On the release-readiness of ML-based products

by

Harsh Patel

A thesis submitted to the

School of Computing

in conformity with the requirements for

the degree of Master of Science

Queen's University

Kingston, Ontario, Canada

April 2024

Copyright © Harsh Patel, 2024

Abstract

In the rapidly evolving fields of machine learning (ML) and artificial intelligence (AI), assessing the release-readiness of software products presents significant challenges. These include the unpredictability of AI results due to non-determinism, the complexity of comprehensive testing without clear model guidelines, and the need to comply with dynamic global regulations. To mitigate potential reputational risks, it is crucial for organizations to conduct detailed release-readiness evaluations. Traditional ML systems require checks like unbiased training data and model quality verification before deployment. Generative AI systems need additional checks to address challenges such as misinformation and defense against adversarial attacks like prompt injection.

This thesis explores release-readiness across traditional and generative AI systems. Traditional approaches often follow established checklists and assume newer models are superior, disregarding the utility of older models post-deployment. An empirical

study across eight long-lived Apache projects with 84,343 commits illustrates the benefits of model recycling, which preserves older models' utility and enhances software release-readiness.

We evaluated three Just-In-Time (JIT) defect prediction models—Random Forest, Logistic Regression, and Neural Network. Our findings show that model recycling significantly outperforms traditional retraining in metrics like recall, g-mean, AUC, and F1 score by up to 30%, 20%, 11%, and 10%, respectively. Model Stacking, the top recycling strategy, improves outcomes in over 50% of the projects but increases inference time.

For generative AI systems, the thesis identifies the absence of a comprehensive checklist tailored to the unique complexities of release-readiness. We synthesized a detailed checklist from a systematic review of 65 grey literature sources and insights from 44 organizations, guiding practitioners in evaluating key release-readiness aspects such as performance, monitoring, and deployment strategies.

This research bridges the gap between theory and practice in release-readiness, advancing the discourse on ML software systems and providing actionable insights for managing ML model lifecycles. It contributes to the development of more reliable and sustainable ML-based software systems, facilitating their successful integration into the real-world.

Acknowledgments

I extend my profound appreciation to Professor Bram Adams and Professor Ahmed E. Hassan for their unwavering support and insightful contributions to my research. Professor Bram, your exceptional ideas and constructive feedback have been a guiding light on this journey. Your mentorship has unlocked the mysteries of research for me. Professor Ahmed, your encouragement to tackle formidable challenges in our field has inspired me to reach new heights. This journey and what lies ahead would not have been possible without your guidance.

I would like to express my sincere gratitude to Dr. Ting Hu and Dr. Mariam Guizani for their willingness to join the jury.

Special thanks to Yihao Chen, Divya Kamath, and Rahul Bajaj, who have been not just mentors but dear friends. Your guidance and companionship have enriched my journey immensely.

To my lab mates, Akshat Malik, Shenyu Zheng, and Jaskirat Singh, thank you for

your collaborative spirit and assistance in my research efforts.

I am profoundly grateful to the Natural Sciences and Engineering Research Council of Canada (NSERC) and Mitacs for their generous funding of my research. Their support has been crucial in advancing the work presented in this thesis.

My heartfelt gratitude extends to Dominique and the team at BNC for their invaluable feedback and insights, which have significantly enriched my research.

My heartfelt gratitude goes to Mit, Maharshi, Nikunj, and Mihir for being pillars of support and true friends. With you, I have found an unwavering source of support and understanding.

I am grateful to Vishwa for her invaluable support. Our shared experiences and bond this year have been incredibly special.

To my housemates and dear friends, Jovian and Jay, thank you for being there when I needed you the most and for making my life happier. Your presence and support have added joy to my days and made this journey all the more memorable.

Lastly, I dedicate this thesis to my family, whose boundless support and love have been my constant source of strength. My parents, in particular, have been the bedrock of my journey through this Master's program. My gratitude towards them knows no bounds, and my love and appreciation are immeasurable.

Co-authorship

FOR all the thesis chapters and related publications, my contributions are designing and refining the research plan; conducting background literature reviews and related work elicitation; collecting datasets; conducting corresponding experiments and data analysis; and writing and polishing the manuscripts.

The work presented in this thesis are listed below:

- Harsh Patel, Bram Adams, and Ahmed E. Hassan. 2023. Post Deployment Recycling of Machine Learning Models; Don't Throw Away Your Old Models!. Journal of Empirical Software Engineering (EMSE). Submitted major revision.

- Harsh Patel, Dominique Boucher, Emad Fallahzadeh, Ahmed E. Hassan, and Bram Adams. 2024. A State-of-the-practice Release-readiness Checklist for Large Language Model-based Software Products. IEEE Software. To be submitted.

Table of Contents

Abstract i

Acknowledgments iii

Co-authorship v

List of Tables viii

List of Figures x

1 Introduction 1
- 1.1 Thesis Statement . . . 4
- 1.2 Thesis Overview . . . 5
- 1.3 Thesis Contributions . . . 6

2 Background and Related Work 9
- 2.1 Release-readiness of ML-based Software Systems . . . 9
- 2.2 Post Deployment Recycling of Machine Learning Models . . . 10

3 Post Deployment Recycling of Machine Learning Models 14
- 3.1 RQ1: Are old models worth saving? . . . 17
- 3.2 Model Recycling Algorithms . . . 42
- 3.3 Empirical Evaluation of Recycling Strategies . . . 59
- 3.4 Threats to Validity . . . 89
- 3.5 Discussion . . . 95
- 3.6 Chapter Summary . . . 106

4 A State-of-the-practice Release-readiness Checklist for Generative AI-based Software Products 107
- 4.1 Methodology . . . 109
- 4.2 Results . . . 112
- 4.3 Threats to Validity . . . 135

4.4 Chapter Summary 135

5 Conclusions and Future Work **137**
5.1 Thesis Contributions 139
5.2 Future Work 140

Bibliography **142**

List of Tables

3.1 Commit Metrics 19
3.2 ApacheJIT: Project Defect Distribution 19
3.3 Data Splits resulting from Sliding Window Sampling 23
3.4 Number and Proportion of Best Models per Project 37
3.5 Confidence interval results for FSR per reused model per project 39
3.6 List of variants evaluated for each recycling strategy 61
3.7 Model Selection (RF): # of projects (out of 8) experiencing significant change per evaluation metric 66
3.8 Model Selection (LR): # of projects (out of 8) experiencing significant change per evaluation metric 67
3.9 Model Selection (NN): # of projects (out of 8) experiencing significant change per evaluation metric 68
3.10 Model Reuse (RF): # of projects (out of 8) experiencing significant change per evaluation metric 69
3.11 Model Reuse (LR): # of projects (out of 8) experiencing significant change per evaluation metric 70
3.12 Model Reuse (NN): # of projects (out of 8) experiencing significant change per evaluation metric 70
3.13 Model Stacking (RF): # of projects (out of 8) experiencing significant change per evaluation metric 74
3.14 Model Stacking (LR): # of projects (out of 8) experiencing significant change per evaluation metric 75
3.15 Model Stacking (NN): # of projects (out of 8) experiencing significant change per evaluation metric 76
3.16 Model Voting (RF): # of projects (out of 8) experiencing significant change per evaluation metric 77
3.17 Model Voting (LR): # of projects (out of 8) experiencing significant change per evaluation metric 78
3.18 Model Voting (NN): # of projects (out of 8) experiencing significant change per evaluation metric 78
3.19 Clustering (RF): # of projects (out of 8) experiencing significant change per evaluation metric 79

3.20 Clustering (LR): # of projects (out of 8) experiencing significant change per evaluation metric . . . 79
3.21 Clustering (NN): # of projects (out of 8) experiencing significant change per evaluation metric . . . 80
3.22 # of projects (out of 8) experiencing significant change for the best recycling strategy variants (RF) . . . 84
3.23 # of projects (out of 8) experiencing significant change for the best recycling strategy variants (LR) . . . 84
3.24 # of projects (out of 8) experiencing significant change for the best recycling strategy variants (NN) . . . 85
3.25 Median latency ratio of recycling to baseline and statistical test results for inference latency of the best variant of each recycling strategy ($\alpha = 0.05$) (RF) . . . 90
3.26 Median latency ratio of recycling to baseline and statistical test results for inference latency of the best variant of each recycling strategy ($\alpha = 0.05$) (LR) . . . 91
3.27 Median latency ratio of recycling to baseline and statistical test results for inference latency of the best variant of each recycling strategy ($\alpha = 0.05$) (NN) . . . 91
3.28 Distinctive Features: Prior Work vs. Our Approaches . . . 99

List of Figures

3.1 Sliding Window Sampling 22
3.2 Feature Selection and Engineering 24
3.3 Evaluating FSR of Models over Time Windows 29
3.4 Identifying Statistical Significance of Feature Survival Rate Differences with the Help of Confidence Intervals (With Group A: Majority Group used for calculating Confidence Interval, Group B: Minority Group compared with Group A) 30
3.5 Camel: Percentage of Data Samples with Best Performance (i.e., $TCDF*100$) per Model Version over Time Windows 33
3.6 Best Models in terms of $FWDP$ per Project. Starred models also won as RFS model for the window they were trained on. 36
3.7 FSR per Winning Model Version per Project 38
3.8 Recycling Strategies 43
3.9 Filtering Model History using Statistical Tests 49
3.10 Inference Latency per Recycling Strategy (RF) 88
3.11 Inference Latency per Recycling Strategy (LR) 89
3.12 Inference Latency per Recycling Strategy (NN) 90

4.1 Frequency of challenges in our grey literature results (The numbers in the flowchart indicate the number of challenges associated with each stage of the LLM lifecycle, as identified through our survey) 113

CHAPTER 1

Introduction

In the dynamic world of machine learning (ML) and artificial intelligence (AI), launching new software products is filled with uncertainties. This unpredictability mainly comes from how ML models perform in real-life situations, which can be hard to predict (Paleyes et al., 2022). Examples of when AI doesn't meet expectations include Microsoft's chatbot releasing inappropriate tweets[1], Samsung banning the use of generative AI by staff following a ChatGPT data leak[2], and Amazon's AI recruitment tool showing bias towards male candidates[3]. Additionally, navigating varying international regulations—like the Artificial Intelligence and Data Act (AIDA)

[1]https://www.theverge.com/2016/3/24/11297050/tay-microsoft-chatbot-racist

[2]https://www.bloomberg.com/news/articles/2023-05-02/samsung-bans-chatgpt-and-other-generative-ai-use-by-staff-after-leak

[3]https://globalnews.ca/news/4532172/amazon-jobs-ai-bias/

in Canada[4], the Federal Trade Commission (FTC) in the USA[5], the AI Act in the EU[6], and China's AI development plan[7]—adds another layer of complexity.

