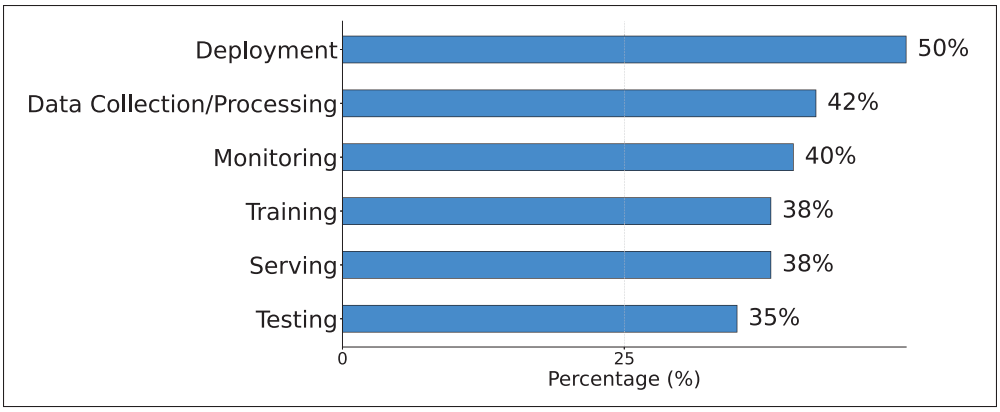


Figure 2.11 Frequency of ML service challenges across different stages of the ML development pipeline

Takeaway 1: It is important to address challenges in integration, resource management, and security to fully unlock the benefits of ML services. Doing so not only ensures project success but also enhances efficiency, scalability, and development capabilities.

2.4.2.4 Catalog Agreement Analysis:

We asked practitioners about the extent to which they agree or disagree on whether the practices in our catalog represent misuses of ML services. Figure 2.12 illustrates the level of agreement for each identified misuse. We found that all practices were considered misuses by practitioners, with agreement levels ranging from 52% to 80%. The highest consensus (80%) was on misuses such as "*Choosing the wrong deployment endpoint*", "*Overwriting existing ML APIs without versioning*", "*Ignoring fairness evaluation*" and "*Misinterpreting output*". In contrast, the lowest consensus (52%) was for "*Not using Batch API for data processing*". This lower score may reflect that some practitioners did not deem this issue to be a misuse, particularly in scenarios where small datasets make batch processing unnecessary. Furthermore, some practitioners were neutral, potentially due to varying contexts and use cases influencing their perceptions of these practices as misuses.

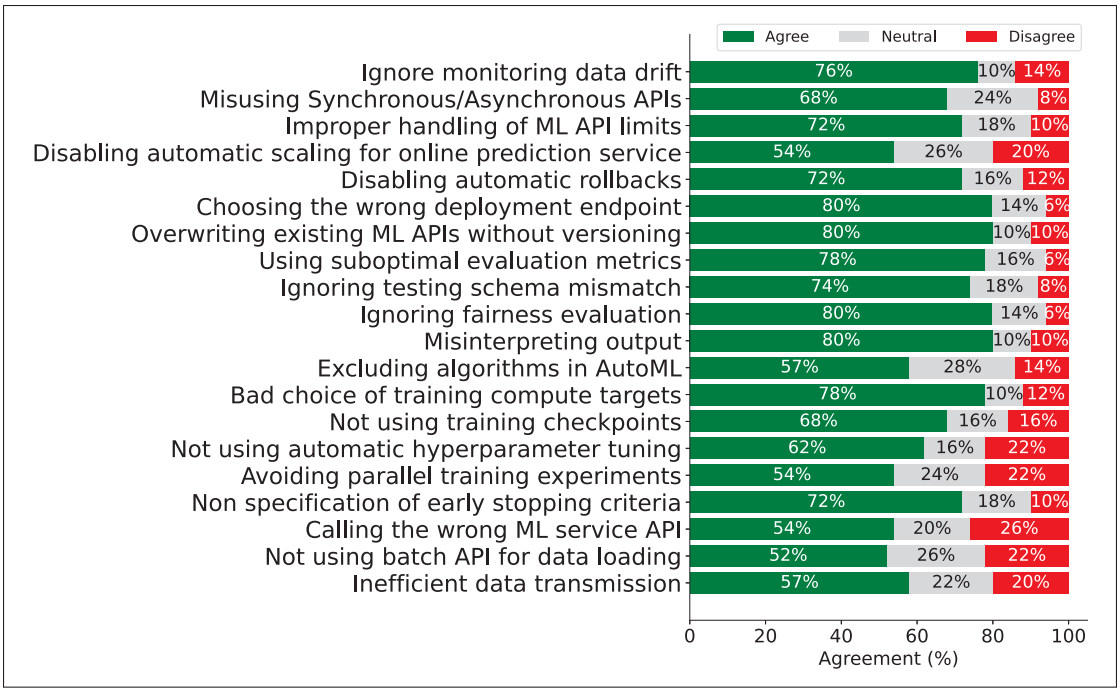


Figure 2.12 ML Service Misuses Agreement Level

2.4.2.5 Correlation Between Experience Levels and Agreement on ML Service Misuses:

This analysis aimed at understanding whether awareness of ML service misuses correlates with practitioner expertise. This is particularly crucial because if misuse awareness is primarily lacking among less experienced practitioners, then targeted training and best-practice interventions should focus on this group. In contrast, if the correlation is weak or absent, then awareness-building efforts may need to be directed throughout the practitioner population. Our results on the correlation between participants' experience levels in AI development and the number of ML service misuses they agreed on revealed a moderate positive and statistically significant correlation (48%) between experience levels and agreement on ML service misuses. In addition, we analyzed the correlation between the experience level of participants and their agreement level with each misuse. In 15 out of 20 misuses, we found positive and statistically significant correlations ranging from 30% to 47%. All correlations are available in our replication package (Ben Amor *et al.*, 2024). The positive correlation indicates that as developers gain more experience in AI development, they are more likely to agree on potential misuses in ML services,

enabling them to recognize misuse scenarios more effectively. These findings emphasize the critical role of practitioner expertise in identifying misuse scenarios. While experienced practitioners are generally more capable of detecting and recognizing potential misuses, less experienced practitioners may benefit from targeted training and increased awareness of best and worst practices in ML service usage. However, the lack of strong correlation between expertise and certain misuses (e.g., "*Inefficient data transmission*", "*Avoiding parallel training experiments*", "*Using suboptimal evaluation metrics*") suggests that some issues are less intuitive and thus require attention regardless of experience level.

There is strong consensus on the importance of our ML service misuse catalog, with most practitioners acknowledging its significant impact. More experienced professionals, in particular, show greater agreement, reflecting widespread awareness of industry challenges and the need for targeted solutions.

2.4.2.6 ML Service Misuses Frequency:

We asked practitioners how frequently they encounter or introduce each of the ML service misuses listed in our catalog. Specifically, for each misuse, practitioners were asked to indicate whether they have encountered the misuse in most ML projects (i.e., more than 75% of projects), in some ML projects (i.e., less than 75% of projects), or never encountered the misuse. To ensure response reliability, we only retained answers where practitioners agreed on the presence of a misuse. As illustrated in Figure 2.13, our findings reveal that ML service misuses are prevalent in the industry, with only 10% to 32% of practitioners reporting that they had never encountered such misuses in their projects. Notably, "*Improper handling of ML API limits*" was the most frequently observed misuse, with 31% of participants indicating its presence in most of their projects. This may be attributed to inadequate documentation from cloud providers on configuring appropriate API limits. In addition, "*Inefficient data transmission*" was the most frequently encountered misuse in some projects, reported by 70% of practitioners. This likely reflects common challenges such as the lack of caching mechanisms to avoid redundant

data transfers. Similarly, "*Not using Batch API for data processing*" was reported in some projects by 68% of respondents, indicating limited awareness of the benefits of batch processing and a tendency toward less efficient data processing practices. On the other end, "*Excluding algorithms in automated ML*" had the highest percentage of never being encountered (32%), followed by "*Calling the wrong ML service API*" (26%). These findings suggest that such misuses are relatively less common in practice.

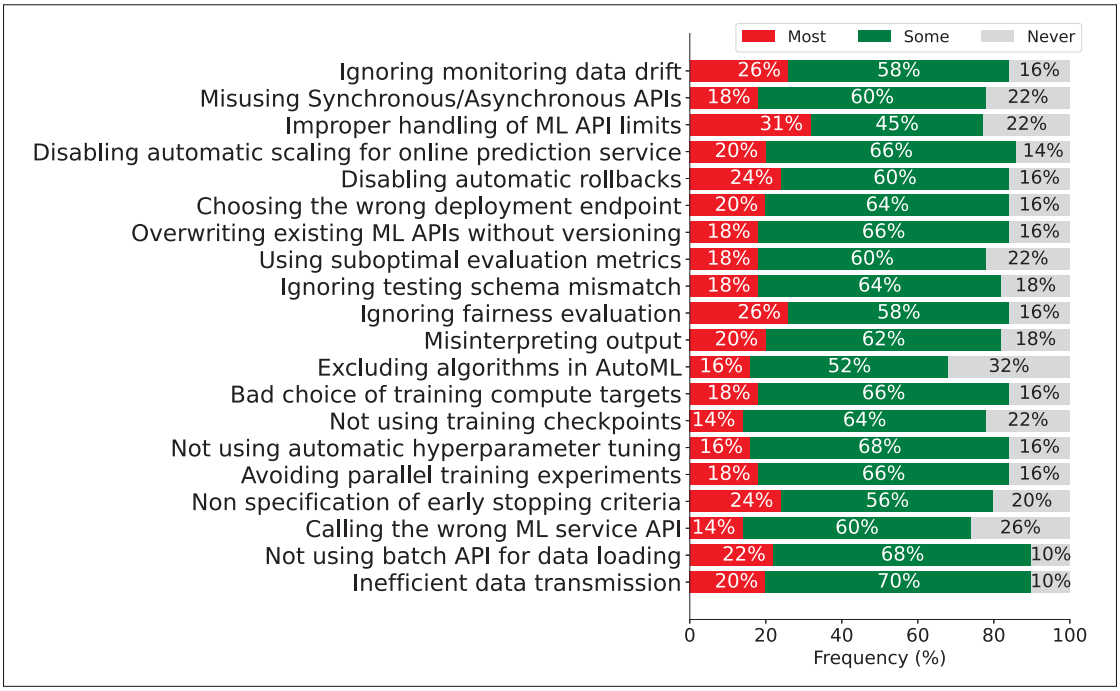


Figure 2.13 ML Service Misuses Frequency

Takeaway 3: Misuses of ML services are prevalent throughout project development, with improper handling of ML API limits being the most common issue, often causing performance bottlenecks and service interruptions, which makes addressing it crucial for smooth operations.

2.4.2.7 Correlation Between Experience Levels and Frequency of ML Service Misuses:

In nine of 20 misuses, we observed positive and statistically significant correlations (ranging from 29% to 36%) between participants' experience levels and the frequency of encountering each ML service misuse. This suggests that more experienced practitioners are better at identifying and recognizing these issues. A possible explanation is that seasoned professionals have a deeper understanding of ML service-based systems, are more familiar with common pitfalls, or are exposed to more complex projects where such misuses are more likely to occur.

2.4.2.8 Reasons of ML Service Misuses:

We asked practitioners about the possible reasons behind the presence of ML service misuses in practice. The survey options for possible reasons were derived from our literature review (Nahar, Zhou & Lewis,G.and Kästner, 2022; Nahar, Zhang, Lewis, Zhou & Kästner, 2023; Bogner *et al.*, 2021), and an *Other* option was included to allow participants to provide additional open-ended input. As shown in Figure 2.14, the most prominent reasons were a lack of understanding of ML service capabilities (58%) and a lack of documentation (50%). These findings point to an urgent need for improved educational resources and more comprehensive documentation from ML cloud service providers. Moreover, half of the practitioners identified a lack of awareness or understanding of best practices for using ML cloud services, and 44% highlighted poor code review processes as key contributors to ML service misuse. This underscores the importance of disseminating best practices and establishing robust code review mechanisms. Other factors included pressure to meet tight deadlines (34%) and the rapid evolution of ML cloud services, which makes it challenging to stay up to date with the latest advancements (30%). Notably, no additional reasons were provided under the *Other* category, suggesting the predefined options offered broad coverage of the relevant reasons.

Furthermore, several of the identified factors were echoed in software engineering blogs and online developer discussions. In these sources, practitioners reported numerous practical challenges when working with ML cloud services (e.g., lack of understanding of ML service

capabilities), which can often contribute to the introduction of ML service misuses. As a result, without sufficient guidance, developers may make incorrect assumptions or adopt suboptimal configurations. For instance, a Stack Overflow thread highlighted that users frequently encounter "*TooManyRequests error*" when interacting with Azure ML REST APIs (Stack Overflow Community, 2022). This issue stems from unclear documentation regarding service request limits, forcing developers to implement their own rate-limiting strategies or rely on trial-and-error: "*I could not find out how many concurrent requests are allowed by Azure Machine Learning batch endpoints, so I ended with a limit of 10 outgoing requests which solved the "TooManyRequests" problem.*" (Stack Overflow Community, 2022). This example illustrates how the absence of clear documentation directly contributes to misuse or inefficient use of ML services. Similarly, a software quality engineer reported on Medium that "*excessive time spent configuring build tools, dependency management, and deployment processes slows the development of ML service-based systems*" (Sharma, 2024). Such experiences help explain why practitioners often feel compelled to adopt ad-hoc solutions, rush deployments, or rely on trial-and-error approaches, mirroring our survey results where tight deadlines (34%) and rapid evolution of services (30%) were cited as reasons for misusing ML services. Also, (Kästner & Goues, 2025) observed that "*organizations often focus on short-term development goals and product success measures, whereas many fairness concerns relate to long-term outcomes, such as feedback loops, and avoiding rare disasters*". This supports the notion that economic and organizational pressures can drive technical decisions that compromise long-term quality, fairness, or responsible ML service usage.

These insights suggest that addressing ML service misuses requires a multi-faceted approach, including improved documentation, better education on ML capabilities, dissemination of best practices, and enhanced access to resources and expertise.

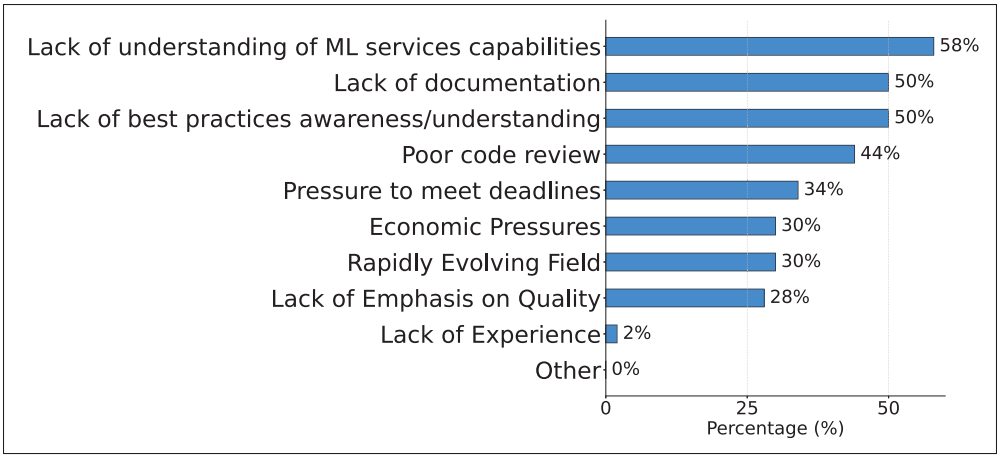


Figure 2.14 Reported Reasons of ML Service Misuses

Takeaway 4: ML service misuse mainly stems from a lack of understanding of their capabilities, insufficient documentation, and limited awareness of best practices, which could lead to inefficiencies and poor performance, thus highlighting the need for better guidance and awareness among practitioners.

2.4.2.9 Mitigation of ML Service Misuses:

We asked the practitioners for mitigation solutions to avoid introducing ML service misuses. As shown in Figure 2.15, the most commonly recommended approach was staying up to date with best practices for ML services and cloud provider standards, as mentioned by 66% of survey participants. This ensures that practitioners are aware of the latest guidelines, tools, and techniques, thereby reducing the likelihood of introducing potential misuses. In addition, 64% of practitioners emphasized the importance of thoroughly validating the quality of data used by ML components, while 58% highlighted the need for regular reviews of model and system performance. Other frequently recommended practices include implementing robust alerting systems for model monitoring (48%), collaborating effectively with domain experts and stakeholders (48%), and actively participating in peer code reviews (48%).

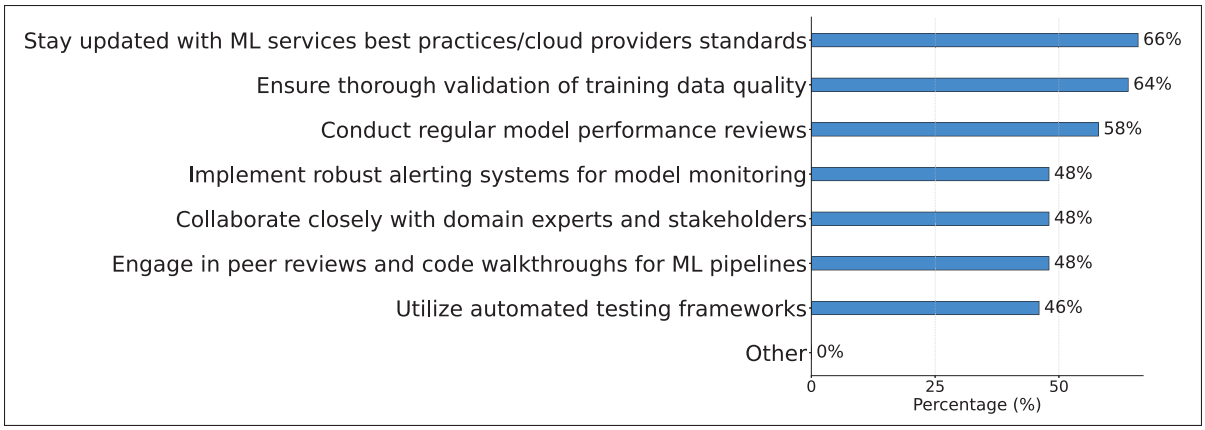


Figure 2.15 ML Service Misuses Mitigation

These strategies are also discussed in software engineering blogs and developer forums. For example, an ML engineer emphasized in a Medium blog the importance of "*implementing a robust data monitoring system to track data quality and distribution over time, and setting up alerts or triggers to notify when significant drift is detected*" (Sohail, 2025). This aligns with the mitigation strategies identified in our survey, particularly conducting regular model performance reviews and implementing robust alerting systems for model monitoring. Furthermore, the same engineer emphasizes the importance of (1) "*carefully preprocessing and cleaning data before training and deployment*" and (2) "*properly handling missing values, outliers, and inconsistencies to minimize the risk of data drift*". These practices reinforce our survey's recommendation to ensure thorough validation of training data quality. Moreover, a blog published by an AI-based knowledge management and automation company emphasized the importance of regular monitoring, stating: "*(Step 3) Regular Monitoring–Continuously monitor fairness metrics as part of your model maintenance routine; Regular checks can help identify and address emerging biases over time*" (Zdrok, 2024). This aligns with the mitigation strategy identified in our survey, which emphasizes the need for regular model performance reviews to proactively detect and address potential issues. Automated testing practices also support regular performance reviews. For example, a blog published by an AI-based testing solutions company highly recommends the use of Azure DevOps since it "*facilitates lower manual error rates and faster and more reliable application delivery by integrating well-known test automation tools*

such as Selenium". They also reported that it "*allows for improved test coverage and early bug detection*" (Testing, 2023). Broader reflections on documentation highlight its strategic importance: "*A significant portion of a software project's lifecycle (over 70%) is consumed by maintenance*" which "*underscores the essential role of documentation in facilitating effective software maintenance*", reported by an AI tech lead (Islam, 2024). Developers also note that successful ML projects require strategic alignment across the organization, beyond technical execution. This supports our survey finding on the importance of close collaboration with domain experts and stakeholders. As noted in a Medium post, "*building models is just one aspect of the process; it is crucial to involve feedback from users and cross-functional teams, incorporate new feature requests, and update initial hypotheses based on changing business needs.*" (Stahl, 3). Also, "*to ensure the success of an AI project, it is crucial to have regular reviews and feedback sessions with stakeholders, including product teams and business leaders*" (Stahl, 3).

However, the practical implementation of these mitigation strategies in real-world ML workflows presents significant challenges. Staying current with best practices and cloud provider updates requires continuous effort, especially in rapidly evolving ecosystems. Likewise, validating data quality and conducting regular model reviews can be resource-intensive, demanding both organizational commitment and skilled personnel (Lwakatare, Rånge, Crnkovic & Bosch, 2021; Chen, 2022; Zhang *et al.*, 2023). While automated testing and AI-assisted code review may alleviate some of this burden, their scope remains limited, and they require careful integration within existing CI/CD pipelines. To improve their effectiveness, there is a pressing need for dedicated tools that can automatically detect and refactor ML service misuses. Such tools would embed best practices at scale, providing actionable insights, enforce consistency, and reduce human error, thereby making mitigation strategies more practical and sustainable in production environments.

Takeaway 5: To effectively mitigate ML service misuse, ML engineers should: (1) continually align with the latest best practices and standards from ML services and cloud providers, (2) rigorously validate ML data to ensure quality, and (3) systematically perform regular performance evaluations to detect issues early.

2.4.3 RQ3. How comparable is the prevalence of ML service misuses in open-source projects to what is observed in industry practice?

To address this research question, we studied the prevalence of the 20 ML service misuses in GitHub projects compared to their frequency in industrial projects as reported by the survey practitioners (Figure 2.13). Figure 2.16 illustrates the prevalence of these misuses, along with the distribution of their occurrences across GitHub projects. As shown in Figure 2.16, "*Ignoring fairness evaluation*" is the most frequent misuse, appearing in over 60 projects, followed closely by "*Ignoring monitoring for data drift*" and "*Inefficient data transmission*", which were found in approximately 50 and 40 projects, respectively. Other common misuses include "*Not using batch APIs for data processing*" and "*Improper handling of ML API limits*", indicating that developers often struggle with data processing efficiency and API constraints. The figure also highlights less frequent but still notable misuses, such as "*Misusing synchronous/asynchronous APIs*", *Avoiding parallel training experiments*", and "*Not using automatic hyperparameter tuning*", each found in fewer than 10 projects. Rare misuses, including "*Disabling automatic rollbacks*", "*Choosing the wrong deployment endpoint*", and "*Excluding algorithms in AutoML*", occur in even fewer projects. Overall, the distribution suggests that, while some misuses are widespread and represent common problems in ML service development, others are rare but can still significantly impact model performance, reliability, and/or maintainability.

The Spearman's correlation between the frequency of encountering ML service misuses in most industrial projects and their occurrence in GitHub projects is 0.46 (*p-value* = 0.04), indicating a moderate, positive, statistically significant correlation, confirming that this relationship is unlikely to be due to random chance. Specifically, this suggests that the distribution of ML service misuses identified in our manual analysis of GitHub projects closely mirrors those

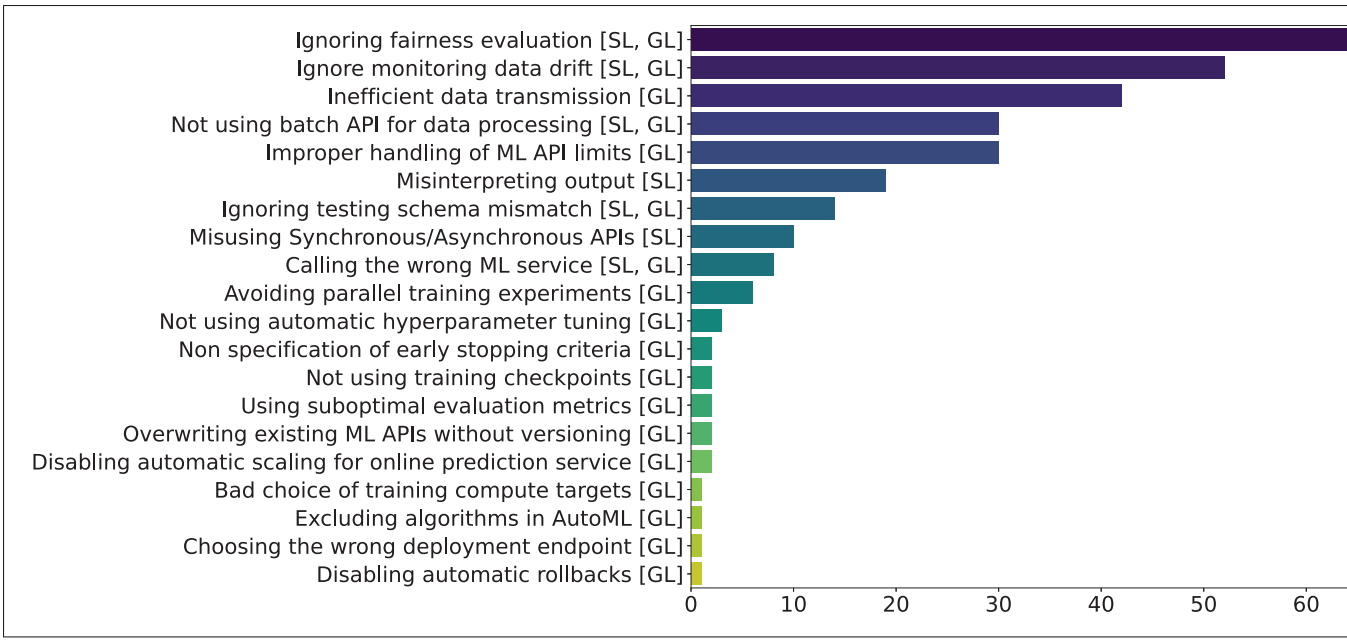


Figure 2.16 Occurrences of ML service misuses in our studied projects, including the misuses identified in the scientific research literature [SL] and the gray literature [GL]

reported by practitioners. The prevalence of such misuses in real-world projects reinforces the need for increased awareness, improved best practices, and the development of automated tools to detect, prevent, and mitigate these issues. Addressing these misuses proactively could significantly enhance the reliability, efficiency, and cost-effectiveness of ML service-based systems in both enterprise and open-source domains. However, some discrepancies should be noted. Our survey results indicate that a significant percentage of participants have frequently encountered specific misuse in their professional experience, such as "*Improper Handling of ML API limits*" (31%). Yet, our analysis of open-source projects revealed that this misuse occurred only in 10% of the cases (30 out of 290), highlighting a notable gap. Several factors may explain this discrepancy. First, survey participants often work with production-level ML systems where misuses are more common but remain undocumented in public repositories. In addition, many ML misuses occur within private workflows, deployment pipelines, or closed-source production environments, making them less visible to open-source communities, such as GitHub, which formed the basis of our analysis. Also, the similar prevalence of ML service misuses in both

open-source and industrial projects such as *"Ignoring fairness evaluation"* and *"Ignoring monitoring for data drift"* may partly stem from developers working with relatively stable or benchmark datasets, where changes in data distribution are perceived as unlikely. For example, when using state-of-the-art or well-established datasets, developers may feel confident enough to consciously deprioritize or omit monitoring mechanisms, which could explain why these misuses persist in practice. Prior work also reported that data drift monitoring is sometimes ignored when datasets are assumed to be neutral or lack protected attributes, leading developers to consider drift monitoring irrelevant in such contexts (Polyzotis *et al.*, 2019).

Also, some practitioners reported having no systematic processes or tools to ensure fairness, instead relying on ad hoc detection by developers who *"spot an issue that looks like a fairness issue to them, and then they talk with each other about it, and then find some specific solution to it"* rather than employing structured and automated approaches (Madaio, Stark, Wortman Vaughan & Wallach, 2020). Some teams even experimented with inferring demographics from indirect features to enable fairness checks, but expressed concerns that such indicators might themselves *"introduce undesirable biases, introducing a need to audit the auditing tool"* (Holstein, Wortman Vaughan, Daumé III, Dudik & Wallach, 2019). Similarly, practitioners working on service or consulting tasks described challenges *"relating to datasets outside of their control and about monitoring deployment contexts or fairness criteria after a system has been handed off to a customer"* (Madaio *et al.*, 2020). Hence, once a system is deployed, teams often lose visibility into the data and model usage, complicating fairness maintenance and causing issues to surface long after deployment. Together, these observations help explain why both *"Ignoring fairness evaluation"* and *"Ignoring monitoring for data drift"* remain prevalent: developers either deprioritize these tasks under assumptions of dataset stability or neutrality, or lack automated mechanisms and post-deployment visibility to consistently perform them. Future research should explore this disconnect by conducting industry case studies or practitioner interviews to understand how such misuses are managed in real-world applications. Integrating insights from practitioners with open-source analysis would provide a comprehensive understanding of ML service misuses and their broader implications.

Takeaway 6: Common ML service misuses, like ignoring fairness evaluation and data drift monitoring, appear similarly in both open-source projects and industry. However, some misuses, like improper handling of ML API limits, are more frequent in industry, reflecting differences between open-source and production environments.

2.5 Discussion and Recommendations

Towards a better understanding of ML service misuses. We present the largest catalog of 20 ML service misuses (summarized in Table 2.2), which we identified through a multi-vocal study and confirmed through a survey of 50 practitioners. We identified 290 occurrences of misuses based on the analysis of a curated set of 377 ML service-based systems. Our study not only maps out these misuses in the literature and ML service-based systems, but also quantifies practitioners' agreement with our catalog and the frequency of encountering these misuses in the industry. Moreover, ML service misuses can create technical debt by embedding hidden complexities and maintenance challenges into the system. For example, ignoring data drift monitoring can cause ML service–based systems to make predictions on operational data that has shifted from the training distribution. Over time, this leads to performance degradation that is difficult to trace to its root cause, risking teams to implement repeated short-term fixes instead of addressing the underlying problem. These accumulated issues increase system complexity, elevate operational costs, and hinder the system's ability to evolve or extend without costly refactoring.

While our survey focused on a subset of 20 validated misuses, we acknowledge that practitioner input on the remaining 14 misuses that we identified in our multivocal study could provide valuable additional insights. Exploring these misuses in future work represents a promising avenue for expanding our catalog and validating their relevance in real-world settings. Also, we should note that the disparity between frequency (e.g., 68%) and agreement (e.g., 52%) for certain misuses, such as "*Not using batch APIs for data processing*", highlights that while frequency indicates how often participants encounter the issue, agreement reflects a shared

Table 2.2 Summary of the Developed Catalog

	ML Development Pipeline Stage	Misuse Name	Example	Recommendation	Frequency %		Agree %
					Most	Some	
1	**Data Collection & Preprocessing**	Not using Batch API for data processing	Processing multiple documents with separate API calls	Load data in batches to optimize performance and prevent OOM issues	22%	68%	52%
2		Inefficient data transmission	Repeated data transfers instead of caching	Cache data locally to cut transfer time and costs	20%	70%	57%
3	**Training**	Non-specification of early stopping criteria	Overtraining without stopping criteria	Set early stopping to reduce training time and avoid overfitting	24%	56%	72%
4		Avoiding parallel training experiments	Missing distribution configuration	Enable parallel training with a distribution config	18%	66%	54%
5		Not using automatic hyperparameter tuning	Manually tuning hyperparameters	Use automated hyperparameter tuning for efficiency	16%	68%	62%
6		Not using training checkpoints	No checkpointing in training	Save checkpoints to resume training and prevent data loss	14%	64%	68%
7		Bad choice of training compute targets	Using non-recommended Azure ML Kubernetes	Choose compute targets suited to the ML task	18%	66%	78%
8		Excluding algorithms in automated ML	Ignoring a promising algorithm	Include all relevant algorithms for better performance	16%	52%	57%
9	**Testing**	Misinterpreting the output	Focusing on 'score' while ignoring 'magnitude'	Use rule-based evaluations with multiple metrics	20%	62%	80%
10		Ignoring fairness evaluation	Not checking demographic biases	Ensure fairness with bias evaluation tools	26%	58%	80%
11		Ignoring testing schema mismatch	Disabling Amazon ML schema alerts	Set alerts for data schema mismatches	18%	64%	74%
12		Using suboptimal evaluation metrics	Using accuracy instead of AUC in imbalanced data	Select evaluation metrics suited to the task	18%	60%	78%
13	**Deployment**	Overwriting existing ML APIs without versioning	No model versioning, preventing rollback	Use ML API versioning for tracking and rollback	18%	66%	80%
14		Choosing the wrong deployment endpoint	Using batch endpoints for real-time tasks	Use online endpoints for real-time and batch for *async* tasks	20%	64%	80%
15		Disabling automatic rollbacks	Disabling automatic deployment rollbacks	Enable auto rollbacks for stable production models	20%	60%	72%
16		Disabling auto-scaling for online prediction service	No auto-scaling in Amazon Sagemaker	Configure auto-scaling for dynamic resource allocation	20%	66%	54%
17	**Serving**	Improper handling of ML API limits	Exceeding API call limits	Set API rate limits for stable performance	31%	45%	72%
18		Misusing Sync/Async ML APIs	Calling asynchronous ML services synchronously	Use async APIs for large inputs, sync for low-latency tasks	18%	60%	68%
19		Calling the wrong ML service API	Using image classification instead of object detection	Choose the right API for the task	14%	60%	76%
20	**Monitoring**	Ignoring monitoring for data drift	Not setting up alerts for data drift	Detect and address skew and drift for model reliability	26%	58%	72%

understanding of its problematic nature. This gap suggests that observing a misuse does not guarantee consensus on its consequences, likely due to variations in participants' interpretations, project contexts, or professional backgrounds. In addition, future research could take a reverse approach by first collecting misuses observed by practitioners in their projects and then examining their presence in open-source projects. This could reveal new, practice-driven misuse patterns and help refine specification approaches to better capture issues that arise in real-world systems. Moreover, a deeper analysis of our results segmented by cloud providers could offer valuable insights into potential differences in misuse patterns across platforms. Unfortunately, the survey data collected in this study did not include sufficient information to perform such an analysis. Future work should explore this aspect, as it could help uncover cloud-specific challenges and inform more targeted recommendations for practitioners.

Towards new methodologies for detecting ML service misuses. Our study establishes a foundation for developing automated detection and refactoring tools to enhance the quality of ML service-based systems. While some misuses can be relatively easy to detect and refactor (e.g., *"Non specification of early stopping criteria*", *"Misinterpreting output*"), others (e.g., *"Ignoring fairness evaluation*" or *"Monitoring for data drift*") may require more sophisticated techniques. Each misuse in our catalog is tied to a specific ML development stage and is often reflected in concrete code patterns or configurations, making them amenable to static or dynamic analysis. For instance, misusing synchronous APIs or missing checkpointing logic can be identified through source code inspection, while deployment misconfigurations (e.g., lack of versioning or disabled auto-scaling) can be detected in infrastructure-as-code or configuration files. These characteristics open opportunities for building static analyzers, IDE plugins, or CI-integrated checkers to proactively detect and address ML service misuses. By cataloging these issues and linking them to observable artifacts, we provide a concrete basis for tooling that supports proactive quality assurance. Moreover, future research should also invest in extending and refining the proposed catalog, formalizing misuse patterns, and developing practical tools to help practitioners adhere to best practices and improve maintainability.

Towards more awareness of ML service misuses. The positive correlation between practitioners' experience levels and the frequency of encountering ML service misuses highlights the importance of knowledge transfer from experienced to less experienced practitioners. This emphasizes the need to raise awareness about best practices in ML service usage and improve the overall quality of ML service-based systems. This carries significant implications for several stakeholders in the ML ecosystem, including software and ML engineers, ML cloud providers, open-source platforms like GitHub, and researchers.

- **For practitioners:** Our catalog of ML service misuses offers valuable insights into common pitfalls and misconfigurations that can compromise the reliability of ML service-powered systems. With this comprehensive reference, practitioners can better understand potential risks and proactively design and integrate ML services that align with best practices. For example, software engineers, who often serve as the bridge between model development and production environments, can leverage this catalog to mitigate risks related to fairness, scalability, and performance.
- **For ML cloud service providers:** Our study highlights the importance of developing tools and platforms to facilitate the detection and prevention of ML service misuse. Some misuses, such as *"Overwriting existing ML APIs without versioning*"), are relatively straightforward and could be automatically detected. Building such tools would help ensure users benefit from more transparent, scalable, and high-performance ML services.
- **For open-source platforms:** Platforms such as GitHub and other developer communities can play a critical role in promoting collaboration to identify and address these misuses. They can support the creation of shared resources that enhance the performance of ML service-based systems. For instance, GitHub could integrate tools that detect common ML service misuses during pull requests or commits, flagging potential issues early in the development lifecycle.
- **For researchers:** Our catalog highlights several underexplored areas in ML service misuse that warrant deeper investigation, such as the underlying rationale behind practitioner decisions and the impact of emerging ML cloud technologies on misuse patterns. Researchers can build upon our findings to develop educational resources and design frameworks that guide

practitioners toward best practices. Future research should also focus on the development of automated detection and refactoring tools specifically targeting ML service misuses.

Overall, our findings trigger an alarm towards investing in more robust tools, policies, and guidelines to safeguard the integrity, reliability, and societal impact of ML cloud services in particular, and ML-based systems in general.

2.6 Threats To Validity

This section outlines the threats to the validity of our findings.

Construct validity: Construct threats to the validity concern the relationship between the theory and the observations made. Regarding our findings, one key concern is the method used to collect GitHub projects with identified misuses. Our scripts for project identification may have unintentionally excluded or included certain projects that were not entirely relevant to the scope of our study. To mitigate this risk, we conducted a manual review of the projects to ensure their relevance to our research on ML service misuses. This additional step helped refine the dataset and enhance its accuracy.

Furthermore, while our catalog of 20 misuses is based on a thorough review of both research and gray literature, there is always a possibility that some misuses were overlooked. Despite being thorough, the catalog may not capture every potential misuse in ML services. For instance, platforms such as Stack Overflow could offer additional valuable insights into other crucial misuses not yet covered in our catalog. To address this, we aim to extend the catalog in a parallel independent study, incorporating contributions from such platforms to further strengthen its completeness and relevance in the context of ML service misuses.