As such, it is crucial for companies to thoroughly assess the readiness of their ML-based software before its public release, ensuring it aligns with best practices at every deployment stage. For instance, selecting the right training data is essential to prevent biases in the model, which could lead to unfair outcomes. Similarly, choosing the correct evaluation metrics is vital; in cases of imbalanced data, where one class dominates, traditional metrics like accuracy might not be ideal. Here, the G-Mean metric could offer a more balanced evaluation. Also, implementing quality checks, such as setting a minimum acceptable level for the Area Under the Curve (AUC) during the testing phase, ensures the model's reliability before it goes live. In case of generative AI systems, additional considerations come into play, reflecting their capacity to produce new, unseen content. These include evaluating the potential for generated outputs to inadvertently introduce biases or inaccuracies and ensuring the AI's decisions align with ethical guidelines.

The deployment process for ML-based software mirrors traditional software deployment in many ways, including the need for adherence to coding standards, testing, and security reviews. Yet, not all practices from traditional software engineering are applicable to ML. For example, the community often encourages swift adoption of the latest library versions in software engineering. However, this approach does not necessarily suit ML models, where newer versions, like GPT-4, might not be the best fit

[4]https://ised-isde.canada.ca/site/innovation-better-canada/en/artificial-intelligence-and-data-act-aida-companion-document

[5]https://www.ftc.gov/news-events/news/press-releases/2024/02/ftc-proposes-new-protections-combat-ai-impersonation-individuals

[6]https://digital-strategy.ec.europa.eu/en/policies/regulatory-framework-ai

[7]https://digichina.stanford.edu/work/full-translation-chinas-new-generation-artificial-intelligence-development-plan-2017/

due to higher costs and slower response times compared to earlier versions like GPT-3.5, which could be more suitable for generating product descriptions. This highlights a fundamental difference: software libraries are deterministic, whereas ML models are nondeterministic and subject to concept drift—a scenario where the statistical properties of the data being predicted change over time, necessitating periodic model retraining (Diethe et al., 2019).

The first part of our research takes a novel stance by advocating for the recycling of older models, which are traditionally discarded (inspired by traditional software engineering practices), recognizing their untapped value. This concept of model recycling is aimed at leveraging the knowledge accumulated by historical models at the granularity of individual model inference requests to enhance future predictions. This strategy directly contributes to improving the release-readiness of ML-based software, providing a more robust framework for deployment by making inference-time release-readiness decisions rather than a static decision. We empirically evaluate the effectiveness of this approach through experiments with Just-In-Time (JIT) defect prediction models (Kamei et al., 2012), showcasing how recycled models can significantly benefit software engineering practices.

Furthermore, the advent of generative AI-based software products, exemplified by the surge in applications following the release of ChatGPT, brings additional complexities to ensuring release-readiness. While some challenges mirror those faced by traditional ML systems—such as the unpredictability of outputs and the difficulty of thorough testing, these problems have led to the need for release checklists, i.e., formal lists of evaluation criteria that organizations should check either manually or (semi)automatically, to determine whether their product is ready for release to end users (Breck et al., 2017;

ISE, 2023; Zinkevich, 2017). However, as generative AI poses unique problems, including the risk of generating false information or being manipulated to reveal sensitive data, traditional checklists for release-readiness of ML systems no longer suffice. To address these challenges, the second part of our research develops a specialized release-readiness checklist for generative AI applications, informed by insights from various organizations. This checklist is designed to help navigate the nuanced landscape of generative AI, ensuring products are ready for public use while maintaining the integrity and reliability expected by end-users.

Through both parts of our research, the overarching goal is to enhance the release-readiness of ML-based software products, providing comprehensive strategies to navigate the uncertainties of deployment while maximizing the potential and safety of existing and new AI technologies.

1.1 Thesis Statement

This thesis argues that the conventional lifecycle management of ML models, characterized by periodic retraining and the subsequent discarding of older models, fails to recognize the latent value these models possess. Furthermore, in response to the unique challenges presented by generative AI-based software products, this thesis argues that existing ML-based software release-readiness checklists need to be updated with a focus on generative AI release-readiness covering the experiences of multiple organizations.

1.2 Thesis Overview

In this section, we provide an outline of this thesis:

1.2.1 Chapter 2: Background and Related Work

This chapter lays the groundwork for the thesis by conducting a comprehensive review of the literature on release readiness checklists and the phenomenon of concept drift in ML. It highlights the existing research gaps concerning the release-readiness of historical ML model reuse and generative AI-powered software products. The text delves into the progression of practices for managing the life cycle of ML models, pointing out the difficulties and limitations inherent in current methods, especially regarding model retraining, concept drift, and disposal of old ML models. Moreover, it points out the distinct challenges presented by generative AI-based software products, emphasizing the necessity for guidelines to guarantee the use of best-practices and their readiness for release.

1.2.2 Chapter 3: Post Deployment Recycling of Machine Learning Models

This chapter begins by assessing the significance of historical ML models in the domain of software analytics, offering an in-depth analysis of the inherent value these older models possess and advocating for their strategic reuse post-deployment. It introduces a structured approach for recycling models post-deployment, illustrating how past models can be effectively utilized to improve release readiness, better utilize the

cost associated with model retraining, and improve the predictive performance of models. Through empirical evaluation and case studies, this chapter substantiates the claim that old models are not merely obsolete artifacts but valuable assets that can contribute significantly to the development of robust, reliable ML-based software products.

1.2.3 Chapter 4: A State-of-the-practice Release-readiness Checklist for Generative AI-based Software Products

This chapter presents a pioneering release-readiness checklist tailored specifically for generative AI-based software products. This chapter delves into the complexities of deploying generative AI-based applications, addressing issues such as hallucination, prompt injection, and the imperative for current knowledge despite a knowledge cutoff. It introduces a comprehensive checklist that not only encapsulates the unique requirements of generative AI-based software but also incorporates insights from multiple organizations. Through a systematic methodology, the chapter articulates the development and validation of this checklist, representing a crucial step forward in securing the release readiness of generative AI-based software products.

1.3 Thesis Contributions

This thesis makes several pivotal contributions to the field of release readiness for ML-based software products. First, by, proposing post-deployment recycling of historical ML models and, second, by providing the first release-readiness checklist for generative AI-based software products. The key contributions are as follows:

- We conducted an extensive empirical evaluation to ascertain the utility of historical JIT defect prediction models. Our research demonstrates the value of these models for future inference tasks across 8 large and long-lived open-source projects, including ActiveMQ, Camel, Cassandra, Flink, Groovy, HBase, Hive, and Ignite.

- The thesis introduces a novel approach to the recycling of machine learning models post-deployment. We propose five distinct strategies for model recycling, aimed at leveraging the learnings from old models to enhance the efficiency and efficacy of current and future model deployments.

- An extensive empirical analysis was conducted to assess the performance of the proposed recycling strategies and their 29 variants. This was compared against the traditional retraining from scratch (RFS) approach for three different types of JIT models. The evaluation provides insights into the effectiveness of each strategy in terms of improving prediction accuracy and operational efficiency.

- We thoroughly investigated the trade-offs between inference latency and model performance across all proposed recycling strategies. This analysis aids in identifying the most suitable recycling approach based on ML practitioners' requirements and constraints.

- Addressing a critical gap in the domain of ML model deployment, we developed the first release-readiness checklist tailored for generative AI-based software products. This checklist is specifically designed to tackle unique challenges associated with generative AI, such as managing hallucinations, evaluating model performance, and ensuring data freshness. It represents a significant step towards

standardizing the release process for generative AI applications and ensuring their robustness and reliability in production environments.

- Unlike existing checklists on traditional ML that primarily focus on single-organization perspectives, our release-readiness checklist for generative AI-based software products incorporates insights from multiple organizations. This inclusive approach ensures that the checklist is broadly applicable and considers a wide range of operational, ethical, and technical challenges encountered in diverse organizational contexts. It lays the groundwork for a more collaborative and comprehensive approach to the release of generative AI-based software products.

CHAPTER 2

Background and Related Work

2.1 Release-readiness of ML-based Software Systems

There have been several studies on release-readiness checklists for (traditional) ML software systems. For example, Breck et al. (2017) present a test rubric based on Google's experience. It discusses the importance of reliability in ML systems, especially as they become integral to real-world applications, highlighting the complexity of testing due to ML's data and model dependencies. It offers 28 tests with a scoring system to assess ML system readiness for production. Zinkevich (2017) also provides a Google-based guide offering key strategies and best practices for machine learning system development. It covers everything from initial integration to advanced aspects like feature

engineering, aimed at guiding ML professionals in creating effective and efficient systems. Microsoft designed a production checklist providing a comprehensive guide for teams to assess if their ML model is ready for production (ISE, 2023). It includes key considerations such as model performance against a baseline, defining training and scoring metrics, benchmarking, data quality monitoring, integration impacts, and ethical concerns, ensuring a thorough readiness evaluation before deployment.

However, despite the breadth of these resources, certain gaps persist. Firstly, this thesis addresses critical gaps by challenging the prevailing assumption that newer models are inherently superior and should be adopted without question. Our research introduces post-deployment recycling of models, offering strategies to guarantee the release readiness of historical models for practitioners. Secondly, in contrast to the existing work, which primarily focuses on traditional ML approaches and often derives from the experiences of a single organization, our investigation into a release-readiness checklist for generative AI-based software products is novel. It extends the conversation by including perspectives from multiple organizations within the realm of generative AI, thereby broadening the scope and applicability of readiness assessment in this emerging field.