Internal validity: Internal threats to validity are concerned with the causal relationship between treatment and outcome. Our search query might not cover all terms related to ML service misuses and could miss important research and gray literature. We accept these threats as our goal was not to provide an exhaustive list of all existing ML service misuses. As a mitigation

solution, we (1) included in our search query the most important keywords related to the software quality of ML service-based systems, and (2) applied forward and backward snowballing to minimize the risk of missing important resources.

Furthermore, social desirability is a bias that leads our survey participants to agree with misuses. To minimize this threat, we did not offer any incentives for practitioners to participate in our survey. We also guaranteed the practitioners their anonymity and emphasized that all the reported information would be for research purposes only. Another internal threat to the validity of our findings is the practitioners' interpretation of the identified misuses. To mitigate this, we provided a link to our online catalog within the survey, allowing participants to access detailed descriptions of each misuse. In addition, we categorized the misuses according to their respective stages within the ML development pipeline, ensuring a structured understanding. The catalog also includes real-world examples of misuse occurrences, along with proper references, to facilitate a more informed and accurate assessment of the misuses. To complement the quantitative findings, our survey included open-ended questions aimed at capturing qualitative insights from practitioners regarding the misuse and best practices of ML services. While several participants responded to the open-ended questions, their answers were brief and lacked depth (e.g., expressing appreciation or thanking the researchers), making it difficult to conduct a meaningful thematic analysis. To mitigate this threat, we excluded these responses from our analysis and instead relied on qualitative evidence from our gray literature review (e.g., online resources, documentation, and prior research papers). We further extended invitations for interviews to practitioners to collect additional insights. However, we did not receive any responses. This aligns with known challenges in software engineering research, where recruiting practitioners for interviews has been widely reported as difficult (Ghaleb, Hassan & Zou, 2022; Kokinda, Moster, Dominic & Rodeghero, 2023). We acknowledge that interview-based data could have provided greater depth and consider this an avenue for future work.

One potential threat to internal validity is manual analysis, which could introduce bias in the findings. Moreover, the notion of what is an ML service "misuse" may be open to interpretation due to the lack of a universally formalized definition in the literature. To mitigate

this, two evaluators (co-authors of this paper, with experience in machine learning and software engineering) independently analyzed the projects based on the same criteria. All discrepancies were resolved through discussion and consensus meetings. The agreement level between the evaluators was measured using Cohen's kappa and showed a strong level of consistency, reducing the risk of human subjectivity and error in the analysis.

External validity: These threats address the generalizability of our findings. Our study focused on a sample of a curated set of 377 GitHub projects, all written in Python. While this sample provides valuable insights into ML service misuses, it may not fully represent the broader landscape of ML service misuses. The limited scope in terms of both project numbers and programming language means that the findings might not be directly generalizable to other programming languages. Future research should focus on expanding this sample to include a more diverse range of projects across different programming languages to enhance the generalizability and applicability of the results.

Moreover, some misuses in our catalog, such as "Not using batch API for data processing" and "Ignoring fairness evaluation," may not apply in all contexts. For example, batch API misuse is relevant only when batch processing is required, and fairness evaluation mainly concerns datasets with protected attributes. However, fairness assessment remains important, as biases can also emerge from complex feature interactions or evolving populations. To address this, we clarified in the catalog the conditions and contexts in which these misuses apply. For example, we distinguished between batch and real-time streaming data when identifying batch API misuse to reduce the risk of misclassifications and misunderstandings. While our catalog is specifically designed around cloud-based ML services, several of the misuses identified are also relevant to ML systems more broadly. However, the impact of the identified misuses in cloud environments is more pronounced and leads to notable effects, such as increased latency, higher costs, and degraded performance. These impacts may be different or less severe in non-cloud or traditional ML environments (Yao *et al.*, 2017).

Though our survey included 50 practitioners (a 12% response rate), we believe this sample is somewhat reasonable, as achieving high response rates is challenging in this kind of studies (Ghazi *et al.*, 2018), especially in our context, given the survey length and the difficulty of identifying professionals with specific experience in ML cloud services. Importantly, the participants represent a diverse group of practitioners, all actively involved in ML service-based projects. This diversity, coupled with the hands-on experience of the participants, strengthens the reliability of our findings as it reflects the insights of subject matter experts. Nevertheless, we acknowledge that the final response rate (12%) and the small number of respondents may introduce self-selection bias and limit the statistical generalizability of our results to the broader ML practitioner population. To mitigate this risk, we reached out to participants through diverse channels (e.g., professional networks and email), ensured representation of different roles and experience levels, and limited outreach to no more than five individuals per company to avoid over-representation of any single organizational perspective.

Another external threat pertains to the coverage of reasons and mitigation solutions for ML misuses. To mitigate this threat, we mentioned possible reasons and mitigation solutions in the survey based on our review of the literature and practical experience. Before disseminating the survey, we conducted a pilot study with three professionals in the field to verify the language and coverage of the proposed answers. Furthermore, we provided an "*Other*" option in the survey, allowing participants to freely input additional open-ended reasons and mitigation solutions for ML service misuses. Yet, none of the 50 practitioners suggested additional reasons or mitigation solutions, indicating that the predefined options comprehensively captured the key factors related to these elements. Despite these efforts, we acknowledge that further research is needed to deepen our understanding of the underlying causes of ML service misuses in software systems and to develop more effective mitigation strategies.

2.7 Conclusion

This chapter presented a comprehensive, multi-vocal empirical study of misuses of ML cloud services. We propose a catalog of 20 ML service misuses based on a research literature review,

a gray literature review, and an analysis of GitHub projects. We defined these misuses in detail, along with their impact on ML service-based systems and the stage at which they are more likely to occur. We validated our catalog through an online survey with ML practitioners, in which we measured the level of their agreement with our catalog and the frequency of encountering each misuse in practice. In addition, we reported, based on practitioners' experience, the common reasons behind the presence of such misuses and the recommended solutions to mitigate them. Our study offers developers interested in ML cloud services a thorough understanding of the bad practices to avoid when developing ML service-based systems. Moreover, our study raises awareness of ML service misuses and encourages the involvement of various stakeholders, including cloud providers, open-source platforms, practitioners, and researchers, to mitigate them. In future work, we aim to explore additional ML service misuses from sources such as Stack Overflow to expand our catalog, as well as develop a tool to automatically detect the presence of these misuses, enhancing the accuracy and speed of identification processes. We also intend to create an automated refactoring solution to effectively remove such misuses, thereby improving code quality and maintainability.

CHAPTER 3

MLMISFINDER: A SPECIFICATION AND DETECTION APPROACH OF MACHINE LEARNING SERVICE MISUSES

Hadil Ben Amor[1] , Niruthiha Selvanayagam[1] , Manel Abdellatif[1] , Taher A. Ghaleb[2] , Naouel Moha[1]

[1] Department of Software Engineering, École de Technologie Supérieure
1100 Notre-Dame West, Montreal, Quebec, Canada H3C 1K3
[2] Department of Software Engineering, Trent University
1600 West Bank Drive, Peterborough, Ontario, Canada K9L 0G2

Article accepted to SANER, December 2025

Abstract

Machine Learning (ML) cloud services, offered by leading providers such as Amazon, Google, and Microsoft, enable the integration of ML components into software systems without building them from scratch. However, the rapid adoption of ML services, coupled with the growing complexity of business requirements, has led to a rise in misuses by software developers, compromising the quality, maintainability, and evolution of ML service-based systems. Though prior research has explored patterns and antipatterns in service-based and ML-based systems separately, automatic detection of ML service misuses remains a challenge. In this work, we propose MLmisFinder, an automatic approach to detect ML service misuses in software systems, aiming to identify instances of improper use of ML services to help developers properly integrate ML components in ML service-based systems. To do so, we propose a metamodel that captures the data needed to detect misuses in ML service-based systems. We then apply on the metamodel a set of rule-based detection algorithms for seven types of misuse. We validated our approach on 107 software systems collected from open-source GitHub repositories and compared our results with a state-of-the-art (SOTA) baseline. Our results show that MLmisFinder effectively detects ML service misuses, achieving an average precision of 96.7% and recall of 97%, outperforming the state-of-the-art baseline. Moreover, MLmisFinder demonstrated high efficiency and scalability in detecting misuses across 817 ML service-based

systems and revealed that such misuses are widespread, particularly in areas such as data drift monitoring and schema validation.

3.1 Introduction

Machine Learning (ML) cloud services have quickly become central components of modern software systems, offered by leading providers such as Amazon, Google, and Microsoft. Their widespread adoption is driven by the ability to streamline ML model development, reducing the need for extensive expertise and the complexity of building models from scratch (Zhang, Yu, Wang & Yan, 2019), which has greatly facilitated the integration of ML services into software systems. Existing ML services range from extremely simple to fully customizable, supporting the development of a wide variety of ML service-based systems (ML, 2024a). The growing demand for business solutions and the increasing popularity of ML services have encouraged developers of all skill levels to adopt them to accelerate system development, maintenance, and evolution (Wan *et al.*, 2021). However, developers do not always follow best practices when designing and using ML services, which can lead to misuses that degrade system quality and complicate maintenance and evolution (OBrien *et al.*, 2022a; Washizaki *et al.*, 2022). ML service misuse involves violating implicit or explicit ML service usage constraints within software systems (Ben Amor, Abdellatif & Ghaleb, 2025; Wan *et al.*, 2021) and improper practices throughout the system's lifecycle. Such misuses can result in critical bugs that negatively affect the accuracy, performance, and cost-effectiveness of ML service-based systems (Wan *et al.*, 2021). For example, ML cloud providers enforce API rate limits to manage resources and prevent overload. Exceeding these limits without proper handling can lead to critical issues, such as increased latency. Similarly, failing to specify a stopping criterion during model training with an ML cloud service can increase both latency and cloud usage costs, as training may continue unnecessarily without significant performance gains (Policy, 2024). Therefore, detecting misuses of ML services is essential for improving software quality and performance.

Previous studies have explored the specification and detection of code smells and ML antipatterns from different perspectives (Bogner *et al.*, 2021; Washizaki *et al.*, 2019; Wan *et al.*, 2021;

Cabral *et al.*, 2024; Wei *et al.*, 2024). However, little attention has been paid to misuses of ML services (Wan *et al.*, 2021; Wei *et al.*, 2024). Therefore, in this work, we propose MLmisFinder, an automated approach to detect ML service misuses in ML service-based systems. MLmisFinder is built on a novel metamodel that unifies the representation of ML service-based systems across different cloud providers to detect ML service misuses. This metamodel forms the foundation for a set of extensible static detection rules tailored to ML service misuses, resulting in a highly accurate detection approach. MLmisFinder supports the detection of seven misuse types, most of which have not been previously studied in the context of ML cloud services. It leverages static analysis of source code to identify these misuses without requiring runtime data, enabling earlier detection during the development lifecycle and ensuring broader code coverage. To validate the accuracy of our findings, we analyzed 107 ML service-based systems and computed the precision and recall of MLmisFinder. We also compared our detection results to a state-of-the-art baseline (Wan *et al.*, 2021), identifying one misuse common to both approaches.

Our results show that MLmisFinder achieves a precision of 96.7% and a recall of 97%, outperforming the only existing SOTA baseline and demonstrating its effectiveness in identifying ML service misuses, with 340 instances of misuses successfully detected across validated systems. We analyzed the prevalence of these misuses across a dataset of 817 ML service-based systems. Our findings reveal that misuses are widespread, with `Ignoring monitoring data drift` and `Ignoring testing schema mismatch` occurring in 97% and 96% of the systems, respectively. This highlights the need for better adherence to best practices and improved developer tools for ML service integration.

The rest of this work is organized as follows. Sections 3.2 presents the background. Section 3.3 describes our approach. Section 3.4 outlines our experimental setup. Empirical evaluation is presented in Section 3.5. We discuss the implications of our findings in Section 3.6 and the threats to validity in Section 3.7. Section 3.8 concludes the work and suggests possible future work.

3.2 Background

In this work, we consider seven misuses discussed in the literature (Ben Amor *et al.*, 2025; Cao *et al.*, 2022; Xu *et al.*, 2022; Mallick *et al.*, 2022; Wan *et al.*, 2021). Our selection was based on three criteria: (a) we selected misuses that cover various stages of the ML development pipeline to demonstrate that our approach is applicable across the lifecycle of ML service-based systems; (b) we prioritized misuses that are detectable through static analysis of code and service configurations; (c) we prioritized misuses having documented impact on maintainability and software quality, causing inefficiencies and technical debt. Therefore, we included seven misuses: "*Not Using Batch APIs for Data Processing*", "*Not Using Training Checkpoints*", "*Non Specification of Early Stopping Criteria*", "*Ignoring Testing Schema Mismatch*", "*Misinterpreting Output*", "*Improper Handling of ML API Limits*", and "*Ignoring Monitoring for Data Drift*", previously described in Section 2.4.1.4.

3.3 Approach

In this section, we present MLmisFinder, a fully automated approach for detecting the ML service misuses outlined in Section 3.2. We should note that our detection approach supports three major ML cloud providers: AWS, Azure and Google. Figure 3.1 provides an overview of our approach. MLmisFinder starts by taking the GitHub repository link of an ML service-based system as input. Then, it clones the repository and parses the source code to extract the necessary data for misuse detection. This data is structured using a dedicated metamodel, which serves as the basis for the detection process. To populate the metamodel, MLmisFinder leverages a set of parsers and generates an Abstract Syntax Tree (AST) to analyze the source code and generate the required metamodel constituents. The metamodel is then instantiated to build a concrete model of the project to analyze. Once the metamodel is instantiated, MLmisFinder applies a set of detection rules (referred to hereafter as detection algorithms) to the model's constituents. These algorithms identify and count the occurrences of each specified ML service misuse. By combining static analysis, metamodel instantiation, and dedicated detection algorithms, MLmisFinder offers the first fully automated approach for detecting such ML service misuses

in software systems. In the following, we describe our metamodel and the detection algorithms for the seven ML service misuses that MLmisFinder detects.

3.3.1 Metamodel Definition

We developed a metamodel to describe the data required by our detection algorithms, ensuring adaptability across diverse environments and use cases. Designed for extensibility, it separates key concepts such as data sources, processing steps, ML services, and configuration parameters, allowing new misuse types to be captured by extending or recomposing existing elements. The metamodel defines the core detection components and their relationships, offering a structured representation of the system's environment, configuration, codebase, cloud services, and data sources. Figure 3.2 illustrates the constituents of our metamodel and their associations.

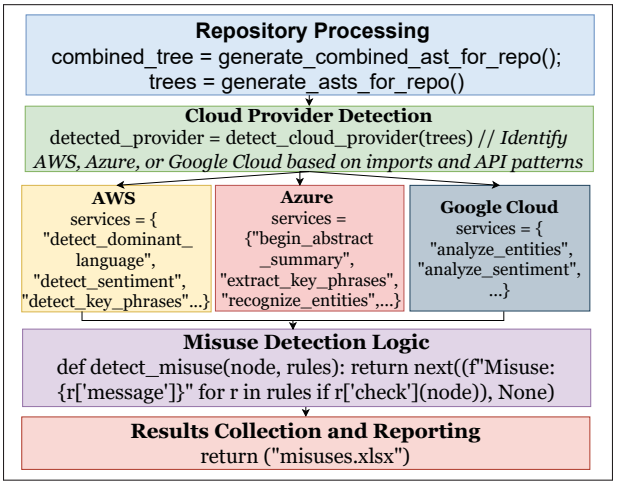


Figure 3.1 Overview of MLmisFinder

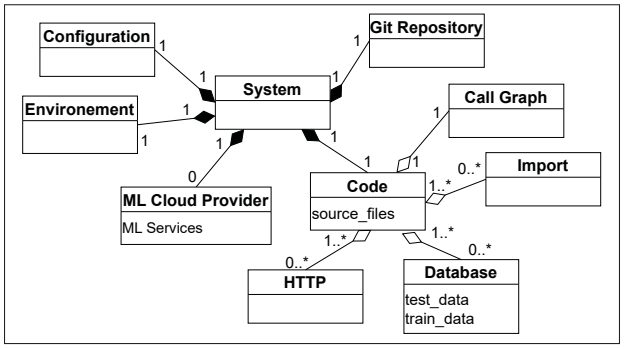


Figure 3.2 Metamodel constituents in MLmisFinder

3.3.2 Metamodel Constituents

- **`System`**: This is the root of the metamodel. It represents the ML service-based application and its operational context. It interacts with key external and internal constituents such as the `Environment`, `Configuration`, `Git Repository`, `ML Cloud Provider`, and `Code`.

- **Environment**: Represents the overall context in which the System operates and manages data about the commonly used environment variables that are dynamically injected into a system.
- **Configuration**: Stores data gathered from the configuration files of the ML services and defines how the System and its services are set up, such as enabled/disabled features.
- **ML Cloud Provider**: Represents external platforms that provide ML services, such as AWS, Azure, or Google Cloud, which are used by the System. It offers ML Services to support various ML functionalities, such as model training, inference, or data processing.
- **Git Repository**: The version-controlled storage that holds the System's Code (source_files) that is analyzed to detect potential misuses.
- **Code**: Consists of the System's source files. It plays a central role in misuse detection and includes:
  - Import: A list of imported libraries and dependencies in the source code.
  - HTTP: A list of external HTTP requests made by the System, often used for ML service calls.
  - Database: Represents the data storage component, containing train_data and test_data, which are critical for model evaluation and prone to misuse.
  - Call Graph: A representation of function calls within the Code, derived from an AST to analyze system interactions.

3.3.3 Metamodel Instantiation

MLmisFinder's metamodel is instantiated by analyzing various aspects of the ML service-based system, as follows.

1. ***Code Extraction***: The System's source code is parsed using Python's *ast* module to extract function calls, control flow structures, and import statements.
2. ***Call Graph Construction***: An AST is used to generate a Call Graph, capturing function relationships and interactions.

3. ***ML Cloud Provider Identification***: The `System` is checked for predefined invocation patterns associated with cloud-based ML platforms such as AWS, Azure, and Google Cloud, including *`boto3`*, *`azureml`*, and *`vertexai`*, respectively. This helps establish a contextual foundation that guides the detection of misuses.
4. ***Database Analysis***: Training and testing datasets are identified by detecting function calls such as `train_test_split`, `train`, `fit`, `predict`, and `evaluate`.
5. ***HTTP Request Detection***: External API calls to ML services are identified and analyzed for potential misuse.
6. ***Configuration and Environment Analysis***: Configuration files and environment variables are examined to identify relevant framework settings and execution parameters.

3.3.4 Detection Algorithms

For each ML service misuse, we defined a detection algorithm to detect its occurrences in a given ML service-based system. Once a misuse is detected, MLmisFinder generates a report containing the repository name, misuse type, number of occurrences, and the specific line of code where the misuse occurred, if applicable (e.g., the case of "*Not Using Batch API for Data Processing*"). In the following, we provide a detailed description of each detection algorithm.

3.3.4.1 Not Using Batch API for Data Processing

This misuse occurs when batch-processing APIs are available but inefficiently used within loops instead of processing multiple inputs in a single request. The detection algorithm verifies the `ML Cloud Provider` used, examines API calls in `Code`, and checks the `Call Graph` to determine if batch APIs are invoked inside loops, as illustrated in Algorithm 3.1.

Listing 3.1: Not Using Batch API for Data Processing

```
# Start: Analyze ML API calls in Code
Compare API calls to known batch-processing APIs
if API supports batch processing:
    Check if API is used inside loop(s)
    if Batch API is used inefficiently:
        Flag misuse
```

```
    else:
        No misuse detected
else:
    No misuse detected
```

3.3.4.2 Not Using Training Checkpoints

Checkpointing helps prevent loss of progress during training interruptions. The detection algorithm examines the `Import` constituent for `ML Cloud Provider` to determine whether a training component is present in the system. If identified, the algorithm proceeds to further examine the `Call Graph` to track checkpoint-saving and restoration functions, and ensures checkpoints are correctly implemented in `Code`, as demonstrated in Algorithm 3.2.

Listing 3.2: Not Using Training Checkpoints

```
# Start: Analyze ML Cloud Provider SDK
if No SDK detected:
    No misuse detected
else:
    Analyze checkpoint saving/restoration calls
    if No checkpointing functions are found:
        Flag misuse
    else:
        if Checkpoints are saved but never restored:
            Flag misuse
        else:
            No misuse detected
```

3.3.4.3 Non Specification of Early Stopping Criteria

Early stopping is crucial to prevent overfitting. The detection algorithm begins by checking whether any training component exists in the `Code`. If present, it subsequently verifies whether early stopping-related libraries are imported in `Import`, analyzes function calls in `Call Graph`, and ensures that proper stopping criteria are set in `Code`, as illustrated in Algorithm 3.3.

Listing 3.3: Non Specification of Early Stopping Criteria

```
# Start: Analyze ML Cloud Provider SDK
if No SDK detected:
    No misuse detected
else:
    Analyze early stopping libraries
    if No early stopping library is imported:
        Flag misuse
    else:
        Analyze early stopping function usage
        if Early stopping is not properly configured:
            Flag misuse
        else:
            No misuse detected
```

3.3.4.4 Ignoring Testing Schema Mismatch

ML pipelines should validate schema consistency between training and testing datasets. The detection algorithm inspects the `Import` constituent for validation libraries, checks whether schema validation functions are used in `Code`, and examines the `Database` to determine if train and test datasets are explicitly compared for schema consistency, as depicted in Algorithm 3.4.

Listing 3.4: Ignoring Testing Schema Mismatch

```
# Start: Analyze Import constituent for schema validation libraries

if Validation libraries are missing:
    Flag misuse
else:
    Analyze Code for validation function usage
    if Validation functions are not used:
        Flag misuse
    else:
        Check if train and test data schemas are compared
        if No schema comparison is performed:
            Flag misuse
        else:
            No misuse detected
```

3.3.4.5 Misinterpreting Output

The detection algorithm inspects the `Code` to determine how the ML model's output is accessed or interpreted. It examines the `Call Graph` for the presence of calls to output-related methods

Listing 3.5: Misinterpreting Output

```
# Start: Analyze Code and Call Graph
if No output methods are called:
    No misuse detected
else:
    if Only one output method is used or outputs are used incorrectly:
        Flag misuse
    else:
        No misuse detected
```

or properties (e.g., `.score`, `.magnitude`) and checks whether only a limited subset of output methods is used in `Code`, potentially indicating incomplete interpretation of model results, as illustrated in Algorithm 3.5.

3.3.4.6 Improper Handling of ML API Limits

ML services impose API limits that should be handled to avoid failures. The detection algorithm examines `Import` for monitoring libraries, verifies their usage in `Code`, inspects the `Call Graph` for relevant monitoring functions, and checks `HTTP` requests for rate limit monitoring by analyzing request headers and parameters, as outlined in Algorithm 3.6.

Listing 3.6: Improper Handling of ML API Limits

```
# Start: Analyze monitoring libraries
if No monitoring library is imported:
    Flag misuse
else:
    Analyze monitoring function usage
    if Monitoring functions are not used:
        Flag misuse
    else:
        Analyze HTTP requests for rate limit monitoring
        if No rate limit checks are performed:
            Flag misuse
        else:
            No misuse detected
```

3.3.4.7 Ignoring Monitoring for Data Drift

ML models should be monitored for data drift to ensure continued accuracy. The detection algorithm analyzes the `Import` constituent for data drift monitoring libraries, verifies their application in `Code`, and ensures appropriate data drift monitoring functions are instantiated, as shown in Algorithm 3.7.

Listing 3.7: Ignoring Monitoring for Data Drift

```
# Start: Analyze drift monitoring libraries
if No monitoring libraries are imported:
    Flag misuse
else:
    Analyze Code for drift monitoring function usage
    if Drift monitoring functions are not used:
        Flag misuse
    else:
        No misuse detected
```

3.4 Experimental Setup

We now discuss the study design to validate our approach. We applied MLmisFinder on a set of ML service-based systems and compared the detected occurrences against a ground truth (i.e., the instances of the misuses found manually in the systems). The following describes the existing baseline approaches, our evaluation dataset, ground truth construction, and research questions.

3.4.1 Baseline

Since our main focus is on detecting ML service misuses, we selected baseline approaches for comparison based on three criteria: (1) it must detect at least one ML service misuse MLmisFinder detects, to align with our objective; (2) it must be peer-reviewed with open-source implementation for replicability and enable a fair comparison with MLmisFinder; (3) it must support ML cloud services and at least one of the major cloud providers in our study. Based on the first two criteria, we found the approach of (Wan *et al.*, 2021) and

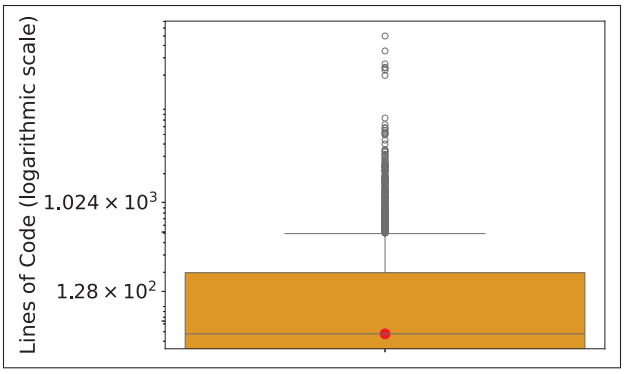
MLScent (Shivashankar & Martini, 2025) to be relevant for our misuse categories. (Wan *et al.*, 2021) detect misuses in the usage of ML APIs from Google and AWS, thus meeting our third selection criterion. The only misuse we share in common with their work is "*Misinterpreting Output*", which aligns directly with our evaluation goals. MLScent is a static analysis tool for detecting ML antipatterns using frameworks such as TensorFlow and PyTorch. It supports the "*Non-specification of Early Stopping Criteria*" misuse, which is the only misuse shared with our work. However, we excluded it since it only detects this misuse at the framework level, not at the ML cloud service level, failing to satisfy the third criterion. Therefore, we retained only the approach of (Wan *et al.*, 2021) to compare with MLmisFinder.

3.4.2 Dataset

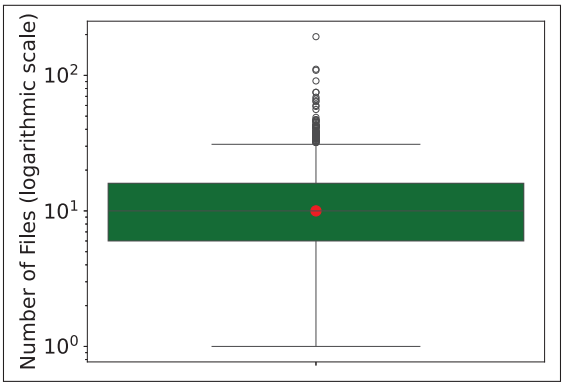
We collected open-source ML service-based projects from GitHub over four months, with no restriction on repository creation dates. We used the GitHub API to automate the retrieval process, allowing structured and efficient access to repository metadata. We used service-specific keywords from cloud provider documentation, such as "*ML cloud*", "*Azure cognitive service*", "*API*", and "*Azure AI*" to identify Azure ML repositories. In addition, we used GitHub's search functionality to detect the presence of these keywords within Python files. Initial data collection yielded 817 GitHub repositories primarily written in Python and related to ML services. We then applied an automated filtration process by analyzing project descriptions and source code to assess relevance. This included checking the presence of Jupyter notebooks and the inclusion of ML-related Python modules. Such criteria refined the dataset to projects actively using ML services, ensuring the quality and focus of our subsequent analysis. All the collected project metadata is available in our replication package (rep, 2025). Figure 3.3 shows statistics on the collected systems, which vary in size, star count, and number of forks.

3.4.3 Ground Truth

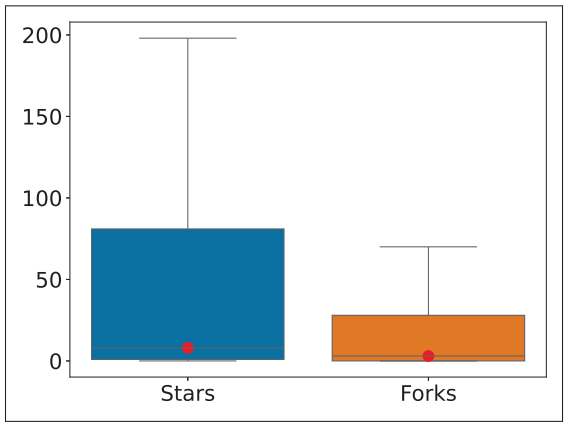
To evaluate MLmisFinder's detection accuracy, we manually constructed a ground truth dataset of ML service misuses. Reviewing all 817 projects, corresponding to 5,719 individual checks per



a) Lines of Code



b) Number of Files



c) Stars and Forks

Figure 3.3 Statistical Distribution of GitHub Repository Metrics

evaluator (seven misuse types per project), would have required significant time and effort. To address this, we employed statistical random sampling to select a representative subset of projects from our collected dataset of open-source ML service-based systems on GitHub (Section 3.4.2). With a ±10 margin of error and a 95% confidence level, we determined a required sample size of 87, which ensures reliable accuracy assessment while remaining feasible for manual analysis. Nevertheless, we analyzed a total of 107 samples to further enhance the reliability and depth of our findings. Three evaluators experienced in ML services and software engineering, manually and independently analyzed the 107 randomly selected projects, to identify instances of the seven misuses. We then calculated Cohen's Kappa coefficient, which reached 84.7%, indicating strong agreement in the manual detection process. This evaluation identified 340 occurrences

of the seven ML service misuses, making our dataset the most comprehensive ground truth collection for ML service misuses, to the best of our knowledge. To facilitate further research, we have publicly released this dataset as part of our replication package (rep, 2025).

3.5 Empirical Evaluation

Our empirical evaluation aims to address three specific research questions.

3.5.1 RQ1. How effective is MLmisFinder in detecting ML service misuses?

Motivation. We aim to evaluate the accuracy of MLmisFinder in identifying the seven misuses through static analysis of ML service-based systems. Specifically, we aim to measure its ability to detect various misuse types across GitHub repositories and compare its detection results with the SOTA baseline.

Results. To evaluate the effectiveness of MLmisFinder in detecting ML service misuses, we applied it to a statistical sample of 107 ML service-based systems to measure its accuracy in identifying such misuses. We compared the detection results with the manually curated ground truth to evaluate precision, recall, and F1 score.

Our empirical evaluation revealed that MLmisFinder consistently achieved high precision and recall, demonstrating its robustness in identifying ML service misuses. A detailed breakdown of detection results across the different misuses is presented in Table 3.1. Specifically, MLmisFinder achieved precision ranging from 80% to 100%, with an average of 96.7%, and values ranging from 76.2% to 100%, with an average of 97%. These results indicate that MLmisFinder correctly detects a large number of ML service misuses while maintaining a low false positive rate. We observe that "*Ignoring monitoring for data drift*" was detected with perfect precision and recall, meaning that MLmisFinder identified all true instances of this misuse without any false positives, highlighting the reliability of our detection algorithm. Certain misuses, such as "*Not using training checkpoints*" and "*Non specification of early stopping criteria*" were detected with relatively high precision (80% and 81%, respectively) and perfect recall, meaning that

MLmisFinder detected every actual misuse of these types while occasionally misclassifying a few non-misuse cases, underscoring the robustness and reliability of our detection algorithms for these particular types of misuses. "*Not using batch API for data processing*" exhibited a slightly lower recall of 90% compared to other misuses, mainly attributed to a limitation in detecting deprecated or unknown ML APIs from cloud providers for batch processing, which are not supported by MLmisFinder as their documentation has been retired, potentially raising another type of warning for ML engineers. MLmisFinder also incorrectly identified two occurrences of the same misuse because of the problem of linked functions, where an ML API is called inside a function that is iterated over in a loop. MLmisFinder may fail to recognize that the ML API is repeatedly invoked in individual iterations instead of processing in an optimized batch mode, especially when the linked functions are complex.

Comparison with *Output Misinterpretation Checker* (Wan *et al.*, 2021). To benchmark MLmisFinder, we ran the *Output Misinterpretation Checker* from (Wan *et al.*, 2021) on a subset of our dataset. While MLmisFinder supports services from three major cloud providers, Wan *et al.*'s tool supports only Amazon and Google APIs. For a fair comparison, we executed their tool on the subset corresponding to these two providers, comprising 74 out of 107 repositories. Of these, 68 were processed successfully, while six failed with *error -2* due to files exceeding 1,000 lines, a known limitation noted in the original study. In valid samples, the baseline achieved a precision of 17.3%, recall of 56.2%, and F1-score of 26.5. While recall was moderate, the low precision and F1-score reflect the limitations of rigid detection rules, which can reduce the reliability of the tool in practice. For example, the tool uses hard-coded heuristics tightly coupled with specific Google ML APIs, limiting its applicability and generalizability across other cloud providers.

In contrast, MLmisFinder uses modular, extensible rules derived from its underlying metamodel of ML service usage and grounded in common misuse patterns across APIs. It performs reliably without file size constraints, enabling broader applicability and more accurate detection of ML service misuses. MLmisFinder employs regular expressions designed to operate across different cloud providers, with patterns broad enough to capture variations in API usage (e.g., *.score*,

.*Sentiment*, .*sentiment*). This generality can lead to false negatives when relevant checks are dispersed across multiple lines, escaping detection. These challenges reflect the limitations of our approach and indicate that, while MLmisFinder mitigates some issues through a structured and extensible metamodel, fully addressing them requires deeper semantic understanding of code behavior.

Table 3.1 Detection Performance Results and Average Execution Time: MLmisFinder vs. Wan et al.

Misuse Type	**MLmisFinder**				**Wan *et al.***			
	Precision	Recall	F1	Avg Time (s)	Precision	Recall	F1	Avg Time (s)
Misinterpreting output	100%	76.2%	86.5%	8.17	17.3%	56.2%	26.5%	70
Not using Batch API for Data Processing	90%	90%	90%	34.73	Not Supported			
Not using training checkpoints	80%	100%	88.9%	1.06	Not Supported			
Non-specification of early stopping criteria	81%	100%	89.5%	1.07	Not Supported			
Ignoring testing schema mismatch	99%	100%	99.5%	1.21	Not Supported			
Improper handling of ML API limits	100%	92.6%	96.2%	1.20	Not Supported			
Ignoring monitoring for data drift	100%	100%	100%	1.18	Not Supported			
Overall	**96.7%**	**97%**	**96.8%**	**50.05**	**12.2%**	**56.2%**	**26.5%**	**70**

Answer to RQ1: MLmisFinder demonstrated a high effectiveness in detecting ML service misuses, achieving an average precision of 96.7% and a recall of 97%. This confirms its reliability as a robust static analysis approach for identifying ML service misuses. It also significantly outperformed the existing baseline in detecting "*Output Misinterpretation*", the only misuse supported by both approaches.

3.5.2 RQ2. How efficient is MLmisFinder in detecting ML service misuses?

Motivation. Our objective is to evaluate the efficiency of MLmisFinder in terms of execution time when detecting misuse of ML services between projects of different sizes and to assess the scalability of our approach. Specifically, we aim to measure how MLmisFinder's performance is affected by factors, such as the number of source files and lines of code within a repository.

Results. To assess the efficiency of MLmisFinder in detecting ML service misuses, we measured its execution time across all 817 projects in our dataset. We also evaluated the execution time for each individual misuse detection algorithm, excluding the time required for metamodel instantiation. The results (presented in Table 3.1) confirm that MLmisFinder is computationally efficient and scalable, making it practical for real-world use, where timely feedback during

development is important. The average execution time of MLmisFinder across the validation projects (up to 19,879 LOCs) is about 50 seconds. As shown in Figures 3.4 and 3.5, execution time scales linearly with LOCs and the number of files, ensuring efficient detection even in large-scale systems. Specifically, while larger projects naturally require more processing time, the increase remains moderate and does not exhibit exponential growth, allowing MLmisFinder to be used effectively across projects of varying complexity and size. We also analyzed variability in execution time across projects of different sizes. As shown in Figure 3.6, small ($\leq 4,045$ LOCs) and medium-sized ($4,046 - 5,169$ LOCs) projects show minimal fluctuations, whereas larger systems ($\geq 5,170$ LOCs) exhibit more variability, with some projects requiring longer analysis. In practice, this means that developers can rely on MLmisFinder for timely detection of ML service misuses, even in large or complex systems, without significant delays in their development workflow.