2.2 Post Deployment Recycling of Machine Learning Models

We discuss the differences between our results and the closest related work on black-box reuse of models in detail in Section **??**. This section discusses the related work concerning JIT defect prediction and drift in software defect prediction models.

2.2.1 JIT Defect Prediction

Defect prediction models have gained substantial attention in research (Zhao et al., 2023). In particular, Just-In-Time (JIT) defect prediction approaches have been proposed to help developers prioritize their limited software quality assurance (SQA) resources on the riskiest commits (Kamei et al., 2012; Kim et al., 2008), since JIT defect prediction aims to predict the defect likelihood of each software change immediately after it is submitted for code review. These models typically utilize various inputs, such as code change metrics (e.g., lines added or modified, complexity of changes, and developer activity patterns) to generate predictions. The output is often a risk score or a binary classification indicating whether a given commit will lead to a defect. These predictions can be expressed as concrete work assignments for a developer to fix a defect due to a change, reducing the time for finding the developer who introduced the defect because the predictions are made early on, this ensures that the details of the code change are still fresh in the developers' minds.

The four most commonly used families of code change properties used in JIT models (Kamei et al., 2012; Kim et al., 2008; Kononenko et al., 2015; Mockus and Weiss, 2000) typically measure (a) the magnitude of the change (Size); (b) the dispersion of the changes across modules (Diffusion); (c) the defect proneness of prior changes to the modified modules (History) and (d) the experience of the author (Author Experience).

2.2.2 Concept Drift in Software Analytics

Similar to other fields (Goncalves et al., 2024; Lu et al., 2018; Singhal et al., 2020), software analytics models such as JIT models are also affected by concept drift due to

evolving software project data and require retraining to maintain performance. Ekanayake et al. (2009) study the effect of concept drift in software projects by evaluating the prediction quality of file-level software defect prediction models over sequential releases of software projects over time. They find that the quality of the prediction follows periods of stability and drift in software projects, suggesting that concept drift is an important factor to consider when investigating defect prediction. Stability refers to a period when the defect prediction quality of a model is continuously above a certain threshold ($AUC > 0.8$). This stability period was observed to be as long as $\sim$ 15 months for one of the projects (Eclipse). However, the authors also found that the overall quality of the prediction drops significantly ($AUC \approx 0.5$) during periods of drift.

Our study is different in that we are evaluating JIT defect prediction models detecting bug-inducing commits in real-time, whereas Ekanayake et al. (2009) study the effect of concept drift over sequential releases of software projects using file-level features such as # revisions and # bugs fixed of critical type. While they study the effects of drift by looking at prediction performance over time using AUC, we evaluate drift by studying the feature survival rate of JIT models over time. Additionally, we evaluate each historical model over time based on its commit-level error performance compared to other historical models to identify the value of each model over time.

A study by McIntosh and Kamei (2018) suggests that fix-inducing changes are a moving target, and that JIT models lose a large proportion of their discriminatory power and calibration scores one year after being trained due to value shifts in the metrics used to train them. Their study evaluates JIT models over time and evaluates the importance scores of code change properties over time. We compare models by their commit-level error performance to quantify the model's value and correlate it with the

feature survival rate of the model to understand impact of drift. Olewicki et al. (2022) study when to retrain a model for detecting brown builds in continuous integration (CI) using a priori heuristics such as comparison to a prior week's performance or to the cumulative product of performance of recent models of past few weeks, which seem effective ways to reduce retraining costs. Both of these studies are related to our work, as they study the retraining problem in software analytics models. However, as both keep retraining new models, the older models are discarded, losing potentially valuable historical knowledge that the model has learned at a specific time. Our study focuses on how to keep reusing historical models, i.e., model recycling.

There have been multiple studies on online JIT defect prediction (Cabral and Minku, 2022; Tabassum et al., 2022; Tan et al., 2015). Although online learning can be an effective solution to retraining from scratch in practice because of the low cost of retraining, it however is prone to catastrophic forgetting (De Lange et al., 2021). For instance, Gao et al. (2023) found that fine-tuning CodeBERT on new datasets leads to substantial performance degradation on the initial dataset by 28.9% and 84.6% for code summarization and vulnerability detection, respectively. Olewicki et al. (2023) found that taking preventive measures for catastrophic forgetting of previously acquired knowledge, improved F1-score significantly from 4-26% for 3 out of 8 projects studied. This is particularly problematic when dealing with sequential data, as the model may prioritize recent data and forget valuable information from the past. Online JIT software defect prediction studies focus on improving the stability of predictions *over time*, while our study focuses on utilizing historical models to improve the quality of predictions *at inference time*. Therefore, our study could be complementary to these studies as it can be used along with online learning to improve the quality of predictions over time.

CHAPTER 3

Post Deployment Recycling of Machine Learning Models; Don't Throw Away Your Old Models!

In the dynamic landscape of ML-based software systems, the quest for optimal model performance—through rigorous selection, training, and fine-tuning—echoes the core principle of ensuring release-readiness (Schelter et al., 2015). Once the initial training of the machine learning models is completed, the focus shifts towards integrating the models into software products, deploying said models in production, then monitoring the performance of the machine learning models in production, identifying issues affecting model performance.

One of the most common ML issues encountered in production is data drift, also known as dataset shift (Quinonero-Candela et al., 2008), which occurs when an ML

model's inference data distribution changes compared to the training data distribution, leading to the model's performance degradation. Such changes can occur discretely, for example after some outside event that affects the inference data distribution, or continuously, with data gradually changing over time. When undetected, this phenomenon can have major adverse effects on model performance, as surveyed in multiple case studies by Paleyes et al. (2022). That said, drift is not necessarily a monotonous phenomenon, as it can manifest as sudden drift, gradual drift or even recurrent drift each of which requires specific monitoring and adaptation strategies to address effectively.

There are multiple techniques for adapting models to new data, including regular retraining from scratch and continual learning (Diethe et al., 2019), but both directions have important drawbacks, fundamentally rooted in the assumption that "newer models are better" prompting older models to be discarded. Yet, retraining models from scratch (RFS) on recent data is costly, both financially, due to the cost of hardware, electricity, or cloud compute time to train, develop, and maintain, and environmentally, due to the carbon footprint required to fuel modern tensor processing hardware (Strubell et al., 2019). With RFS, knowledge or insights gained from previous model versions are not transferred to the new model; therefore, there is no continuity between models. Continual learning, also referred to as lifelong learning (Chen and Liu, 2018), is a sequential learning process, requiring only a small portion of input data at once to evolve or fine-tune a model. However, it is prone to catastrophic forgetting: performance on previously learned data could significantly degrade over time as new data points are added (De Lange et al., 2021). Therefore, we focus on improving RFS model retraining in our study, critically evaluating the common belief in the superiority of

new models which often leads to the unnecessary obsolescence of older models.

We claim that, even with data drift, old models have value, and should be reused ("*recycled*") to improve future model performance. Based on this insight, we propose to use post-deployment recycling strategies that exploit past knowledge of model performance along with knowledge of data distribution at inference time to recommend models from the past for future inference when data distribution similarities are detected. The recycled models are used in a black-box manner, i.e., without fine-tuning or training, for inference on new data. Such reuse is still a challenging task as it now requires predicting the best model for a given inference data point, which can introduce additional noise. However, it can be beneficial since it can improve the overall prediction performance, and, because it treats the model as a black box, it can be adopted easily in existing ML pipelines.

We empirically evaluate our approaches in the context of Just-In-Time (JIT) defect prediction models (Kamei et al., 2012), a popular type of ML model in the domain of software engineering, using data of 8 large, long-lived open-source projects from the ApacheJIT dataset (Keshavarz and Nagappan, 2022), we address the following research questions:

- **RQ1**: Are old JIT prediction models worth saving?
- **RQ2**: What variant works best for each recycling strategy?
- **RQ3**: Which recycling strategy performs the best overall?

Our main contributions are:

- We empirically evaluate and identify the value that historical JIT models can provide to future inference on ActiveMQ, Camel, Cassandra, Flink, Groovy, HBase,

Hive and Ignite projects.

- We propose five different strategies for recycling software analytics models.
- We empirically evaluate the performance of these strategies and 29 variants compared to RFS for three different types of JIT models.
- We evaluate the trade-offs between each recycling strategy in terms of time-to-inference and metric performance.

3.1 RQ1: Are old models worth saving?

This section outlines the design and results of our initial empirical study trying to understand the value of old machine learning models in the context of JIT defect prediction.

3.1.1 Motivation

As ML models age, their prediction accuracy is expected to decrease due to data drift (Cabral and Minku, 2022; Ekanayake et al., 2009; McIntosh and Kamei, 2018; Olewicki et al., 2022). The general assumption is that it is necessary to regularly retrain the machine learning model with new data to make sure the model's predictions remain relevant to newer production data. In this race to avoid the negative effects of drift, ML teams rely on this assumption and simply discard old models without any second thought. This encourages us to find out whether old machine learning models (MLMs) are really of no value in practice, or whether there is any value in preserving old models.

3.1.2 Study Design

3.1.2.1 Studied Systems

3.1.2.1.1 Dataset

Several datasets are available for JIT defect prediction (Jiang et al., 2013; Kamei et al., 2012; Keshavarz and Nagappan, 2022; McIntosh and Kamei, 2018). Keshavarz and Nagappan (2022) built a large JIT bug prediction dataset called ApacheJIT comprising the commits of 14 popular Apache projects. ApacheJIT has 106,674 commits, 28,239 of which are labelled bug-inducing. Commits in ApacheJIT have the same features as Kamei et al. (2012), listed in Table 3.1.