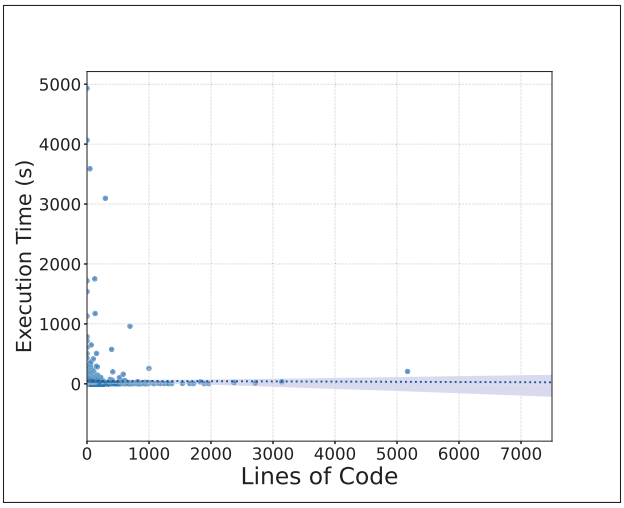


Figure 3.4 Execution Time vs. Lines of Code

Moreover, Table 3.1 shows that most misuse detection algorithms execute in approximately one second, highlighting the efficiency of MLmisFinder. For "*Misinterpreting Output*", MLmisFinder takes around 8 seconds, roughly nine times faster than the *Output Misinterpretation Checker* (Wan *et al.*, 2021), further demonstrating its superior performance. The detection of "*Not Using Batch API for Data Processing*" is more time-consuming, averaging 35 seconds, primarily due to call graph analysis and loop structure evaluation, which require identifying specific ML

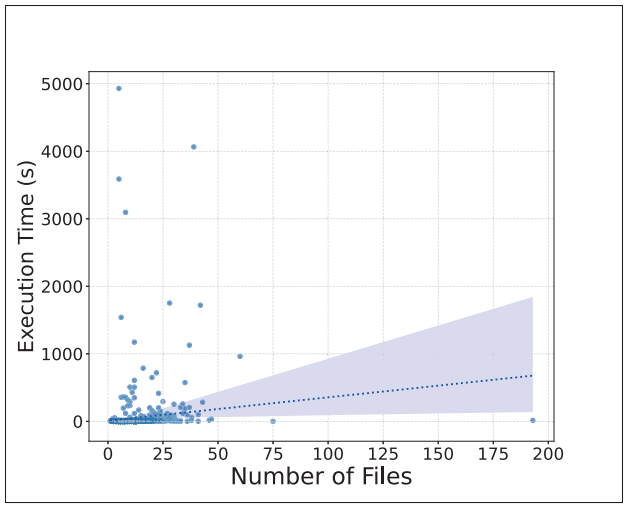


Figure 3.5 Execution Time vs. Number of Files

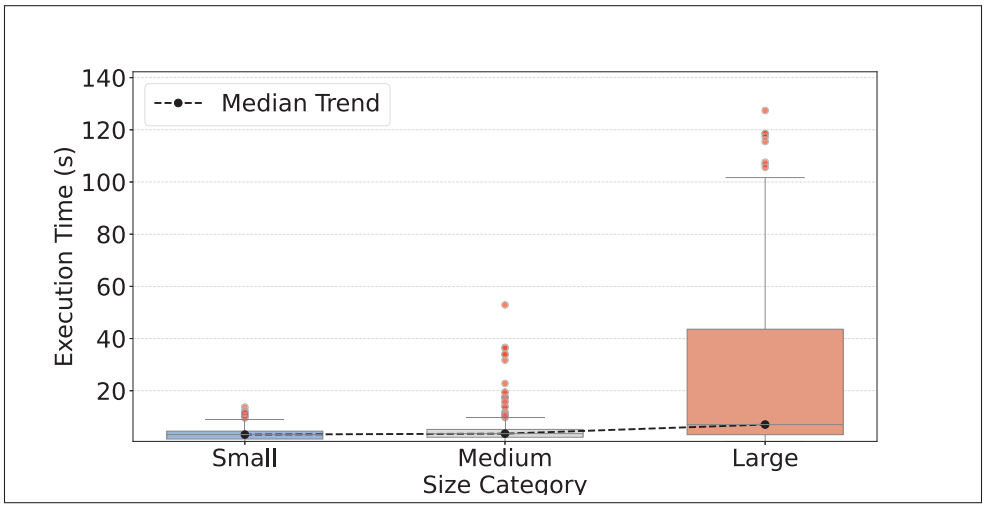


Figure 3.6 Execution Time Variability Across Projects Size

service invocation patterns and assessing inefficient use of batch processing. Nevertheless, the computational overhead remains manageable, and the longer execution time is justified by the complexity of the misuse being detected.

Answer to RQ2: MLmisFinder is efficient and scalable, with execution time scaling linearly with project size. Most algorithms run in about one second, except for "*Not Using Batch API for Data Processing*". Despite some variation in larger projects, execution time remains low, supporting real-world applicability.

3.5.3 RQ3. How prevalent are ML service misuses in ML service-based systems?

Motivation. We aim to analyze the distribution of ML service misuses across different ML service-based systems. Specifically, we aim to understand how frequently different types of misuses occur and check whether certain misuses are more prevalent in real-world projects.

Results. Running MLmisFinder across the 817 repositories revealed a widespread but variable presence of ML service misuses, with some occurring at notably high rates. "*Ignoring monitoring for data drift*" was the most frequently detected misuse, appearing in 98% of repositories (803 occurrences), followed closely by "*Ignoring testing schema mismatch*", found in 97% of projects (789 occurrences). These results suggest that many ML service-based systems do not explicitly implement mechanisms to track data distribution shifts or ensure schema consistency across different stages of the ML development pipeline. However, possible limitations in detection scope should be considered. The absence of data drift monitoring in the source code does not necessarily imply it is completely ignored; it could be implemented elsewhere, such as in external monitoring services, logging mechanisms, or the infrastructure layer, which static analysis may not capture. Similarly, schema validation may not always be relevant, particularly for projects using simple or standard datasets, where developers may not find explicit schema checks necessary, potentially explaining the high prevalence of this detected misuse.

Other misuses, while less common, still occur at notable rates. "*Not using batch API for data processing*" was detected in 48% of repositories, suggesting that nearly half of the projects process data inefficiently, potentially increasing latency and cloud costs. "*Improper handling of ML API limits*" was observed in 36% of repositories, indicating that many developers do not explicitly manage rate limits, which could lead to system failures or degraded performance when API quotas are exceeded. Less frequent misuses include "*Non-specification of early stopping*

criteria" (24%, 199 occurrences), "*Not using training checkpoints*" (17%, 145 occurrences), and "*Misinterpreting output*" (1.5%). Although less prevalent, these misuses can still result in inefficiencies in model training, leading to longer training times and higher computational costs.

Answer to RQ3: There is a widespread and diverse presence of ML service misuse across the projects, with "*Ignoring monitoring for data drift*" and "*Ignoring testing schema mismatch*" being the most common, occurring in 98% and 97% of the projects, respectively.

3.6 Discussion

We describe the limitations of MLmisFinder and the implications of our results for researchers and practitioners.

Implications for Researchers and Practitioners. Our findings have important implications for researchers and practitioners. For practitioners, MLmisFinder offers a concrete and effective solution for detecting and addressing ML service misuses early in the development lifecycle of ML service-based systems. Integrating MLmisFinder into CI/CD pipelines, for example, enables organizations to proactively identify and correct potential misuses before they propagate, directly improving software quality. By catching misuses early, practitioners can enhance maintainability, reduce costly post-deployment fixes, prevent performance degradation, and ensure consistent adherence to best practices for ML service integration.

For researchers, our work emphasizes the need for automated techniques to detect and mitigate ML service misuses, as few approaches target the specific misuse types we address in ML service-based systems. Unlike traditional ML systems, ML services have unique architectural and configuration constraints (e.g., service-specific APIs and hyperparameter tuning), giving rise to new misuse types not handled by existing tools. Our work fills this gap by identifying and detecting a novel class of misuses. The high prevalence of these misuses underscores the importance of raising awareness and improving best practices for ML services. Future research could leverage advanced ML techniques to detect more complex misuses and further refine detection algorithms.

Limitations of MLmisFinder. Despite the effectiveness of MLmisFinder in detecting the seven ML services misuses in our study, some limitations should be acknowledged. First, several detection algorithms in MLmisFinder rely on a manually curated list of cloud ML libraries, which we build by reviewing the official documentation of the three cloud providers supported by MLmisFinder. However, this manual curation process is time-consuming and prone to omissions, particularly for deprecated or newly introduced libraries. This also requires frequent updates for new ML services releases. To address these limitations, automated methods for tracking ML cloud services changes as well as web scrapping of cloud providers documentation updates could be applied to maintain an up-to-date list and maintain the high detection accuracy of our approach.

Second, MLmisFinder relies on static analysis and rule-based detection, which may not fully capture the dynamic behavior and runtime issues inherent in ML service-based systems. This limitation is also noted in other (anti)pattern detection approaches (Tighilt *et al.*, 2023; Wan *et al.*, 2021; Cardozo *et al.*, 2023; Z. Li, 2005). Certain misuses, such as "*Ignoring monitoring for data drift*" and "*Improper handling of ML API limits*", could benefit from continuous monitoring and real-time analysis to improve detection, capabilities that our current implementation does not support. Despite this, MLmisFinder performs strongly using static analysis alone, achieving 96.7% precision and 97% recall, demonstrating its effectiveness in detecting ML service misuses. Integrating a hybrid detection approach that combines static and dynamic analysis represents a promising direction for future work.

Lastly, MLmisFinder currently supports detecting seven ML service misuses across three major cloud providers. While effective, extending support to additional providers, services, and misuse types would increase its applicability as cloud ML offerings continue to grow. The metamodel and modular detection logic make MLmisFinder inherently extensible, allowing new providers, services, and misuses to be integrated with minimal effort. This design enables adaptation to evolving ML service ecosystems, improving detection coverage and maintaining effectiveness in real-world settings, while also supporting cross-provider comparisons and deeper understanding of common misuse patterns.

3.7 Threats To Validity

Internal validity. One potential internal threat to the validity of our study is the reliance on a manually constructed ground truth for evaluating MLmisFinder. To mitigate this threat, multiple evaluators were involved to independently identify instances of misuses and minimize individual biases. Furthermore, we measured inter-rater agreement and observed a high level of consistency among evaluators which reinforces the reliability of our ground truth.

Construct Validity. To develop our approach, we leverage the *ast* Python module, as to the best of our knowledge, there is currently no established method in the literature for parsing Python repositories specifically tailored to our needs. However, the ast module fails to parse files when there is a syntax error in the source code. To mitigate skipping important files, we implemented a preprocessing step that skips lines of code containing syntax errors. This step plays a crucial role in ensuring the robustness of the code analysis by filtering out lines that could introduce errors during parsing.

External validity. MLmisFinder is currently designed for Python-based software systems, which may limit its applicability to other programming languages widely used in industry. However, Python is the language most commonly used for ML development, and we specifically targeted the three most prominent cloud providers in the ML cloud ecosystem. The reliance on a metamodel and dedicated detection algorithms makes MLmisFinder adaptable to other languages and extensible to additional misuses. While we demonstrate the effectiveness of MLmisFinder within this scope, broader validation is needed to improve the generalizability of our results. While static analysis cannot capture misuses introduced via Infrastructure-as-Code or runtime configuration, none of the 107 projects in our dataset relied on such mechanisms, limiting the impact of this limitation. Despite this, our static rules achieved high accuracy across all seven categories, demonstrating their effectiveness. A hybrid static-dynamic approach remains a promising direction for addressing runtime variability and edge cases. Future work will focus on extending support to other programming languages, expanding the range of detectable

misuses, and supporting additional cloud providers to further enhance the generalizability of our results.

3.8 Conclusion

We propose MLmisFinder, a fully automated approach for detecting ML service misuses using a reusable metamodel and rule-based detection algorithms that identify seven common misuse types through static analysis of ML service-based systems. We validated our approach on 107 ML service-based systems on GitHub and compared our results to a SOTA baseline. MLmisFinder achieved an average precision of 96.7% and a recall of 97%, outperforming the only existing baseline. We further applied our approach to 817 ML service-based systems. The results revealed a widespread prevalence of ML service misuses, particularly in data drift monitoring and schema validation, underscoring the need for better adherence to best practices in ML service integration. In future work, we plan to extend MLmisFinder to detect more ML service misuses and incorporate hybrid static-dynamic analysis to capture runtime issues. We also aim to add automated refactoring to fix detected misuses, improving ML service integration, code quality, and system reliability.

CHAPTER 4

REFACTORING ML SERVICE MISUSES WITH LLMS

Introduction

In the previous chapters, we identified a catalog of ML service misuses based on a multi-vocal empirical study. We also constructed a manually validated ground-truth dataset of ML service misuse occurrences in 107 Github projects. Building on this, we developed MLmisFinder, an automated and highly accurate detection approach capable of identifying ML cloud misuses in real-world ML service-based systems. Since such misuses can significantly hinder system maintenance, their refactoring is essential to improve software quality and reduce technical debt. While these insights provide a strong foundation for understanding recurring misuses and their practical impact, they also raise the question of whether SOTA LLMs can automatically refactor these misuses. In this chapter, we evaluate the effectiveness of LLMs in removing ML cloud misuse occurrences in ML service-based systems. We assess four different LLMs on a representative sample of ML service-based systems. Refactoring correctness is validated manually by two annotators with ML engineering expertise, and the results are analyzed in terms of both accuracy and execution time. Our results show that **GPT (gpt-oss-20b)** achieves the highest refactoring accuracy and provides the fastest execution times.

4.1 Approach

This section introduces our proposed approach for LLM-based refactoring of ML service misuses. The approach mainly consists of a preprocessing pipeline and a carefully designed prompt construction process. As shown in Figure 4.1, once a misuse is detected by MLmisFinder, the code of the ML service-based system to be analyzed is passed to the preprocessing pipeline before being fed to the LLMs for refactoring.

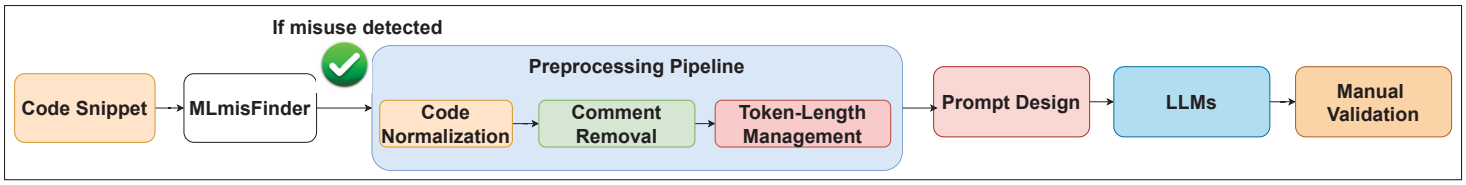


Figure 4.1 Refactoring Approach

4.1.1 Preprocessing Pipeline

Before sending any source code to the LLM, we apply a preprocessing pipeline to ensure consistency, and compatibility with token limits. The preprocessing involves the following steps:

- **Code Normalization:** We normalize indentation and whitespace to a consistent style, remove trailing spaces and redundant blank lines, and standardize line endings to prevent model-specific interpretation differences.
- **Comment Removal:** Non-essential comments, such as inline developer notes, commented-out experimental code, debugging statements, and auto-generated documentation blocks, are stripped from the snippet. Only comments necessary for understanding the misuse, if any, are preserved.
- **Token-Length Management:** To avoid prompt truncation and degraded output quality, we first estimate the token count for each combined prompt and snippet. If a snippet risks exceeding a model's maximum token window, we retain only the minimal code necessary to preserve the misuse and document the reduction process to ensure semantic equivalence.

Each cleaned code snippet then becomes the final input to the standardized prompt template used across all LLMs.

4.1.2 Prompt Design

To maximize reliability and enforce structural consistency, we adopt a zero-shot prompt template applied uniformly across all LLMs in our experiments (Listing 4.1). This template is designed to ensure minimal-change refactoring, meaning that the models are required to fix the misuse

Listing 4.1: Prompt Template

```
You are a senior software engineer specializing in code quality, refactoring, and design
    patterns.
I will provide you with a code snippet that contains the "{misuse_name}" misuse.
Misuse Description:
{misuse_description}
Task:
Refactor the code to fix the misuse using the same Machine Learning (ML) service used in the
    original code (e.g., Azure ML, Amazon SageMaker, or Google Vertex AI).
Important Instructions:
- Do NOT change variable names.
- Do NOT merge, reorder, or restructure the code.
- Do NOT change formatting, comments, or any code unrelated to the misuse.
- Fix the misuse using the ML service native functionalities when applicable (e.g., early
    stopping, checkpointing, logging, monitoring).
- Make only the minimum required changes to fix the misuse.
- Preserve the same ML service or SDK as in the original snippet.
Code Snippet:
{code_snippet}
Response Format:
Refactored Code:
[Provide the fully refactored code here, keeping all original structure intact]
Summary of Changes:
[Provide a concise bullet-point list describing exactly what was changed to fix the misuse]
```

without renaming variables, restructuring code, or modifying formatting. It also enforces ML service fidelity, ensuring that any fixes use the same service originally called in the snippet, thereby maintaining compatibility with the intended cloud environment. The template explicitly forbids the invention of fictional APIs or helper methods, which reduces hallucinations that could break the code. Furthermore, it requires that all changes are clearly documented in a summary, so that every modification made by the model can be traced and verified.

Importantly, the template is designed to support scalability: all misuse names and descriptions are stored in a JSON file with the format `{misuse_name: description}`, which allows developers to easily add new misuses without modifying the prompt structure or the LLM workflow.

Overall, the template is intentionally restrictive to avoid common failure cases, such as rewriting the snippet entirely or addressing the wrong problem, thereby guiding the LLMs to focus precisely on the targeted misuse.

4.2 Empirical Evaluation

In this section, we will describe the research questions and the experimental setup that guide our empirical evaluation.

4.2.1 Research Questions

We aim in this study to answer the following research questions:

- ***RQ1. Can modern LLMs accurately refactor ML service misuses with minimal disruption to the original code?*** We aim to investigate whether LLMs can precisely fix subtle misuses in ML service calls while making minimal unintended changes or introducing hallucinated functionality. By doing so, we can quantify the reliability of LLM-assisted refactoring in real-world ML projects.
- ***RQ2. How do different LLMs compare in execution time for ML service misuse refactoring?*** Our goal is to compare multiple LLMs, which differ in model size, training data, deployment strategy, and code specialization, to evaluate trade-offs between accuracy, speed, and adherence to minimal-change principles. Execution time is measured across all 107 projects in the dataset. This comparison helps guide developers in selecting the most suitable model for automated ML service misuse repair.

4.2.2 Experimental Setup

In this section, we present our experimental setup. We describe the dataset used to evaluate the refactoring approach and introduce the LLMs included in our study. We also outline the manual evaluation procedure conducted by two annotators to assess the accuracy of the refactoring produced by each LLM.

4.2.2.1 Dataset

We use the same manually validated misuse dataset introduced in Section 3.4.3, along with the seven misuses defined in Section 3.2, ensuring consistency with prior analyses. As a reminder,

the seven misuses included in our analysis are: "*Not Using Batch APIs for Data Processing*", "*Not Using Training Checkpoints*", "*Non Specification of Early Stopping Criteria*", "*Ignoring Testing Schema Mismatch*", "*Misinterpreting Output*", "*Improper Handling of ML API Limits*", and "*Ignoring Monitoring for Data Drift*".

Figure 4.2 illustrates the distribution of lines of code and the number of tokens across all code snippets in the dataset, which comprises 107 projects and 340 misuses, as described in Section 3.4.3. The boxplot shows that the median snippet contains approximately 315 lines of code and 2,890 tokens, while the interquartile ranges indicate that most snippets fall between 127–641 lines ($127 \leq \text{LOCs} \leq 641$) and 1,136–6,502 tokens ($1,136 \leq \text{tokens} \leq 6,502$), emphasizing the substantial variability and confirming the diversity of the dataset in terms of both lines of code and token counts. Overall, the figure demonstrates that while most snippets are of moderate size, a few extreme cases deviate significantly from typical snippet sizes.

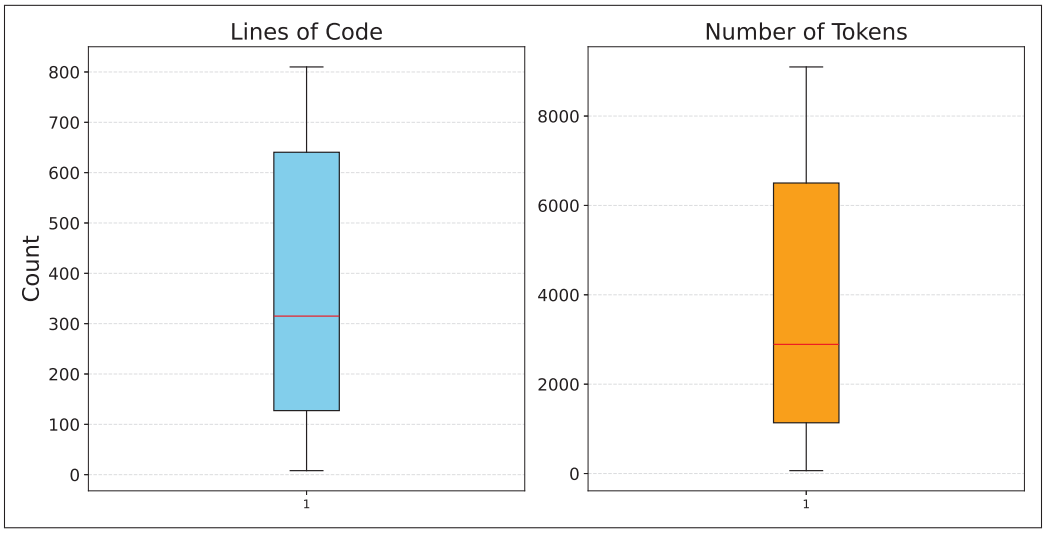


Figure 4.2 Distribution of Lines of Code and Number of Tokens

4.2.2.2 Evaluated Large Language Models

To investigate the capabilities of LLMs in refactoring ML service misuses, we evaluate four modern models that differ in size and training specialization:

- **Llama3 (llama3:8b)** is a 4.7 GB Docker image implementing a causal language model optimized for code completion. It was run locally via Ollama with GPU acceleration and supports up to 8k tokens per prompt.
- **Gemma (gemma:7b)** is a 5.0 GB Docker image, fine-tuned specifically for code refactoring and ML service usage patterns. Like **Llama3**, it was run locally using Ollama with GPU acceleration and supports up to 8k tokens per prompt.
- **GPT (gpt-oss-20b)** is an open-source model with 20 billion parameters, accessed via the `GROQ` API [1]. It is optimized for code generation and reasoning tasks, and can process up to 8k tokens per request. In our experiments, it was used remotely to simulate a cloud-based workflow.
- **Qwen (qwen/qwen3-32b)** is a 32-billion-parameter open-source model, also accessed through the `GROQ` API. It is specialized for code understanding and refactoring tasks, with a token limit of 8k per request, and was similarly used in a remote cloud setup.

All models are tested under identical conditions, using the same misuse instances, prompt template, and evaluation metrics. Specifically, **Gemma** and **Llama3** were run locally using the `Ollama`[2] platform on a standard workstation equipped with an Intel Core i5-1135G7 CPU (4 cores, up to 4.2 GHz), 16 GB RAM, and dual GPUs (Integrated Intel Iris Xe Graphics and NVIDIA GeForce MX450), ensuring sufficient computational resources for local inference. This allowed for full control over token limits, batching, and prompt handling. In contrast, **GPT-oss-20b** and **Qwen3-32b** were accessed remotely via the `GROQ` API, using standard endpoints for code completion. API calls were made programmatically with consistent prompt formatting and input preprocessing to ensure comparability with the locally run models.

This setup ensures that all models are evaluated fairly, while also reflecting practical usage scenarios for both local and cloud-based LLM deployments.

[1] https://console.groq.com/home

[2] https://ollama.com

4.2.2.3 Ground Truth

Running the projects directly on the cloud (such as Azure ML, Amazon SageMaker, or Google Vertex AI) for refactoring evaluation proved to be impractical. Doing so would require access to each project's private datasets, extensive credentials, and significant manual effort to debug deployment and runtime issues across all projects using different cloud providers. Additionally, the cost and time associated with spinning up resources for every misuse instance to assess their impact on the system's latency and cloud usage cost, for example, before and after the refactoring would be prohibitive. Therefore, instead of executing the projects in the cloud, we rely on *manual code inspection of the LLM-assisted refactoring* on preprocessed code snippets to evaluate correctness. Given that reviewing all outputs for the 107 projects (340 misuses, Section 3.4.3) across the four LLMs is practically infeasible, we focus on evaluating a statistically representative sample of the misuses. To ensure that this subset reliably reflects the full dataset, we adopt a **confidence level** of 95% and a **margin of error** of ±10%, leading to a total of 76 refactoring instances per model to be manually inspected (a total of 304 refactoring instances).

Two annotators (a PhD and a master student) manually and independently evaluated each sampled LLM-generated refactoring. A refactoring is considered accurate if it satisfies the following criteria: **(1)** the misuse is fully addressed, **(2)** no unrelated code edits are introduced, such as formatting changes, reordering, merging, or renaming, **(3)** only the minimal necessary changes are applied, **(4)** no hallucinated functionality or invented methods are added, and **(5)** the summary of changes provided by the model accurately matches the actual modifications in the code. This ensures statistically meaningful results while keeping the manual validation effort manageable. We used Cohen's kappa coefficient to measure agreement (Banerjee *et al.*, 1999), and any disagreements were resolved through discussion. For this manual evaluation, the two annotators achieved substantial agreement, with a Cohen's kappa of 98%.

4.3 Results

We will describe in this section our empirical results.

4.3.1 RQ1: Can modern LLMs accurately refactor ML service misuses with minimal disruption to the original code?

Across all misuses, we found notable differences in refactoring accuracy and behavior. As shown in Table 4.1, **GPT** has an accuracy of 82%, and consistently performs best in 58% of the misuses. Table 4.2 confirms that **GPT** ranks as the best-performing model for six of the seven identified ML service misuses. It strikes the strongest balance across several key aspects. First, it reliably uses only real, documented ML service APIs, ensuring that all refactoring remain compatible with the original service calls. Second, it usually produces refactored code snippets that can be integrated directly into projects without additional modifications. Third, it adheres strictly to minimal-edit constraints, making only the necessary changes to fix the misuse. In addition, it usually avoids introducing unnecessary abstractions and does not invent fictional endpoints, functions, or parameters. Finally, it consistently interprets the misuse correctly, addressing the intended problem without solving unrelated issues.

Qwen demonstrates strong performance in automated ML service refactoring, achieving an accuracy of 67% (Table 4.1) and effectively addressing complex issues such as monitoring, schema validation, batch processing, and rate-limit handling. It outperforms other models in refactoring 37% of the misuses, particularly excelling in scenarios involving the "*Ignoring Testing Schema Mismatch*" misuse (Table 4.2). It correctly identified the tasks' requirements and applied minimal, targeted changes in the right locations, preserving original code structure and output formats. **Qwen** integrates seamlessly with native ecosystems, including Azure Application Insights, AWS S3/SNS, and Google or Twitter APIs, enabling robust monitoring, telemetry collection, and drift detection. The model implemented proper batch processing by collecting data into lists, making single batch API calls, and mapping results back to the original files. For rate-limit handling, **Qwen** caught specific exceptions while logging relevant metrics for later analysis. There were also a few minor library mistakes and some hard-coded schemas.

Other models exhibit characteristic limitations that highlight the challenges of automated refactoring (Table 4.1). For example, **Llama3** achieved an accuracy of only 2.6% and was the best-performing model in just a single case out of 76 (1.3%). While it is capable of performing

minor syntax improvements and code formatting, it frequently misinterprets domain-specific ML tasks, such as data drift monitoring, schema validation, batch processing improvements, or API rate-limit handling, as generic software refactoring not ML service specific. As a result, its outputs are often superficial or irrelevant, providing trivial basic formatting fixes (e.g., PEP 8-style changes) while ignoring the actual engineering requirements. It also demonstrates hallucinations and contextual errors, sometimes claiming the system uses one cloud provider while it relies on another, or generating generic tutorial scripts instead of modifying the user's code. Critically, it does not implement essential monitoring or validation mechanisms, nor does it handle API limits or batch processing. These limitations can lead to outputs that misleadingly appear to solve problems while failing to address performance, or reliability, highlighting the risks of relying on automated refactoring for complex ML engineering tasks.

Gemma also demonstrates hallucinations and generates fictional APIs, inventing methods and attributes that do not exist in real SDKs, which can create false confidence in its outputs (Table 4.1). It achieved an accuracy of 1.3% and was the best-performing model in only one of the evaluated cases (1.3%). Furthermore, it fails to handle API limits and batch processing, and it sometimes produces incomplete or internally confused outputs, showing that it cannot reliably reason about ML services. Overall, while **Gemma** can generate syntactically plausible code, its outputs are mostly irrelevant, misleading, and unsafe to rely on for real ML system refactoring.

Notably, two of the evaluated refactoring cases (2.4%) could not be solved by any of the four models, highlighting fundamental limitations of current LLM-based refactoring for ML services.

Table 4.1 Accuracy of LLMs

Model	GPT	Qwen	Llama3	Gemma
Accuracy (%)	82	67	2.6	1.3

Table 4.2 Best-performing LLM for each ML service misuse

Misuse	Best Model
Ignoring Monitoring Data Drift	GPT
Ignoring Testing Schema Mismatch	Qwen
Improper Handling Of ML API Limits	GPT
Misinterpreting Output	GPT
Non Specification Of Early Stopping Criteria	GPT
Not Using Batch API For Data Processing	GPT
Not Using Training Checkpoints	GPT

4.3.2 RQ2: How Do Different LLMs Compare in Execution Time for ML Service Misuse Refactoring?

Table 4.3 clearly demonstrates that execution time varies significantly across LLMs depending on both the model and the specific misuse scenario. The **GPT** model consistently exhibits the lowest execution time, indicating it is the fastest model across all misuse categories. Conversely, the **Gemma** and **Llama3** models have the highest execution times across the board, running approximately 100 times slower than **GPT**'s. The **Qwen** model generally falls between **GPT** and the others, displaying moderate to high execution times, though substantially lower than **Gemma** and **Llama3** in most cases.

The observed differences in execution time can be attributed in part to deployment: **Gemma** and **Llama3** were run locally via Ollama, which may introduce additional overhead due to local resource constraints and GPU management. In contrast, **Qwen** and **GPT** were accessed remotely through the `GROQ` API, allowing faster responses as the models run on optimized cloud infrastructure. While remote access provides low-latency inference and eliminates the need for local computational resources, it can also be more costly, especially for large-scale or frequent usage. Additionally, reliance on cloud services introduces potential limitations such as network dependency, and API rate limits. Thus, while the approach accelerates experimentation and reduces hardware requirements, it requires balancing performance gains against operational costs and usage constraints.

The analysis of average execution time per misuse for each model highlights significant differences in performance across both models and misuse categories (Table 4.3). The **GPT** model exhibits the lowest execution times across all misuses, ranging from approximately 2 seconds for "*Misinterpreting Output*" to 4.59 seconds for "*Ignoring Monitoring Data Drift*", with an overall average of 3.86 seconds. This demonstrates its consistently fast response regardless of the misuse type. The **Gemma** model, in contrast, shows substantially higher execution times, varying between 242.7 seconds ("*Not Using Batch API for Data Processing*") and 632.8 seconds ("*Not Using Training Checkpoints*"), with an overall average of 449.86 seconds. Execution times remain consistently high across all misuses, reflecting the overhead of running **Gemma** locally via Ollama.

Llama3 exhibits the highest latency overall, with times ranging from 247.8 seconds ("*Non Specification of Early Stopping Criteria*") to 669.9 seconds ("*Not Using Training Checkpoints*"), resulting in an overall average of 478.66 seconds. Similar to **Gemma**, **Llama3**'s high execution times can be attributed to its local execution setup, which likely introduces additional resource overhead, particularly for more complex misuses. The **Qwen** model demonstrates moderate performance, with execution times between 5.5 seconds ("*Non Specification of Early Stopping Criteria*") and 13.5 seconds ("*Ignoring Monitoring Data Drift*"), averaging 10.13 seconds overall. While slower than **GPT**, **Qwen** is still considerably faster than **Gemma** and **Llama3** due to its deployment through the `GROQ` API, leveraging optimized cloud infrastructure. Across all models, misuses such as "*Ignoring Testing Schema Mismatch*" and "*Not Using Training Checkpoints*" tend to require the longest execution times for the slower models, suggesting that these misuses may involve more complex code patterns or API interactions that increase processing cost. In contrast, misuses like "*Misinterpreting Output*" generally require less time, indicating simpler detection and refactoring steps for the models.

Overall, these results illustrate a clear trade-off between model deployment and execution speed: locally run models (**Gemma** and **Llama3**) incur significant latency, while cloud-accessed models (**GPT** and **Qwen**) achieve substantially faster response times. Importantly, execution speed

alone does not imply accuracy or robustness, emphasizing that both latency and refactoring quality must be considered when selecting a model for ML service misuse repair.

Table 4.3 Execution time of LLMs for different ML service misuses (in seconds)

Misuse	GPT	Gemma	Llama3	Qwen
Ignoring Monitoring Data Drift	4.59	280.26	603.56	13.46
Ignoring Testing Schema Mismatch	4.41	593.48	610.20	13.16
Improper Handling Of ML API Limits	3.83	589.02	631.33	11.18
Misinterpreting Output	2.01	326.99	320.88	7.17
Non Specification Of Early Stopping Criteria	4.47	483.78	247.77	5.52
Not Using Batch API For Data Processing	3.22	242.68	267.04	10.90
Not Using Training Checkpoints	4.55	632.83	669.90	9.56
Average (sec)	3.86	449.86	478.66	10.13

4.4 Discussion

Our empirical evaluation demonstrates that LLMs are capable of refactoring ML service misuses with varying degrees of accuracy, speed, and adherence to minimal-change principles. While some models perform exact refactoring, others may introduce unintended edits or hallucinated code, highlighting the importance of careful model selection and prompt design. These results have practical implications for both software developers and researchers.

- **Implications for Developers.** Our findings suggest that some LLMs can serve as powerful tools for assisting the refactoring of ML service misuses. Developers can leverage these models to automatically refactor ML service misuses, improving code quality and reducing manual effort. However, caution is necessary: hallucinations, where a model invents non-existent APIs or behaviors, remain a risk, and not all large models guarantee higher accuracy. Beyond accuracy, developers must consider practical factors such as cost, deployment complexity, maintenance burden, and data privacy when integrating LLMs into their workflows. Additionally, thoughtful adoption as part of CI/CD pipelines is essential to ensure that model-driven refactoring is both safe and seamlessly integrated into existing

development processes. Strict prompt design and careful validation of outputs remain critical to ensuring reliable and trustworthy refactoring.

- **Implications for Researchers.** From a research perspective, our results highlight several avenues for future work. Integrating automated static analysis checks to validate LLM-generated repairs could improve reliability and reduce manual verification effort. Incorporating additional evaluation quality metrics, such as hallucination rate, and P@k (e.g., $k = 1$ means checking whether the top suggestion is correct), would allow more comprehensive assessment of model performance. It is also important to experiment with one-shot and few-shot prompting, since our current study focuses on zero-shot prompts. Such experiments could help determine whether providing minimal context examples improves the model's understanding of ML service misuses and leads to more precise or efficient code refactorings. Moreover, understanding the interplay between model size, deployment environment, and prompt design remains a critical area to optimize LLM-assisted code repair across diverse ML services and project contexts.

4.5 Threats To Validity

Internal validity. The assessment of LLM-refactored code was performed manually, which inherently requires expertise across a wide range of ML services, cloud provider APIs, and SDKs. This manual evaluation ensures careful consideration of both accuracy and potential unintended changes introduced by the model. However, relying on evaluators judgment introduces some subjectivity, as different evaluators might interpret code modifications slightly differently. To mitigate this, multiple evaluators independently reviewed each refactoring, and any disagreements were resolved through discussion. While this process improves reliability, the results may still be influenced by the evaluators' experience and familiarity with specific ML services and frameworks.

Construct Validity. Concerns whether the measurements used in this study accurately capture the intended concepts. In our experiments, execution time may not perfectly reflect model

efficiency, as it can be influenced by the local hardware, system load, or GPU/CPU configurations, particularly for models like Ollama that run locally.

External validity. While our analysis focuses on the seven misuses defined in our dataset, an important question is whether, and how, LLMs would perform when confronted with additional or emerging misuse types not explicitly represented in our ground truth. The misuses included in this study span data processing, training configuration, evaluation, and system-level API usage, but they do not exhaust the broader landscape of ML development pitfalls. Some misuses, such as security-related vulnerabilities, fairness issues, or misconfigurations specific to particular frameworks, may require different forms of contextual reasoning or domain knowledge. It is therefore unclear whether LLMs would generalize effectively or whether their performance would degrade when the misuse relies on subtle domain semantics that are underrepresented or inconsistently documented in their training data. Exploring LLM behavior on a wider range of misuse categories, and understanding which characteristics make a misuse more or less detectable, constitutes an important direction for future work. This broader perspective would help establish whether LLM-based analysis can serve as a reliable, comprehensive aid for practitioners, or whether its applicability is limited to certain types of misuses.

Conclusion

This chapter demonstrates that LLMs can effectively refactor ML service misuses at scale when guided by a carefully designed prompt template and evaluated against a meticulously validated dataset. The quality of these evaluations is further reinforced by a manual review conducted by two annotators, achieving a Cohen's kappa of 98%, which attests to the reliability and consistency of the assessment. Among the evaluated models, **GPT** was the most accurate (82%) and the fastest model with an average execution time of 3.86 seconds per refactoring.

CONCLUSION AND RECOMMENDATIONS

This project addressed a critical and largely unexplored problem in modern software engineering: the misuses of ML cloud services. While ML services are increasingly adopted due to their scalability, accessibility, and reduced development effort, improper usage introduces significant risks related to ML cloud service-based systems performance, and maintainability.

Existing research had mainly focused on general code smells, ML anti-patterns, and traditional software misuses, leaving ML service misuses largely undocumented. To fill this gap, this project delivered a comprehensive contribution covering specification, automated detection, and automated refactoring of ML service misuses. First, a systematic review of the literature established that no prior study had thoroughly investigated ML service misuses as a distinct class of software anti-patterns. This finding motivated the need for a dedicated empirical investigation. Through a large-scale multi-vocal empirical study combining academic literature, gray literature, and real-world GitHub projects, we proposed a validated catalog of 20 ML service misuses. Each misuse was precisely defined, contextualized within the ML development lifecycle, and analyzed in terms of its impact. The practitioner survey confirmed both the practical relevance and high prevalence of these misuses in industry. Moreover, the study highlighted the root causes of these misuses, including lack of ML expertise, poor documentation, and rapid evolving cloud APIs.

Second, this project introduced MLmisFinder, the first fully automated static analysis approach dedicated to detecting seven ML service misuses. Leveraging a reusable metamodel and rule-based detection algorithms, MLmisFinder achieved a precision of 96.7% and a recall of 97%, significantly outperforming the only existing baseline. The large-scale analysis of 817 GitHub systems further revealed that ML service misuses are widespread, especially in critical areas such as data drift monitoring and schema validation.

Finally, this project demonstrated that LLMs can effectively automate the refactoring of ML service misuses. Using a carefully engineered prompt and a rigorously validated dataset, the study showed that LLMs, particularly **GPT**, can generate accurate, and efficient refactoring solutions. Taken together, this work provides the first complete lifecycle solution for ML service misuses: from specification and empirical grounding, to automated detection and refactoring using LLMs.