In this study, we focused on utilizing commits from 8 specific projects within ApacheJIT as shown in table 3.2, excluding the remaining six projects for the following reasons:

- **Hadoop HDFS and MapReduce**: ApacheJIT contains commits from both the Hadoop HDFS and MapReduce projects. However, many commits from these projects are not segregated, making it challenging to accurately identify the project of origin for each commit and perform a study on these projects independently. It is plausible that this issue arose due to the projects' repositories being independent initially, but later consolidated into a shared repository named Hadoop.

- **Kafka, Spark, Zeppelin, and Zookeeper**: These projects have a notably low number of commits, of approximately 1,000 or even fewer. In our study, this limited number of commits translates to an insufficient number of time windows for each project for training and evaluating JIT models.

Table 3.1: Commit Metrics

Dimension	Metric	Description
Size	la	total number of lines added in commit
	ld	total number of lines deleted in commit
Diffusion	nf	number of files modified in commit
	nd	number of directories modified in commit
	ns	number of subsystems modified in commit
	entropy	distribution of change over files in commit
History	ndev	number of unique developers who previously changed the modified files of a commit
	age	average time since the previous change of the modified files of a commit
	nuc	number of unique changes to the modified files of a commit
Author Experience	aexp	number of prior commits by the commit author
	arexp	same but in the last # months
	asexp	number of prior commits to the subsystem of this commit by the commit author

Table 3.2: ApacheJIT: Project Defect Distribution

Project	# Total Commits	# Buggy Commits (%)	Time Period (Years)
ActiveMQ	6,126	1,404 (∼22%)	2005-2019
Camel	22,700	3,078 (∼13%)	2007-2019
Cassandra	8,159	3,117 (∼38%)	2009-2019
Flink	11,691	2,811 (∼24%)	2010-2019
Groovy	8,059	1,614 (∼20%)	2003-2019
Hbase	8,730	3,782 (∼43%)	2007-2019
Hive	6,842	4,223 (∼61%)	2008-2019
Ignite	12,036	2,439 (∼21%)	2014-2019

3.1.2.1.2 Machine Learning Models

Over the years, many machine learning algorithms have been used to predict buggy commits. However, Zeng et al. (2021) found that "*simple*" JIT defect prediction approaches outperform more complex approaches like CC2Vec and DeepJIT. Hence, we decided to use the JITLine model of Pornprasit and Tantithamthavorn (2021), which is one of the best-performing JIT defect prediction models and uses Random Forest as a classifier. To evaluate generalizability of our findings, we also evaluate our recycling strategies using a simpler model such as the Logistic Regression model inspired from Kamei et al. (2012) and a more complex Deep Neural Network model from Hoang et al. (2019) called DeepJIT. The details about the features used in each model are discussed later in Section 3.1.2.5.

Furthermore, since our study focuses on evaluating historical models, it is worth noting that the field of software engineering provides easy access to such models. This accessibility is facilitated by the availability of open-source software projects, which allow for the mining of chronological changes made to the software.

3.1.2.2 Data Preparation

This study aims to evaluate old JIT machine learning models' performance over time. Since each ApacheJIT project has a finite number of data samples, we use a sliding window technique to create more data splits. Figure 3.1 illustrates the sliding window sampling technique.

We first sort the commits by author date, then using a sliding window of one thousand commits that shifts by increments of two hundred, we create more windows for a dataset. The data windows mimic the real-time data stream encountered by machine

learning models in production. For every window created, we establish both a training and validation set. The training set comprises 80% of the data, randomly sampled from a window, while the validation set holds the remaining 20% of the data scattered across the window.

JIT defect prediction faces verification latency (Cabral et al., 2019), which is particularly important in online learning. This means that in reality a new data point (commit) can only be used for future (online) training once its actual label is known, which could take a variable amount of time. I.e., retrospectively assuming those data points' labels to be known instantly after a prediction is not realistic. Hence, to make empirical evaluation more realistic, it is custom to choose a latency threshold to delay the use of new data points for retraining. Song et al. (2023) found that reducing this latency threshold (from 90 to 15 days) led to an increase of up to 45.86% in mislabeling defect-inducing examples. To address this threat to validity, we keep a gap of 200 samples (median of 57 days) between our training and testing sets, denoted as "Delayed Labelling (DL)" (also known as verification latency) in Figure 3.1. Subsequently, we use the two hundred samples after the DL gap as the test set, resulting in a test set of the same length as the validation and DL sets.

Based on the sliding window sampling technique, we create a total of 386 data splits across eight projects, as shown in Table 3.3. The minimum number of commits for a project is 6,126 (ActiveMQ), which is far greater than the number of commits required to create a window (i.e., 1,000) and results in a minimum of 26 data splits according to the following formula:

$$\#\,\text{Data Splits} = \lfloor \frac{\#\,\text{Commits} - \text{Window Size}}{\text{Window Shift}} \rfloor + 1$$

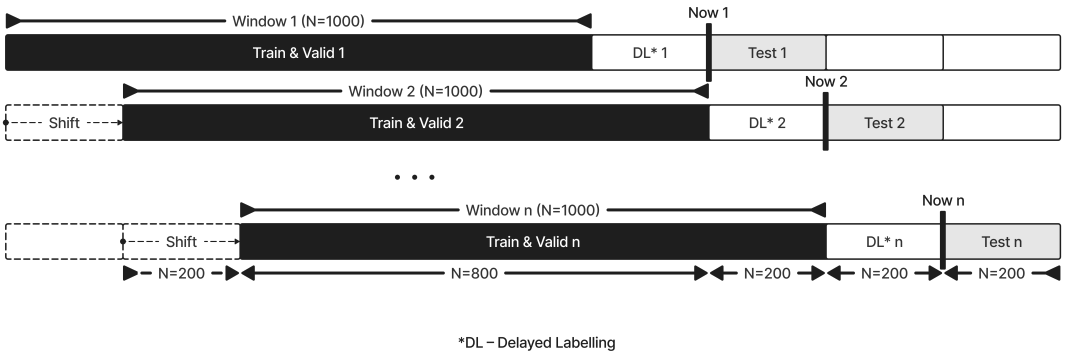


Figure 3.1: Sliding Window Sampling

3.1.2.3 Model Version History

Using the above sliding window sampling, each resulting time window serves as the training data for a new RFS model. Ekanayake et al. (2009) used a constant training period length of 2 months and evaluated defect predictions over each month while our training period is of a median of 9 months over different projects. We also maintain a history of 15 model versions consisting of 8-123 months of data depending on the project and with a median of 33 months for each window evaluated. This decision strikes a balance between excessive reliance on distant model history and retaining sufficient context.

On top of 15 windows for 15 historical models, the current window is required to train the RFS model as our baseline, the next window is skipped to address verification latency, and the window after that is used for evaluation. Therefore, the total # evaluated windows for each project is lower than the total # windows (data splits) created for the project as listed in Table 3.3. As the sliding window advances, the model history

Table 3.3: Data Splits resulting from Sliding Window Sampling

Project	**# Total Commits**	**# Data Splits**	**# Evaluated Windows**
ActiveMQ	6,126	26	9
Camel	22,700	109	92
Cassandra	8,159	36	19
Flink	11,691	54	37
Groovy	8,059	36	19
Hbase	8,730	39	22
Hive	6,842	30	13
Ignite	12,036	56	39

moves forward. When the model history size surpasses the predefined limit of 15 windows, we discard the oldest version, ensuring we continuously have updated models while maintaining a manageable history.

In section 3.3, in addition to the variants with limited model history, we also evaluate variants of recycling strategies without limits on the model history size for each window. This allows us to consider a broader historical perspective.

3.1.2.4 Handling Class Imbalance

Since software projects typically have very few bugs relative to the total number of commits, JIT datasets typically are imbalanced. Several sampling strategies are available to address imbalanced class problems, like oversampling and undersampling. For the

JITLine and Logistic Regression models, we chose the oversampling approach Synthetic Minority Over-sampling Technique (SMOTE) as used by Pornprasit and Tantithamthavorn (2021). The SMOTE method of over-sampling the minority class involves creating synthetic minority class examples, and it can achieve better classifier performance than only undersampling the majority class (Chawla et al., 2002). For DeepJIT, each window of data is resampled with replacement into smaller batches having equal distribution of both positive and negative labels to address the class imbalance. In our sliding window sampling shown in figure 3.1, we apply SMOTE or resampling to each training set only, while the validation and test sets remain unchanged.

3.1.2.5 Feature Selection and Engineering

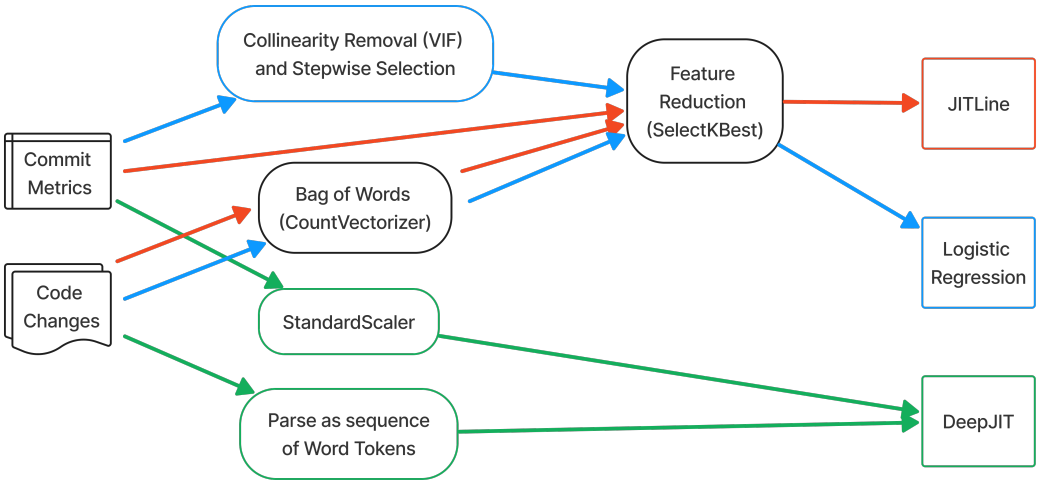


Figure 3.2: Feature Selection and Engineering

The pre-processing pipeline of features for each model is shown in Figure 3.2. We

use the same features as Pornprasit and Tantithamthavorn (2021) for all models. However, there are some differences in the way these features are processed for each model, as discussed below.