Building on the findings and limitations of this project, several promising research directions emerge for future investigation. An important avenue is to explicitly model ML service misuses as a distinct form of ML-specific technical debt. Unlike traditional code smells, these misuses can affect not only software quality attributes such as maintainability and reliability, but also model-related properties including accuracy, fairness, robustness, and operational cost. Developing quantitative metrics to capture the accumulation, severity, and long-term impact of ML service technical debt would enable more systematic management and prioritization of remediation efforts.

While this project identified and validated a catalog of 20 ML service misuses, future studies should explore additional sources such as Stack Overflow discussions. Such sources can reveal emerging misuses that may not yet be visible in open-source repositories. This line of work would support the construction of a continuously evolving, community-driven knowledge base that reflects the rapid evolution of ML cloud services and their usage patterns.

Although MLmisFinder achieves high accuracy through static analysis, certain classes of misuses, including monitoring data drift, and schema validation, may require dynamic or hybrid analysis techniques. Integrating runtime monitoring, execution logs, and telemetry data with static analysis represents an important next step toward more comprehensive misuse detection. Such a hybrid approach would enable the identification of misuses that only manifest under specific workloads, data distributions, or deployment conditions.

The success of LLM-based refactoring demonstrated in this project opens the door for fully autonomous ML service maintenance pipelines, in which detected misuses are automatically refactored, verified, and integrated through CI/CD workflows. Future research should explore the design of LLM pipelines, the integration of automated regression testing for LLM-generated refactorings, and the development of human-in-the-loop validation frameworks that balance automation with developer oversight. Despite their effectiveness, LLMs introduce new risks, including hallucinated transformations, inconsistent code changes, security vulnerabilities, and high operational costs. Future work should therefore focus on cost-aware refactoring strategies, the use of model ensembles to improve reliability, and secure prompt engineering techniques suitable for enterprise and safety-critical ML service-based systems. Addressing these challenges is essential for establishing trust in fully automated ML service refactoring.

This project establishes ML service misuses as a first-class research problem in software engineering and demonstrates that combining literature review, static analysis, and LLM-based automation can deliver practical, scalable, and high-impact solutions to address ML service misuses. As cloud-based AI systems continue to expand across safety-critical and business-critical domains, the automated detection and refactoring of ML service misuses will become a foundational pillar for building robust, trustworthy, and sustainable ML service-based systems.

APPENDIX I

SUMMARY OF COLLECTED STUDIES IN THE ACADEMIC LITERATURE REVIEW

Table-A I-1 Summary of the Collected Studies in our Literature Review

Study	Year	Summary
ML Service-based Systems		
Are Machine Learning Cloud APIs Used Correctly? (Wan *et al.*, 2021)	2021	Investigates errors in the use of Google and AWS ML cloud APIs that can degrade functionality, performance, or cost efficiency. The study identifies **8 common antipatterns and misuse errors**.
ML-based Systems (Bad Practices)		
A Survey on Bias and Fairness in Machine Learning (Mehrabi *et al.*, 2021)	2021	This paper surveys **bias and fairness in machine learning**, covering fairness definitions, domains affected by bias (e.g., classification, regression, NLP), and mitigation algorithms. It highlights the transition from equality to the less explored notion of equity.
MATCHMAKER: Data Drift Mitigation in Machine Learning for Large-Scale Systems (Mallick *et al.*, 2022)	2022	The document discusses strategies for mitigating data drift in large-scale ML systems. However, the specific conclusions are not provided in the available text, focusing instead on references to methods and techniques related to **data drift management**.

Continued on next page

Table-A I-1 – *Continued from previous page*

Study	Year	Summary
Code Smells for Machine Learning Applications (Zhang *et al.*, 2022)	2022	Compiles a catalog of ML-specific code smells by analyzing existing literature, GitHub commits, and Stack Overflow discussions. The study documents **22 ML-specific code smells**.
23 Shades of self-admitted technical debt: an empirical study on machine learning software (OBrien *et al.*, 2022b)	2022	Analyzes comments related to Self-Admitted Technical Debt (SATD) in open-source ML systems, classifying both software and ML-specific SATD types. The study introduces **8 new ML SATD groups**.
Automated Testing of Software that Uses Machine Learning APIs (Wan *et al.*, 2022)	2022	This paper introduces Keeper, **a coverage-guided automated testing framework that detects bugs and suggests fixes in software using ML APIs.** It generates test cases via symbolic execution and ML inverse functions, leveraging search engines and synthesis techniques.
Collaboration Challenges in Building ML-Enabled Systems: Communication, Documentation, Engineering, and Process (Nahar *et al.*, 2022)	2022	This paper discusses **collaboration challenges in AI/ML software projects** based on a literature review, focusing on collaboration issues raised during interviews.

Continued on next page

Table-A I-1 – *Continued from previous page*

Study	Year	Summary
"Project smells" — Experiences in Analysing the Software Quality of ML Projects with mllint (Van Oort *et al.*, 2022)	2022	This paper introduces ***"project smells"* as a holistic perspective on ML software quality** and presents mllint, a static analysis tool for detecting and mitigating these smells.
Understanding Performance Problems in Deep Learning Systems (Cao *et al.*, 2022)	2022	This paper presents an in-depth study of **performance problems in DL systems using TensorFlow and Keras**. It analyzes symptoms, root causes, and bug-prone stages, and introduces a static checker, *DeepPerf*, to detect new issues.
Data Leakage in Notebooks: Static Detection and Better Processes (Yang, Brower-Sinning, Lewis & Kästner, 2022a)	2022	This paper outlines common **data leakage issues in ML** and proposes a static analysis approach to detect them automatically. It includes a large-scale study of public notebooks and offers process design recommendations for prevention.
Compatibility Issues in Deep Learning Systems: Problems and Opportunities (Wang *et al.*, 2023)	2023	Characterizes compatibility issues in DL systems and provides a taxonomy of common problems and resolution strategies. The study identifies **9 types of compatibility issues**.

Continued on next page

Table-A I-1 – *Continued from previous page*

Study	Year	Summary
Pitfalls in language models for code intelligence: A taxonomy and survey (She *et al.*, 2023)	2023	Reviews common pitfalls encountered in language models designed for code intelligence tasks. The study introduces a **taxonomy of 13 pitfalls**.
Demystifying and Detecting Misuses of Deep Learning APIs (Wei *et al.*, 2024)	2024	Conducts an empirical study on the misuse of DL APIs in PyTorch and TensorFlow. It proposes a misuse detector based on Large Language Models (LLMs) and defines **three categorization dimensions of DL API misuse**, identifying **12 common types of DL API misuse**.
Machine Learning Systems are Bloated and Vulnerable (Zhang *et al.*, 2024)	2024	This paper investigates **bloat in ML container images**, focusing on TensorFlow and PyTorch. It analyzes **how reducing bloat affects execution performance and security vulnerabilities**, providing insights on **optimizing ML deployments** by removing unnecessary components.
ML-based Systems (Best Practices)		

Continued on next page

Table-A I-1 – *Continued from previous page*

Study	Year	Summary
Fairness and Transparency of Machine Learning for Trustworthy Cloud Services (Antunes *et al.*, 2018)	2018	This paper discusses the importance of **fairness and transparency in ML, particularly for cloud services**. It presents future research directions for ensuring fair and transparent ML applications in trusted cloud environments, with an emphasis on the ATMOSPHERE ecosystem.
Data Validation for Machine Learning (Polyzotis *et al.*, 2019)	2019	This study outlines a **data validation system used at Google, demonstrating its impact on improving ML model performance**. A case study shows how identifying data divergences between training and production data led to a 2% improvement in app installation rates.
Studying Software Engineering Patterns for Designing Machine Learning Systems (Washizaki *et al.*, 2019)	2019	This study investigates the presence of software engineering patterns for ML systems, identifying **33 relevant patterns** through a systematic review of research and gray literature. These patterns apply to various stages of the ML development and deployment process.

Continued on next page

Table-A I-1 – *Continued from previous page*

Study	Year	Summary
Adoption and Effects of Software Engineering Best Practices in ML (Serban *et al.*, 2020)	2020	Compiles a catalog of best practices in ML engineering and surveys their adoption and impact. The study identifies **29 engineering best practices** applicable to ML applications.
Practitioners' insights on machine-learning software engineering design patterns: a preliminary study (Washizaki *et al.*, 2020)	2020	Reports findings from a literature review and survey on ML development practices. The study identifies **15 ML design patterns** adopted in practice.
Sensemaking Practices in the Everyday Work of AI/ML Software Engineering (Wolf & Paine, 2020)	2020	This study presents findings from an ethnographic field study in a large tech company, focusing on **sensemaking practices in AI/ML software development projects**, analyzed through structured interviews, participant observation, and artifact analysis.
Amazon SageMaker Clarify: Machine Learning Bias Detection and Explainability in the Cloud (Hardt *et al.*, 2021)	2021	This paper discusses **the implementation and technical challenges of Clarify, a SageMaker Studio tool for bias detection and model explainability**. It reports on its deployment, scalability evaluation, customer feedback, and use cases.

Continued on next page

Table-A I-1 – *Continued from previous page*

Study	Year	Summary
Scheduling ML Training on Unreliable Spot Instances (Yang, Khuller, Choudhary, Mitra & Mahadik, 2021)	2021	Proposes a cost-effective scheduling strategy for ML training on interruptible spot instances, achieving near-optimal performance at just 23–48% of the cost of on-demand instances.
Discovering Repetitive Code Changes in Python ML Systems (Dilhara, Ketkar, Sannidhi & Dig, 2022)	2022	Analyzes patterns of repetitive code changes in Python-based ML systems. The study identifies **22 recurring modification patterns** in ML code.
Software Engineering Design Patterns for ML Applications (Washizaki *et al.*, 2022)	2022	Conducts a literature review on software engineering design patterns for ML applications. The study identifies and reviews **15 design patterns** used in ML software engineering.
Checkpointing and Deterministic Training for Deep Learning (Xu *et al.*, 2022)	2022	This study addresses **checkpointing techniques and the importance of deterministic training in DL**, especially in the context of long training times and hardware failures.

Continued on next page

Table-A I-1 – *Continued from previous page*

Study	Year	Summary
Responsible AI Pattern Catalog: A Collection of Best Practices for AI Governance and Engineering (Lu *et al.*, 2024)	2024	This paper **introduces a catalog of Responsible AI design patterns intended to help teams build AI systems that align with principles such as fairness, accountability, transparency, and ethics.** The patterns are derived from empirical studies and industry best practices and are meant to be reusable solutions for common challenges in trustworthy AI development.
How do Machine Learning Projects use Continuous Integration Practices? An Empirical Study on GitHub Actions (Bernardo, Da Costa, Medeiros & Kulesza, 2024)	2024	This empirical study compares **Continuous Integration (CI) practices in ML and non-ML GitHub projects.** It shows that ML projects tend to have longer build times and a higher incidence of false positives in CI systems, highlighting challenges specific to ML workflows.
Investigating the Impact of SOLID Design Principles on Machine Learning Code Understanding (Cabral *et al.*, 2024)	2024	This study explores **the impact of SOLID design principles on the readability of ML code by data scientists**. It shows that applying SOLID principles improves code comprehension, with the Single Responsibility Principle having the most significant effect.

Continued on next page

Table-A I-1 – *Continued from previous page*

Study	Year	Summary
Identifying architectural design decisions for achieving green ML serving (Durán, Martínez-Fernández, Martinez & Lago, 2024)	2024	This paper examines **architectural decisions in green ML serving systems, particularly focusing on energy efficiency**. The study identifies performance efficiency as the most studied quality attribute, suggesting a need for more research on energy-efficient ML architectures.
Traditional Software Systems		
API Misuse Detection Method Based on Transformer (Yang, Ren & Wu, 2022b)	2022	This paper proposes a transformer-based method for **detecting API misuse**, comparing its performance with existing methods such as n-grams and MuDetect. The transformer models show improved performance in detecting API misuse, especially in recall.
A Systematic Review on Software Design Patterns (Rahman, Chy & Saha, 2023)	2023	Conducts a systematic review of software design patterns, their applications, recognition through ML techniques, and their impact on software quality. The study identifies **23 design patterns** categorized into three groups.

APPENDIX II

SUMMARY OF THE COLLECTED STUDIES IN THE GRAY LITERATURE REVIEW

Table-A II-1 Summary of the Collected Studies in the Gray Literature Review

Study	Year	Summary
Importance of Tuning Hyperparameters of Machine Learning Algorithms (Weerts *et al.*, 2020)	2020	This work presents a procedure for determining default hyperparameter values and a methodology to evaluate the **importance of hyperparameter tuning**. It includes benchmark experiments across datasets, algorithms, hyperparameters, metrics, and search strategies.
A Study of Checkpointing in Large Scale Training of Deep Neural Networks (Rojas *et al.*, 2020)	2020	This study **evaluates checkpointing implementations in three DL frameworks** (Chainer, TensorFlow, and PyTorch) in high-performance computing environments, considering computational costs, file sizes, scaling, and deterministic behavior.
Insights into Performance Fitness and Error Metrics for Machine Learning (Naser & Alavi, 2020)	2020	This paper discusses the challenges in selecting appropriate performance evaluation metrics for machine learning models.

Continued on next page

Table-A II-1 – *Continued from previous page*

Study	Year	Summary
An empirical study of self-admitted technical debt in machine learning software (Bhatia, Khomh, Adams & Hassan, 2023)	2023	Investigates SATD in ML systems and identifies ML-specific SATD types. The study introduces **2 new types of SATD: Configuration debt and Inadequate tests**.
Prevalence of Code Smells in Reinforcement Learning (Cardozo *et al.*, 2023)	2023	Investigates the presence of code smells in reinforcement learning (RL) projects. The study identifies **8 code smells** related to code organization, abstraction, and expression concerns.
A Meta-Summary of Challenges in Building Products with ML Components – Collecting Experiences from 4758+ Practitioners (Nahar *et al.*, 2023)	2023	This meta-summary synthesizes **the challenges faced by practitioners in building products with ML components**, based on surveys of over 4758 practitioners. It highlights issues related to specification, quality assurance, and tool support throughout the ML product development lifecycle.
Quality assurance for artificial intelligence: A study of industrial concerns, challenges and best practices (Wang *et al.*, 2024)	2024	Explores quality assurance practices for AI systems (QA4AI) through expert interviews and validation surveys. The study identifies **21 QA4AI best practices** applicable across different stages of AI development.

Continued on next page

Table-A II-1 – *Continued from previous page*

Study	Year	Summary
Data Quality Antipatterns for Software Analytics (Bhatia, Lin, Rajbahadur, Adams & Hassan, 2024)	2024	Examines how ML-specific data quality antipatterns affect software defect prediction models. **9** antipatterns were identified, with overlapping issues complicating cleaning priorities. The order of cleaning significantly impacts model performance, and certain antipatterns, are statistically correlated with performance.

LIST OF REFERENCES

(2025). MLmisFinder. Retrieved from: https://anonymous.4open.science/r/MLmisFinder.

Ali Abd Al-Hameed, K. (2022). Spearman's correlation coefficient in statistical analysis. *International Journal of Nonlinear Analysis and Applications*, 13(1), 3249–3255.

Amazon Web Services. (2024a). Autoscaling. Retrieved from: https://docs.aws.amazon.com/autoscaling/application/userguide/what-is-application-auto-scaling.html.

Amazon Web Services. (2024b). Evaluation Alerts. Retrieved on 2024-06-24 from: https://docs.aws.amazon.com/machine-learning/latest/dg/evaluation-alerts.html.

Antunes, N., Balby, L., Figueiredo, F., Lourenco, N., Meira, W. & Santos, W. (2018). Fairness and transparency of machine learning for trustworthy cloud services. *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pp. 188–193.

API, B. (2024). https://learn.microsoft.com/Batch multiple documents.

Ashurst, C. & Weller, A. (2023). Fairness without demographic data: A survey of approaches. *Proceedings of the 3rd ACM Conference on Equity and Access in Algorithms, Mechanisms, and Optimization*, pp. 1–12.

Banerjee, M., Capozzoli, M., McSweeney, L. & Sinha, D. (1999). Beyond kappa: A review of interrater agreement measures. *Canadian journal of statistics*, 27(1), 3–23.

Ben Amor, H., Abdellatif, M. & Ghaleb, T. A. (2025). A Comprehensive Multi-Vocal Empirical Study of ML Cloud Service Misuses. *arXiv preprint arXiv:2503.09815*.

Ben Amor, H., Abdellatif, M. & Ghaleb, T. A. (2024). Replication Package. Retrieved from: https://github.com/hadil1999-creator/A-Comprehensive-Multi-Vocal-Empirical-Study-of-ML-Cloud-Service-Misuses.

Bernardo, J. H., Da Costa, D. A., Medeiros, S. Q. d. & Kulesza, U. (2024). How do machine learning projects use continuous integration practices? an empirical study on GitHub actions. *Proceedings of the 21st International Conference on Mining Software Repositories*, pp. 665–676.

Bhatia, A., Khomh, F., Adams, B. & Hassan, A. E. (2023). An empirical study of self-admitted technical debt in machine learning software. *arXiv preprint arXiv:2311.12019*.

Bhatia, A., Lin, D., Rajbahadur, G. K., Adams, B. & Hassan, A. E. (2024). Data Quality Antipatterns for Software Analytics. Retrieved from: https://arxiv.org/abs/2408.12560v1.

Bogner, J., Verdecchia, R. & Gerostathopoulos, I. (2021). Characterizing technical debt and antipatterns in AI-based systems: A systematic mapping study. *2021 IEEE/ACM International Conference on Technical Debt (TechDebt)*, pp. 64–73.

Bothmann, L., Peters, K. & Bischl, B. (2022). What is fairness? On the role of protected attributes and fictitious worlds. *arXiv preprint arXiv:2205.09622*.

Bova, C., Jaffarian, C., Crawford, S., Quintos, J., Lee, M. & Sullivan-Bolyai, S. (2017). Intervention fidelity: monitoring drift, providing feedback, and assessing the control condition. *Nursing research*, 66(1), 54–59.

Cabral, R., Kalinowski, M., Baldassarre, M., Villamizar, H., Escovedo, T. & Lopes, H. (2024). Investigating the Impact of SOLID Design Principles on Machine Learning Code Understanding. *Proceedings of the IEEE/ACM 3rd International Conference on AI Engineering-Software Engineering for AI*, pp. 7–17.

Cao, J., Chen, B., Sun, C., Hu, L., Wu, S. & Peng, X. (2022). Understanding performance problems in deep learning systems. *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 357–369.

Cardozo, N., Dusparic, I. & Cabrera, C. (2023). Prevalence of code smells in reinforcement learning projects. *2023 IEEE/ACM 2nd International Conference on AI Engineering–Software Engineering for AI*, pp. 37–42.

Chen, H. (2022). *DATA QUALITY EVALUATION AND IMPROVEMENT*. (Ph.D. thesis, University of North Texas).

Cordeiro, J., Noei, S. & Zou, Y. (2025). LLM-Driven Code Refactoring: Opportunities and Limitations. *2025 IEEE/ACM Second IDE Workshop (IDE)*, pp. 32–36.

Dilhara, M., Ketkar, A., Sannidhi, N. & Dig, D. (2022). Discovering repetitive code changes in Python ML systems. *Proceedings of the 44th international conference on software engineering*, pp. 736–748.

Dong, S., Wang, P. & Abbas, K. (2021). A survey on deep learning and its applications. *Computer Science Review*, 40, 100379.

Durán, F., Martínez-Fernández, S., Martinez, M. & Lago, P. (2024). Identifying architectural design decisions for achieving green ML serving. *Proceedings of the IEEE/ACM 3rd International Conference on AI Engineering-Software Engineering for AI*, pp. 18–23.

Endpoints, I. (2024a). Inference Endpoint. Retrieved from: https://learn.microsoft.com/en-us/azure/machine-learning/concept-endpoints?view=azureml-api-2.

Endpoints, O. (2024b). Autoscaling. Retrieved from: https://learn.microsoft.com/en-us/azure/machine-learning/how-to-autoscale-endpoints?view=azureml-api-2&tabs=cli.

Fowler Jr, F. J. (2013). *Survey research methods*. Sage publications.

Framework, A. W.-A. (2024). Machine Learning. Retrieved from: https://docs.aws.amazon.com/pt_br/wellarchitected/latest/machine-learning-lens/mloe-09.html.

Garousi, V., Felderer, M. & Mäntylä, M. V. (2019). Guidelines for including grey literature and conducting multivocal literature reviews in software engineering. *Information and software technology*, 106, 101–121.

Ghaleb, T. A., Hassan, S. & Zou, Y. (2022). Studying the interplay between the durations and breakages of continuous integration builds. *IEEE Transactions on Software Engineering*, 49(4), 2476–2497.

Ghazi, A. N., Petersen, K., Reddy, S. S. V. R. & Nekkanti, H. (2018). Survey research in software engineering: Problems and mitigation strategies. *IEEE Access*, 7, 24703–24718.

Google Cloud. (2024). Scaling ML Predictions. Retrieved from: https://cloud.google.com/blog/products/ai-machine-learning/scaling-machine-learning-predictions?hl=en.

Google Cloud. [Accessed: 2025-08-11]. (2025). Vertex AI Documentation. Retrieved from: https://cloud.google.com/vertex-ai/docs.

Grigorescu, S., Trasnea, B., Cocias, T. & Macesanu, G. (2020). A survey of deep learning techniques for autonomous driving. *Journal of field robotics*, 37(3), 362–386.

Guo, Q., Cao, J., Xie, X., Liu, S., Li, X., Chen, B. & Peng, X. (2024). Exploring the potential of chatgpt in automated code refinement: An empirical study. *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pp. 1–13.

Hardt, M., Chen, X., Cheng, X., Donini, M., Gelman, J., Gollaprolu, S., He, J., Larroy, P., Liu, X., McCarthy, N. et al. (2021). Amazon sagemaker clarify: Machine learning bias detection and explainability in the cloud. *Proceedings of the 27th ACM SIGKDD conference on knowledge discovery & data mining*, pp. 2974–2983.

Holstein, K., Wortman Vaughan, J., Daumé III, H., Dudik, M. & Wallach, H. (2019). Improving fairness in machine learning systems: What do industry practitioners need? *Proceedings of the 2019 CHI conference on human factors in computing systems*, pp. 1–16.

Hyperparameters. [Accessed: 2024-05-11]. (2024). Hyperparameters. Retrieved from: https://learn.microsoft.com/en-us/azure/machine-learning/how-to-tune-hyperparameters?view=azureml-api-2.

Islam, N. [Accessed: 2025-08-18]. (2024). Mastering the Art of Software Documentation: A Comprehensive Guide for Developers and Tech Professionals. Retrieved from: https://medium.com/@nomannayeem/mastering-the-art-of-software-documentation-a06aa5d7e697.

Kitchenham, B. (2004). Procedures for performing systematic reviews. *Keele, UK, Keele University*, 33(2004), 1–26.

Kokinda, E., Moster, M., Dominic, J. & Rodeghero, P. (2023). Under the bridge: Trolling and the challenges of recruiting software developers for empirical research studies. *2023 IEEE/ACM 45th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, pp. 55–59.

Kuang Piao, Y. C., Carlors Paul, J., Da Silva, L., Moradi Dakhel, A., Hamdaqa, M. & Khomh, F. (2025). Refactoring with LLMs: Bridging Human Expertise and Machine Understanding. *arXiv e-prints*, arXiv–2510.

Kuwajima, H., Yasuoka, H. & Nakae, T. (2020). Engineering problems in machine learning systems. *Machine Learning*, 109(5), 1103–1126.

Kästner, C. & Goues, C. L. [Accessed: 2025-09-22]. (2025). Fairness. Retrieved from: https://mlip-cmu.github.io/book/26-fairness.html.

Lewis, G., Ozkaya, I. & Xu, X. (2021). Software architecture challenges for ml systems. *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 634–638.

Li, T. & Zhang, Y. (2024). Multilingual code refactoring detection based on deep learning. *Expert Systems with Applications*, 258, 125164.

Lu, Q., Zhu, L., Xu, X., Whittle, J., Zowghi, D. & Jacquet, A. (2024). Responsible ai pattern catalogue: A collection of best practices for ai governance and engineering. *ACM Computing Surveys*, 56(7), 1–35.

Lwakatare, L. E., Rånge, E., Crnkovic, I. & Bosch, J. (2021). On the experiences of adopting automated data validation in an industrial machine learning project. *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pp. 248–257.

Madaio, M. A., Stark, L., Wortman Vaughan, J. & Wallach, H. (2020). Co-designing checklists to understand organizational challenges and opportunities around fairness in AI. *Proceedings of the 2020 CHI conference on human factors in computing systems*, pp. 1–14.

Mallick, A., Hsieh, K., Arzani, B. & Joshi, G. (2022). Matchmaker: Data drift mitigation in machine learning for large-scale systems. *Proceedings of Machine Learning and Systems*, 4, 77–94.

Masuda, S., Ono, K., Yasue, T. & Hosokawa, N. (2018). A survey of software quality for machine learning applications. *2018 IEEE International conference on software testing, verification and validation workshops (ICSTW)*, pp. 279–284.

Mehrabi, N., Morstatter, F., Saxena, N., Lerman, K. & Galstyan, A. (2021). A survey on bias and fairness in machine learning. *ACM computing surveys (CSUR)*, 54(6), 1–35.

Microsoft. (2024a). Azure ML Designer. Retrieved from: https://learn.microsoft.com/en-us/azure/machine-learning/concept-designer?view=azureml-api-2.

Microsoft. (2024b). AutoML. Retrieved from: https://learn.microsoft.com/en-us/azure/machine-learning/concept-automated-ml?view=azureml-api-2.

Microsoft. (2024c). AutoML training. Retrieved on 2024-06-15 from: https://learn.microsoft.com/en-us/azure/machine-learning/how-to-configure-auto-train?view=azureml-api-2&tabs=python.

ML, A. [Accessed: 2024-09-07]. (2024a). AutoML. Retrieved from: https://cloud.google.com/automl?hl=fr.

ML, A. [Accessed: 2025-01-15]. (2024b). Tech Community. Retrieved from: https://techcommunity.microsoft.com/t5/ai-machine-learning-blog/train-and-score-hundreds-of-thousands-of-models-in-parallel/ba-p/1547960.

Mohan, M. & Greer, D. (2018). A survey of search-based refactoring for software maintenance. *Journal of Software Engineering Research and Development*, 6(1), 3.

Molléri, J. S., Petersen, K. & Mendes, E. (2020). An empirically evaluated checklist for surveys in software engineering. *Information and Software Technology*, 119, 106240.

Nahar, N., Zhou, S. & Lewis,G.and Kästner, C. (2022). Collaboration challenges in building ml-enabled systems: Communication, documentation, engineering, and process. *Proceedings of the 44th international conference on software engineering*, pp. 413–425.

Nahar, N., Zhang, H., Lewis, G., Zhou, S. & Kästner, C. (2023). A Meta-Summary of Challenges in Building Products with ML Components–Collecting Experiences from 4758+ Practitioners., 15 pages. *arXiv preprint arXiv:2304.00078*.

Naser, M. & Alavi, A. (2020). Insights into performance fitness and error metrics for machine learning. *arXiv preprint arXiv:2006.00887*.

OBrien, D., Biswas, S., Imtiaz, S., Abdalkareem, R., Shihab, E. & Rajan, H. (2022a). 23 shades of self-admitted technical debt: an empirical study on machine learning software. *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 734–746.

OBrien, D., Biswas, S., Imtiaz, S., Abdalkareem, R., Shihab, E. & Rajan, H. (2022b). 23 shades of self-admitted technical debt: an empirical study on machine learning software. *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 734–746.

OpenAI, O. A. [Accessed: 2024-08-10]. (2024). OpenAI. Retrieved from: https://techcommunity.microsoft.com/t5/fasttrack-for-azure/optimizing-azure-openai-a-guide-to-limits-quotas-and-best/ba-p/4076268.

Oueslati, K., Lamothe, M. & Khomh, F. (2025). RefAgent: A Multi-agent LLM-based Framework for Automatic Software Refactoring. *arXiv preprint arXiv:2511.03153*.

Overflow, S. [Accessed: 2025-01-07]. (2024). Best Practice. Retrieved from: https://stackoverflow.com/questions/51940106/what-is-the-best-practice-to-develop-cd-ci-when-you-use-ml-studio-apis.

Ping, D. (2022). *The Machine Learning Solutions Architect Handbook: Create machine learning platforms to run solutions in an enterprise setting*. Packt Publishing Ltd.

Policy, T. (2024). Termination Policy. Retrieved from: https://learn.microsoft.com/en-us/azure/machine-learning/how-to-tune-hyperparameters?view=azureml-api-2#early-termination.

Polyzotis, N., Zinkevich, M., Roy, S., Breck, E. & Whang, S. (2019). Data validation for machine learning. *Proceedings of machine learning and systems*, 1, 334–347.

Practices, G. B. (2024a). Alert. Retrieved from: https://cloud.google.com/architecture/ml-on-gcp-best-practices#fine-tune-alert-thresholds.

Practices, M. B. (2024b). Google. Retrieved from: https://cloud.google.com/architecture/ml-on-gcp-best-practices#use-training-checkpoints-to-save-the-current-state-of-your-experiment.

Quotas & limits. [Accessed: 2025-02-20]. (2024). Azure OpenAI. Retrieved from: https://learn.microsoft.com/en-us/azure/ai-services/openai/quotas-limits#quotas-and-limits-reference.

Rahman, M., Chy, M. S. H. & Saha, S. (2023). A systematic review on software design patterns in today's perspective. *2023 IEEE 11th International Conference on Serious Games and Applications for Health (SeGAH)*, pp. 1–8.

Recupito, G., Pecorelli, F., Catolino, G., Lenarduzzi, V., Taibi, D., Di Nucci, D. & 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.

Rojas, E., Kahira, A., Meneses, E., Gomez, L. & Badia, R. (2020). A study of checkpointing in large scale training of deep neural networks. *arXiv preprint arXiv:2012.00825*.

Sagemaker. [Accessed: 2025-08-11]. (2025). Train Machine Learning Models with Amazon SageMaker. Retrieved from: https://aws.amazon.com/sagemaker/ai/train/.

ScriptRunConfig, A. [Accessed: 2024-09-23]. (2024). Python SDK. Retrieved from: https://learn.microsoft.com/en-us/python/api/azureml-core/azureml.core.scriptrunconfig?view=azure-ml-py.

Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M., Crespo, J.-F. & Dennison, D. (2015). Hidden technical debt in machine learning systems. *Advances in neural information processing systems*, 28.

Serban, A., Van der Blom, K., Hoos, H. & Visser, J. (2020). Adoption and effects of software engineering best practices in machine learning. *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pp. 1–12.

Sharma, S. [Accessed: 2025-08-18]. (2024). The 100+ Problems that Affect Software Development Today. Retrieved from: https://medium.com/@ss-tech/the-100-problems-that-affect-software-development-today-e4fea7cb4f1a.

She, X., Liu, Y., Zhao, Y., He, Y., Li, L., Tantithamthavorn, C., Qin, Z. & Wang, H. (2023). Pitfalls in language models for code intelligence: A taxonomy and survey. *arXiv preprint arXiv:2310.17903*.

Shivashankar, K. & Martini, A. (2025). MLScent: A tool for Anti-pattern detection in ML projects. *IEEE/ACM 4th International Conference on AI Engineering–Software Engineering for AI (CAIN)*, pp. 150–160.

Sohail, P. [Accessed: 2025-09-22]. (2025). Riding the Waves of Data Drift: Best Practices for ML Success. Retrieved from: https://prvzsohail.medium.com/riding-the-waves-of-data-drift-best-practices-for-ml-success-85ed9f0b0b39.

Stack Overflow Community. [Accessed: 2025-08-18]. (2022). Azure Machine Learning REST API request limits. Retrieved from: https://stackoverflow.com/questions/72598952/azure-machine-learning-rest-api-request-limits.

Stahl, A. [Accessed: 2025-08-18]. (3). Developing AI/ML Projects For Business — Best Practices. Retrieved from: https://medium.com/@stahl950/developing-ai-ml-projects-for-business-best-practices-2881927462ff.

Targets, C. (2024). Training compute targets. Retrieved from: https://learn.microsoft.com/en-us/azure/machine-learning/concept-compute-target?view=azureml-api-2#training-compute-targets.

Testing, F. [Accessed: 2025-08-18]. (2023). Automated Testing in Azure DevOps: Best Practices and Tools. Retrieved from: https://www.frugaltesting.com/blog/automated-testing-in-azure-devops-best-practices-and-tools.

Tighilt, R., Abdellatif, M., Trabelsi, I., Madern, L., Moha, N. & Guéhéneuc, Y.-G. (2023). On the maintenance support for microservice-based systems through the specification and the detection of microservice antipatterns. *Journal of Systems and Software*, 204, 111755.

Tranquillin, M., Lakshmanan, V. & Tekiner, F. (2023). *Architecting data and machine learning platforms: enable analytics and AI-driven innovation in the cloud*. " O'Reilly Media, Inc.".

Van Oort, B., Cruz, L., Loni, B. & Van Deursen, A. (2022). " Project smells" experiences in analysing the software quality of ML projects with mllint. *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, pp. 211–220.

Victoria, A. & Maragatham, G. (2021). Automatic tuning of hyperparameters using Bayesian optimization. *Evolving Systems*, 12(1), 217–223.

Wan, C., Liu, S., Hoffmann, H., Maire, M. & Lu, S. (2021). Are machine learning cloud APIs used correctly? *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp. 125–137.

Wan, C., Liu, S., Xie, S., Liu, Y., Hoffmann, H., Maire, M. & Lu, S. (2022). Automated testing of software that uses machine learning apis. *Proceedings of the 44th International Conference on Software Engineering*, pp. 212–224.

Wang, C., Yang, Z., Li, Z., Damian, D. & Lo, D. (2024). Quality assurance for artificial intelligence: A study of industrial concerns, challenges and best practices. *arXiv preprint arXiv:2402.16391*.

Wang, J., Xiao, G., Zhang, S., Lei, H., Liu, Y. & Sui, Y. (2023). Compatibility issues in deep learning systems: Problems and opportunities. *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 476–488.

Washizaki, H., Uchida, H., Khomh, F. & Guéhéneuc, Y.-G. (2019). Studying software engineering patterns for designing machine learning systems. *2019 10th International Workshop on Empirical Software Engineering in Practice (IWESEP)*, pp. 49–495.

Washizaki, H., Khomh, F., Guéhéneuc, Y.-G., Takeuchi, H., Natori, N., Doi, T. & Okuda, S. (2022). Software-engineering design patterns for machine learning applications. *Computer*, 55(3), 30–39.

Washizaki, H., Takeuchi, H., Khomh, F., Natori, N., Doi, T. & Okuda, S. (2020). Practitioners' insights on machine-learning software engineering design patterns: a preliminary study. *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 797–799.

Weerts, H., Mueller, A. & Vanschoren, J. (2020). Importance of tuning hyperparameters of machine learning algorithms. *arXiv preprint arXiv:2007.07588*.

Wei, M., Harzevili, N., Huang, Y., Yang, J., Wang, J. & Wang, S. (2024). Demystifying and Detecting Misuses of Deep Learning APIs. *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pp. 1–12.

Wohlin, C. (2014). Guidelines for snowballing in systematic literature studies and a replication in software engineering. *Proceedings of the 18th international conference on evaluation and assessment in software engineering*, pp. 1–10.

Wolf, C. T. & Paine, D. (2020). Sensemaking practices in the everyday work of AI/ML software engineering. *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, pp. 86–92.

Xu, X., Liu, H., Tao, G., Xuan, Z. & Zhang, X. (2022). Checkpointing and deterministic training for deep learning. *Proceedings of the 1st International Conference on AI Engineering: Software Engineering for AI*, pp. 65–76.

Yang, C., Brower-Sinning, R. A., Lewis, G. & Kästner, C. (2022a). Data leakage in notebooks: Static detection and better processes. *Proceedings of the 37th IEEE/ACM international conference on automated software engineering*, pp. 1–12.

Yang, J., Ren, J. & Wu, W. (2022b). Api misuse detection method based on transformer. *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*, pp. 958–969.

Yang, S., Khuller, S., Choudhary, S., Mitra, S. & Mahadik, K. (2021). Scheduling ML training on unreliable spot instances. *Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing Companion*, pp. 1–8.

Yao, Y., Xiao, Z., Wang, B., Viswanath, B., Zheng, H. & Zhao, B. Y. (2017). Complexity vs. performance: empirical analysis of machine learning as a service. *Proceedings of the 2017 Internet Measurement Conference*, pp. 384–397.

Ying, X. (2019). An overview of overfitting and its solutions. *Journal of physics: Conference series*, 1168, 022022.

Z. Li, Y. Z. (2005). PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. *ACM SIGSOFT Software Engineering Notes*, 30(5), 306–315.

Zdrok, O. [Accessed: 2025-09-22]. (2024). Fairness Metrics in AI: Your Step-by-Step Guide to Equitable Systems. Retrieved from: https://shelf.io/blog/fairness-metrics-in-ai/?utm_source=chatgpt.com.

Zhang, C., Yu, M., Wang, W. & Yan, F. (2019). MArk: Exploiting cloud services for Cost-Effective, SLO-Aware machine learning inference serving. *USENIX Annual Technical Conference (USENIX ATC 19)*, pp. 1049–1062.

Zhang, H., Cruz, L. & Van Deursen, A. (2022). Code smells for machine learning applications. *Proceedings of the 1st international conference on AI engineering: software engineering for AI*, pp. 217–228.

Zhang, H., Alhanahnah, M., Ahmed, F. A., Fatih, D., Leitner, P. & Ali-Eldin, A. (2024). Machine Learning Systems are Bloated and Vulnerable. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 8(1), 1–30.

Zhang, L., Howard, S., Montpool, T., Moore, J., Mahajan, K. & Miranskyy, A. (2023). Automated data validation: An industrial experience report. *Journal of Systems and Software*, 197, 111573.

Zhang, Y., Shen, G., Zhang, L., Zheng, M. & Zheng, K. (2025). Refactoring for Java-Structured Concurrency. *Applied Sciences*, 15(5), 2407.