JITLine – JITLine uses both commit metrics listed in Table 3.1 and code changes (i.e., added and removed lines of code in a commit). The code changes are converted into count vectors using the Bag of Words (BoW) technique. BoW is a common technique for extracting features from text documents that involves counting the number of times a word appears in a document, then transforming the text into a vector of word counts. Since code changes can include thousands of code tokens, including commit metrics, this leads to a very large feature space, considering our window size of 1,000 data points, and that a general rule-of-thumb calls for at least 10-15 samples per feature to avoid overfitting [1].

To reduce the feature space, we use scikit-learn's $SelectKBest$ feature selection method (Pedregosa et al., 2011) to select the top 100 features (tokens) based on the chi-squared test. The chi-squared test is often employed for feature selection in text classification tasks because it helps identify the most relevant features by measuring the independence between each feature (term or word) and the class labels. It focuses on selecting features that have significant associations with the classes, allowing for the prioritization of terms that contribute the most discriminative information for classification.

Logistic Regression – We use the same features as JITLine, but to remove collinearity between commit metrics features, we use the Variance Inflation Factor (VIF) score to remove features having a score above 10 (James et al., 2013). Additionally, we use stepwise selection to remove features that do not contribute to the model's performance.

[1] https://postindustria.com/how-much-data-is-required-for-machine-learning

Finally, we use the same feature selection method as JITLine to select the top 100 features.

DeepJIT – The original DeepJIT model does not use commit metrics from Table 3.1 as features. Instead, it uses the commit messages and code changes. However, to keep consistency across all models, we use the same features as JITLine and Logistic Regression models, i.e., both commit metrics and code changes. To improve time to convergence, we use scikit-learn's $StandardScaler$ to scale the features to a mean of 0 and a standard deviation of 1 (Pedregosa et al., 2011). DeepJIT does not require feature reduction since it automatically understands the semantic features of each deleted or added line in each changed file. Then, it aggregates these features to generate a new representation (embedding) of the changed file, which is used to construct the features of the code changes (Hoang et al., 2019).

3.1.2.6 Evaluating Model Performance

We use standard performance metrics such as F1 score, AUC, Precision and Recall to evaluate each model's performance. Apart from these metrics, we also evaluate models using Specificity, Geometric Mean and Average Precision.

For binary classification, G-mean is the square root of the product of the minority and majority class recall values ($\sqrt{recall \times specificity}$). The minority recall is the ratio $tp/(tp+fn)$ where tp is the number of true positives and fn is the number of false negatives, quantifying the ability to avoid false negatives. Specificity is the ratio $tn/(tn+fp)$ where tn is the number of true negatives and fp is the number of false positives. It quantifies the ability to avoid false positives (Lemaître et al., 2017).

For readability purposes, we only discuss g-mean, AUC, F1 score, precision and

recall in our summary plots and tables in the thesis, while the replication package also includes the other performance metrics, i.e. Specificity and Average Precision (Patel, 2023). Since the g-mean is a geometric mean of performance measured separately on each class, it is a better metric for the performance of imbalanced classification tasks (Kubat et al., 1997).

3.1.2.7 Analysis of Old ML Models

To understand how trained ML models change their performance over time, we evaluate each ML model version on test sets of future time windows. Based on their error on each data sample $Y_{error} = abs(Y_{prob} - Y_{actual})$ we find the best-performing old model for each future commit. We then lift up this information to the level of time windows by determining the model versions performing best on the largest number of commits in that window. This metric can be defined as the Temporal Commit Dominance Factor ($TCDF$), which is calculated as follows:

$$TCDF(m_i, w_j) = \frac{\#\text{ Commits in } w_j \text{ for which } m_i \text{ achieves the lowest } Y_{error}}{\#\text{ Commits in } w_j}$$

Where m_i is the model version, w_j is the window and $i \leq j$.

$TCDF$ quantifies the model's dominance and effectiveness over a range of commits within a given time window. For each model, we subsequently calculate the percentage of future windows in which that version was the best in terms of $TCDF$ in order to quantify its (future) value. This is defined as the Future Window Dominance Percentage ($FWDP$) and is calculated as follows:

$$FWDP(m_i) = \frac{\text{\# Future Windows for which } m_i \text{ achieves the highest } TCDF}{\text{Total \# Future Windows}}$$

We do not use statistical tests to identify winning model versions, since even if a winning model only marginally wins on more commits, the RFS model still is not that appealing.

To understand the performance of older models in the presence of drift, we also evaluate the Feature Survival Rate (FSR) of models over time (Olewicki et al., 2022). As explained in Figure 3.3, we do this by first calculating, for each model of each time window, the top 20 features based on their feature importance. Then, for each pair of time windows (i, j; where, $j > i$), we find the percentage of the top 20 features from model i that survived in model j. FSR can be calculated as follows:

$$FSR(m_i, m_j) = \frac{\text{\# Selected Top Features from } m_i \text{ that survived in } m_j}{\text{\# Selected Top Features in } m_i (= 20)} * 100$$

Where m_i and m_j are the model versions and $j > i$.

To measure feature importance, we use the SHapley Additive exPlanation (SHAP) values proposed by Lundberg and Lee (2017), which is a Classifier Agnostic Feature Importance method. It is one of the more recent global feature importance methods that gives consistent results not impacted by feature interactions (Rajbahadur et al.,

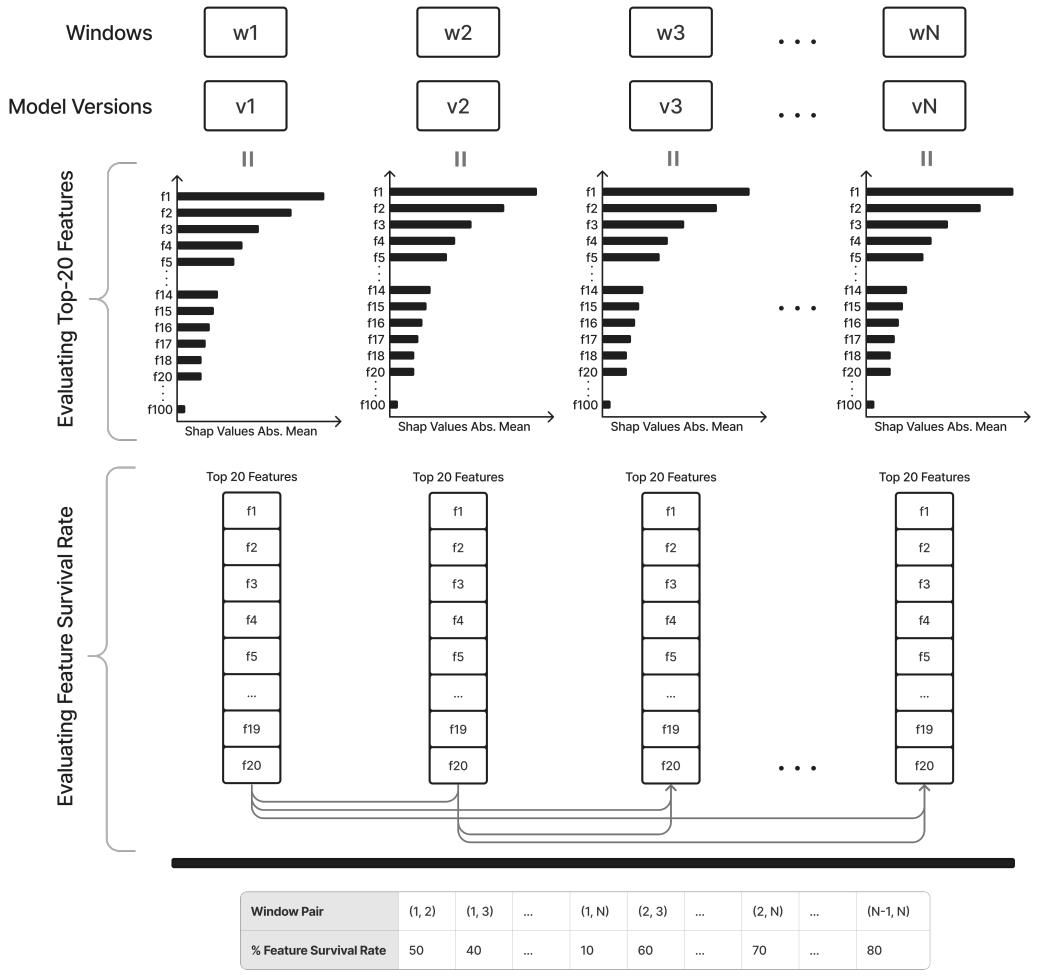


Figure 3.3: Evaluating FSR of Models over Time Windows

2021). We rank features based on the mean SHAP value first, then its standard deviation (from high to low) across all test samples in a time window, and select the top 20 features. The high mean is an indicator of the overall high importance of a feature across all test samples, while the high standard deviation is an indicator of the changing importance of a feature across test samples meaning that the feature may have a

positive (predicting commit as buggy) or negative impact on prediction depending on the test sample.

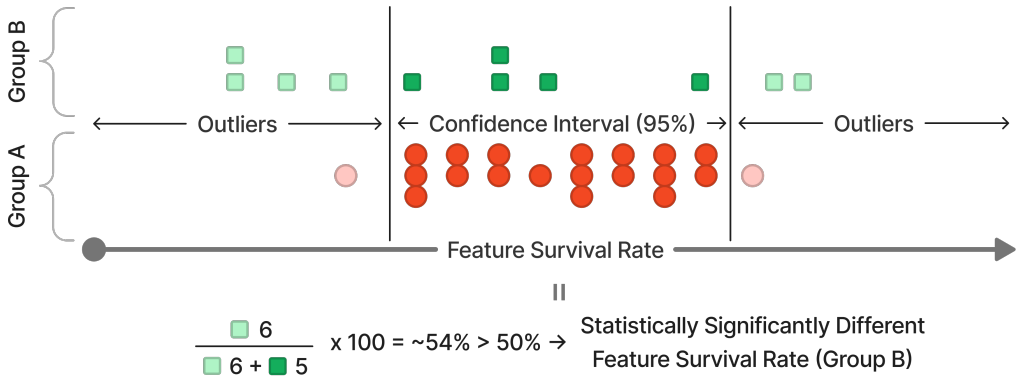


Figure 3.4: Identifying Statistical Significance of Feature Survival Rate Differences with the Help of Confidence Intervals (With Group A: Majority Group used for calculating Confidence Interval, Group B: Minority Group compared with Group A)

To understand whether drift plays any role in the observation of a model being the best for a given future time window or not, we want to identify whether there is a statistically significant difference between feature survival rates of a model in the windows where the model is winning vs. not winning. For this, we perform in each studied project a statistical analysis of all models that won in at least one window. We observed that one of these two groups (typically the "*winning*" group, sometimes "*not-winning*") sometimes can have fewer than six data samples, which makes it difficult to use statistical tests like Wilcoxon signed-rank test. Therefore, as shown in Figure 3.4, we calculate the confidence interval ($\alpha = 0.05$) of the feature survival rate for the majority group (i.e., either the "*winning*" or "*not-winning*" group) and identify the data samples of the minority group falling outside the confidence interval of the majority

group. Since our original goal is to understand the difference in statistical significance, we consider the difference in feature survival rate statistically significant if 50% or more minority data points fall outside the majority confidence interval.

Models from the last few windows have fewer chances to win due to the lack of future test windows. Hence, to ensure sufficient data points for calculating confidence intervals, we exclude model versions that are within ten windows of the end window.

Sometimes a model may win in an immediate next window but may not win far in the future. On the contrary, a model may not win immediately but may win far in the future. So, to understand the potential of model reuse, we investigate the period of potential reuse by identifying the last and first windows a model is winning in. This gives us an idea of how far in the future a model can be reused.

3.1.3 Results

Old models have value. Based on TCDF, we found that six projects (ActiveMQ, Camel, Cassandra, Groovy, HBase and Ignite) have an old model dominating in $\geq$ 52% of their time windows. The other project (Flink) has an old model dominating in $\geq$ 32% of their time windows. Only one project (Hive) has more uniformly distributed windows where it can be difficult to predict the winning model. Figure 3.5 evaluates, for a given time window i of project Camel (X-axis), all past model versions, to calculate their classification error (Y_{error}) on that window's test commits (Y-axis). Each cell in the figure shows the value of $TCDF$, i.e., the ratio of commits in which that cell row (model version) outperformed all other, model versions available for that window (vertical column) in terms of absolute error. For a given time window (vertical column), the higher the number of commits for which a model is the best, the darker the colour of

the cell.

The figure shows long, horizontal red lines for a minority of model versions, suggesting that these model versions continuously performed better for an extended time period. For example, model $v1$ has the longest and darkest horizontal red line expanding approximately until window 86 (covering $\sim$ 74% of time windows, comprising commits from $\sim$ 101 months). This effect is shifting to other model versions over time, e.g., model $v60$ and later model $v90$, following a similar trend of a continuous horizontal red line. In contrast, we note that the diagonal is not dark, indicating that RFS models do not automatically outperform older models, even though they have been trained using the most recent data for that inference time window. Figures for other projects show similar patterns and have been included in the replication package (Patel, 2023).

Figure 3.6 shows, for each project, the $FWDP$ for the model versions (with $FWDP > 0$). As per the figure, model $v1$ for project Camel was considered the best model version for $\sim$ 74% of the time windows, which confirms our finding above, with model $v90$ in second position winning in $\sim$ 8% of the windows. The figure shows that old models dominated at least $\geq$ 52% of the time windows for all projects except Flink and Hive. Although Flink did not have a single dominating old model, it had three dominating old models that constituted $\sim$ 78% of total windows. For Hive, it was found that models $v23$ and models from $v25$ to 28 were RFS models winning their targeted window while $v7$ was the only model able to win time windows up to 3 times consecutively, whereas the remaining versions only won 1 or 2 consecutive windows. This could be because of the high ratio of buggy commits ($\sim$ 60%) present in Hive compared to other projects ($<$ 50%), which may require model adaptations that old models cannot accommodate.

As to why early models seem to dominate a substantial percentage of time windows,

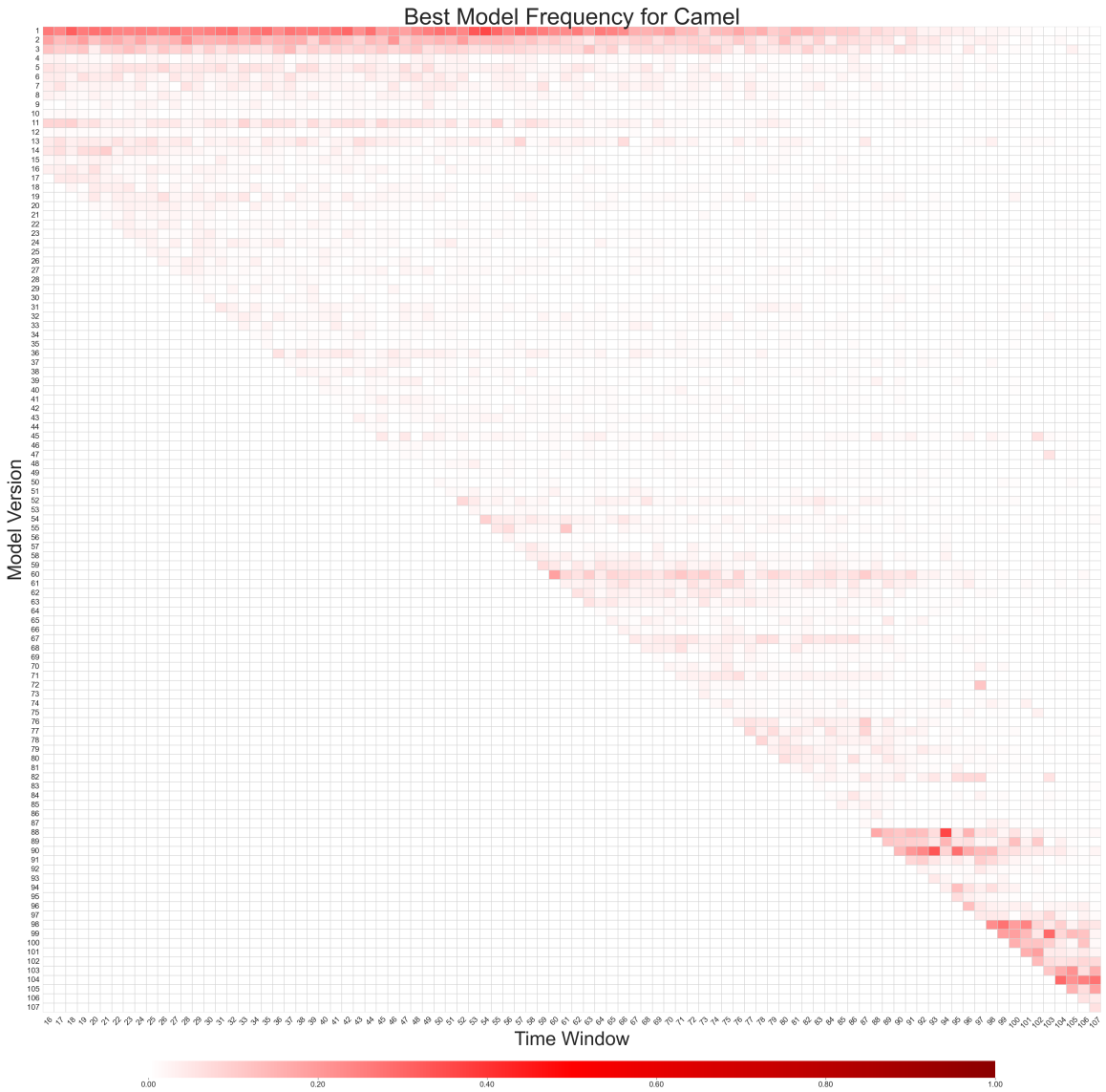


Figure 3.5: Camel: Percentage of Data Samples with Best Performance (i.e., $TCDF * 100$) per Model Version over Time Windows

we have a number of hypotheses. First, the nature of software development evolves

over time, especially during major release development periods marked by substantial feature and restructuring changes that affect larger segments of a codebase. Conversely, minor releases, following Lehman's fifth law and emphasizing the "conservation of familiarity", highlight that growth is constrained by the necessity to maintain familiarity (Herraiz et al., 2013). Many software projects follow an agile process with iterations of N weeks, where changes within each iteration are more cohesive than across iterations. Therefore, consecutive iterations often focus on more similar changes than non-consecutive ones, allowing older models to remain relevant for extended periods.

For example, in Figure 3.5, noticeable triangle patterns align with major version releases of Apache Camel[2], such as the periods from window 16 to 87, 88 to 97, 98 to 103, and 104 to the end. Winning models like $v1$, trained on commits from 2007-2008, coincide with the release of Camel $v1.0.0$. Similarly, models $v88$, $v98$, and $v104$, trained on commits from 2017, 2018-2019, and 2019, respectively, align with the releases[3] of Camel $v2.20.0$, $v2.24.0$, and $v3.0.0$. This suggests that the changes in subsequent releases might not be substantial enough (conservation of familiarity) to warrant retraining the model, explaining the sustained performance of older models.

Second, JIT (Just-in-Time) software defect prediction encounters concept drift, which involves changes in the ratio of examples for each predicted class (Cabral et al., 2019). Such changes reflect shifts in the software development process, including evolving priorities or refactoring efforts. For example, a development team might initially prioritize the graphical user interface (GUI) and later shift focus to implementing business logic, introducing a concept drift in the relationship between input features and labels.

[2]https://github.com/apache/camel
[3]https://camel.apache.org/releases/#camel

Additionally, alterations in software development practices, like refactoring, can influence the rate of defect-inducing changes, resulting in shifts in the class distribution.

As seen in Figure 3.5, the observable triangle patterns coincide with shifts in the label distribution. For example, windows 16-87 exhibit a median of approximately 157 buggy commits, windows 88-97 show a median of about 60 buggy commits, windows 98-103 display a median of around 25 buggy commits, and windows 104 to the end demonstrate a median of approximately 20 buggy commits. Building on the earlier reasoning, the older model leading the distribution change appears to outperform the more recent RFS model, which is trained on the latest data.

For most projects, the percentage of RFS models winning was meagre($< 14\%$), except for Hive ($\sim 38\%$), where using the RFS models can contribute the most to achieving the best performance. Table 3.4 shows the percentage of windows where the latest RFS model outperformed the older models, when evaluated on test commits from its respective test window. Hive's ratio of best models to total models of $\sim 30\%$ could be because of the high ratio of buggy commits ($\sim 60\%$) present in Hive compared to other projects ($< 50\%$), which may require model adaptations that old models cannot accommodate, as discussed above.

The median FSR values of 17 out of 26 winning model versions in winning windows consistently range from 30% to 70%. Figure 3.7 illustrates the FSR for model versions on their winning time windows compared to their non-winning time windows for each project. Please note that because project Hbase had only one old model version ($v1$) winning all future windows, we do not see any data points in Figure 3.7 to compare with the winning windows. Overall, looking at the quartiles indicated in the figure, we can see that the FSR of models in winning time windows is higher than their

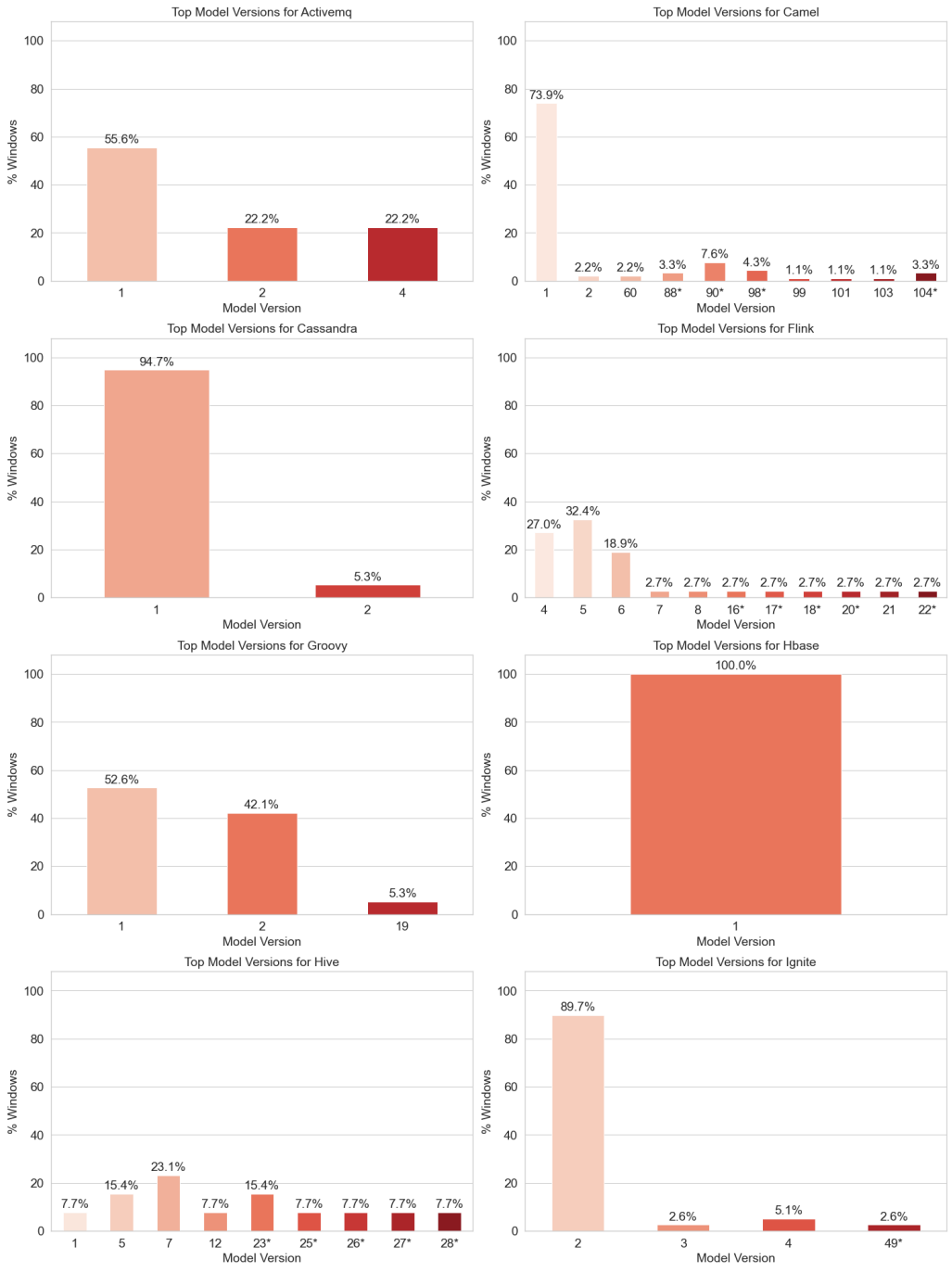


Figure 3.6: Best Models in terms of $FWDP$ per Project. Starred models also won as RFS model for the window they were trained on.

Table 3.4: Number and Proportion of Best Models per Project

Project	# Total Models	# Winning Models	% Winning Models	% RFS Models Winning[1]
ActiveMQ	26	3	11.54	0.00
Camel	109	10	9.17	4.35
Cassandra	36	2	5.56	5.00
Flink	54	11	20.37	13.51
Groovy	36	3	8.33	5.00
Hbase	39	1	2.56	4.35
Hive	30	9	30.00	38.46
Ignite	56	4	7.14	2.56

Outlier

[1] Ratio of # Winning RFS Models over # Evaluated Windows (see Section 3.1.2.3)

FSR in non-winning time windows. At first sight, this finding aligns with the common wisdom that model retraining becomes necessary when high drift occurs. Table 3.5 indeed confirms that the FSR for the windows in which a model is winning is higher than for windows in which the model is not winning. This is confirmed for 17 out of 26 winning model versions by the fact that for $> 50\%$ of minority (mostly from winning windows) data points the FSR falls outside the confidence interval (OCI) of the majority group (mostly from non-winning windows).

However, the Median Feature Survival Rate (MFSR) in winning windows is still relatively low with only 7 out of 26 model versions having an MFSR of at least 50%, indicating that older models still outperform the RFS models in the presence of non-trivial concept drift. The reason behind this unexpected observation could be that the non-surviving features of these older models do not contribute significantly to the prediction and the rules learned by the model from an older time window are more fundamental and can be applied to a large proportion of new commits, indicating repeating

patterns in the data.

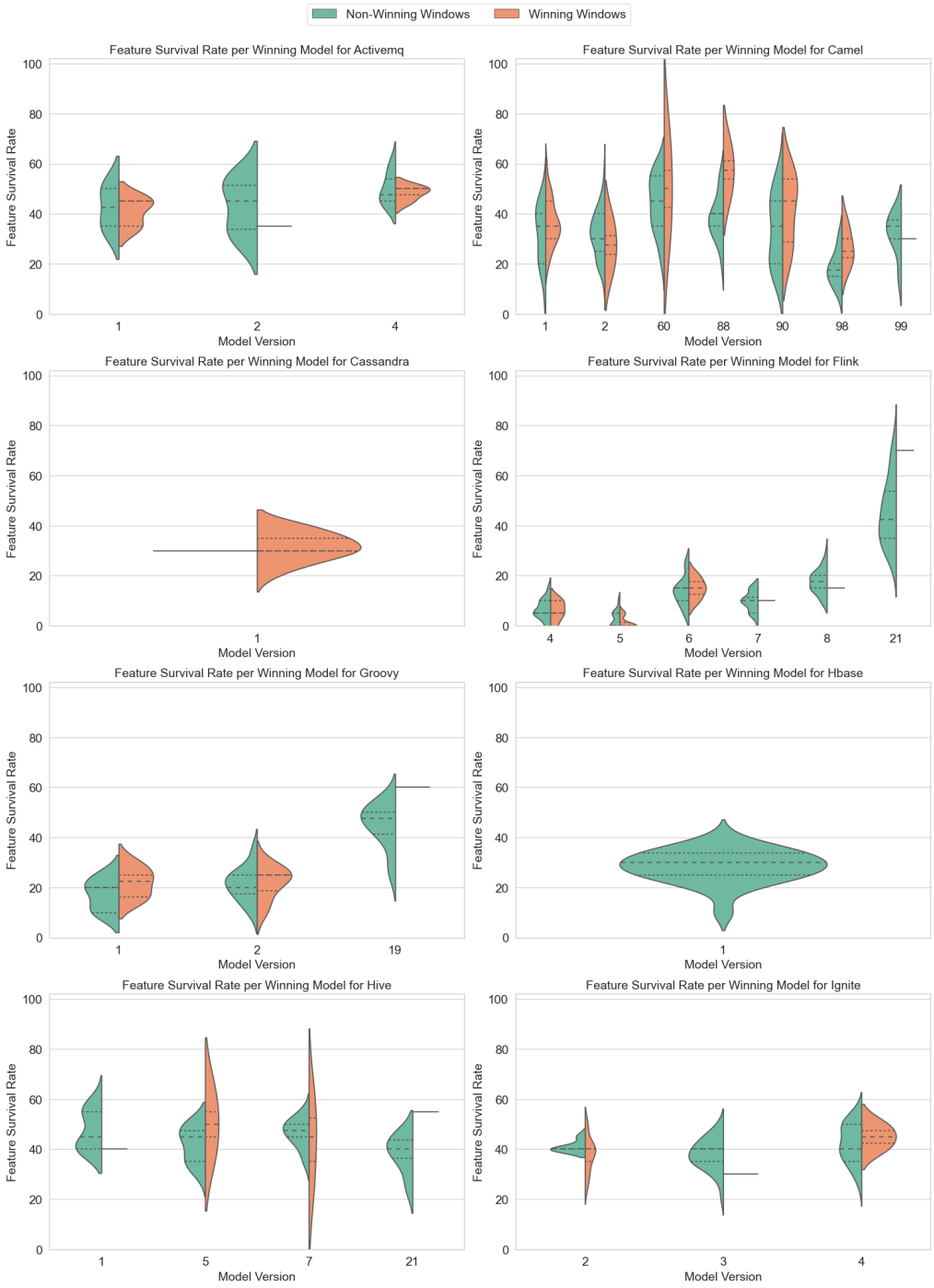


Figure 3.7: FSR per Winning Model Version per Project

Table 3.5 shows that the # of time windows of a model from its first winning window to its last winning window, ranges from 3 **to** 70 **for more than** 60% **of the winning model versions.** The longest winning (reuse) period was for model $v1$ for project camel, with the last winning window of that model being as far as 70 windows, despite the MFSR across the 70 winning windows being relatively low (35%). A similar pattern of long-range winning and low FSR can be seen for other projects too. Additionally, the # of winning windows for a model is more than 1 for ∼ 65% of the winning model versions. The longest reuse period is indicated in bold in Table 3.5, and ranges from 3-71 windows, and with MFSR ranging from 0-50%. Where a model has more than 1 winning window (65% of winning models), the median reuse period is ∼ 17 and the MFSR median is ∼ 35%. Despite the low MFSR of winning model versions, these models continue to perform well and win in future windows as far as 70 windows away from their first winning window, indicating their potential to be recycled. This intriguing outcome challenges the common belief that models must always be retrained from scratch and discarded.

Table 3.5: Confidence interval results for FSR per reused model per project

Project	Model Version	# Windows		OCI[1]		WW[3]		MFSR[4]
		W[2]	NW[2]	#	%	# Windows	Window	%
ActiveMQ	2	1	8	0	0.00	0	15	35.0

Continued on next page

Table 3.5 – continued from previous page

Project	Model Version	# Windows		OCI[1]		WW[3]		MFSR[4]
		W[2]	NW[2]	#	%	# Windows	Window	%
	4	3	6	0	0.00	**7**	16	50.0
Camel	1	68	24	21	87.50	**70**	15	35.0
	2	2	90	2	100.00	34	45	27.5
	60	2	45	2	100.00	14	72	50.0
	88	2	17	2	100.00	5	88	57.5
	90	6	11	3	50.00	6	90	45.0
	98	3	6	1	33.33	2	98	25.0
	99	1	7	0	0.00	0	102	30.0
Cassandra	1	18	1	0	0.00	**18**	15	30.0
Flink	4	10	27	6	60.00	**29**	20	5.0
	5	12	25	12	100.00	**29**	22	0.0
	6	7	30	4	57.14	28	18	15.0
	7	1	36	0	0.00	0	45	10.0
	8	1	36	1	100.00	0	32	15.0
	21	1	30	1	100.00	0	24	70.0
Groovy	1	10	9	4	44.44	**17**	15	22.5

Continued on next page

Table 3.5 – continued from previous page

Project	Model Version	# Windows		OCI[1]		WW[3]		MFSR[4]
		W[2]	NW[2]	#	%	# Windows	Window	%
	2	8	11	3	37.50	16	17	25.0
	19	1	14	1	100.00	0	19	60.0
Hive	1	1	12	1	100.00	0	16	40.0
	5	2	11	1	50.00	**3**	15	50.0
	7	3	10	2	66.67	**3**	17	45.0
	21	1	6	1	100.00	0	21	55.0
Ignite	2	34	5	1	20.00	**38**	15	40.0
	3	1	38	1	100.00	0	28	30.0
	4	3	36	1	33.33	20	29	45.0

[1]OCI – % of minority windows where FSR lies Outside the Majority 95% Confidence Interval

[2]W – # Windows where model version is winning, NW - # Windows where model version is non-winning

[3]WW – Winning Window, i.e., the window where the given model version was winning

[4]MFSR – Median FSR on Windows in which a model version is winning

Summary for Research Question 1

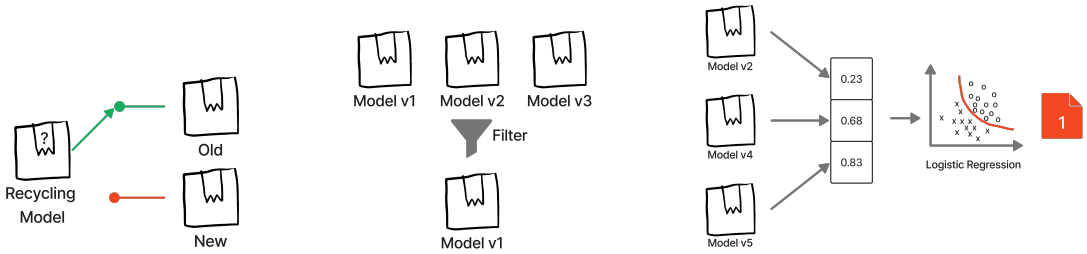
Old models have the best performance in ≥ 52% and ≥ 32% of future time windows for six and one projects respectively, up to a median of 3-70 windows in the future. This suggests that saving and potentially recycling old models to make predictions in future windows could make more sense than RFS in the context of JIT.

3.2 Model Recycling Algorithms

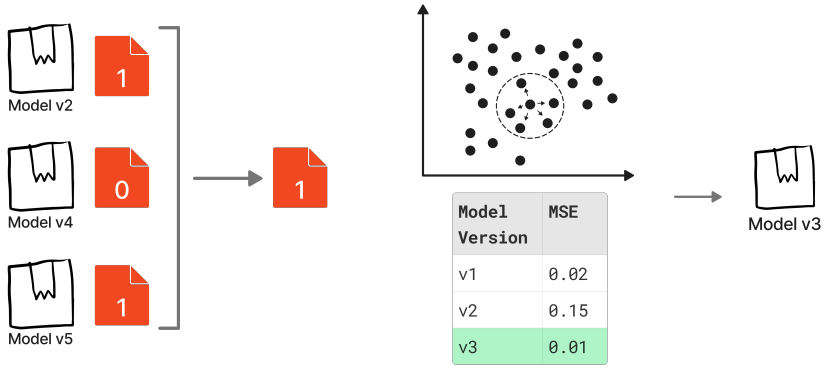
Given that RQ1 showed that old models have value, we need to find the best approach to recycle them. *Recycling* is the process of reusing old models to improve the prediction performance at a future inference time. Recycling not only emphasizes the preservation of valuable historical models, but also encourages ML practitioners to derive continued benefits from these models' accumulated knowledge.

However, the process of devising effective recycling strategies is not without its challenges. The crux of the challenge lies in striking a balance between the utility of old models and the potential risks associated with their use. On the one hand, outdated models might fail to capture the nuances of evolving data patterns, leading to diminished prediction accuracy. On the other hand, deploying older models too conservatively might overlook opportunities where their insights could still prove valuable. To address this challenge, we propose a range of distinct strategies for recycling old models. The common goal behind each strategy is to make informed estimations about which old models to use and when to use them.

This section describes the general idea behind model recycling and the concrete recycling strategies we propose to implement this concept. An overview of these strategies is shown in Figure 3.8, and we now discuss each of these in detail.



(a) Model Selection (b) Single Model Reuse (c) Model Stacking



(d) Model Voting (e) Clustering

Figure 3.8: Recycling Strategies

The design of the five recycling strategies began with an exploration of intuitive techniques such as model selection, single model reuse, and clustering, aimed at identifying the best historical model from the history. This approach evolved as we recognized similarity with ensemble methods, specifically model stacking and model voting, which traditionally combine predictions from different model variants at the same point in time to enhance overall prediction performance. However, our approach adapts

these ensemble methods by considering model variants that are trained at different points in time and treating each model as a black-box. By integrating both straightforward reuse and more complex ensemble tactics, these strategies collectively provide a wide range of techniques that not only enhance predictive accuracy but also provide ways to recycle historical models effectively.

3.2.1 Model Selection

As shown in Figure 3.8a, the model selection strategy chooses at inference time, between the best old model thus far and the latest RFS MLM. To make this work, a model selection *recycling* model has to be trained pre-inference that predicts what version of the model (old or latest) to use for prediction at inference time. Algorithm 2 shows the process involved in this strategy, while Algorithm 1 shows the process involved in finding the best old model from the model history (used by Algorithm 2).

pre-inference – As shown in Algorithm 2, for each window C, we first prepare (pre-inference) the training features (X_{train}) by combining the training data from the model version history including the current time window (C). The features are created from commit metrics and code changes as explained in Section 3.1.2.5. Second, before we can create the oracle (Y_{train}), we select the best old model from history, as shown in Algorithm 1. This old model can be chosen based on metrics like g-mean or F1-score of the historical models. Once the best old model is identified, to create the oracle (Y_{train}) we then determine the best model version between the best old model and RFS model for each commit in the training data based on their error ($abs(Y^i_{prob} - Y^i_{actual})$) on that commit. Finally, we train a *recycling* model able to select either the best old model or the RFS model using the prepared features (X_{train}) and oracle (Y_{train}).

during-inference – As new commits arrive, we use the *recycling* model to predict the best model version for each commit, then apply the predicted model version (i.e., old or new) to make predictions for the new commit.

Additionally, we also designed a variant of model selection where we do not train a *recycling* model, but during inference, we just average the probabilities of both models as $Y_{prob}^{final} = 1/2 * (Y_{prob}^{v_i} + Y_{prob}^{v_j})$. This variant requires inference from both old and new models.

Algorithm 1: Find Best Model from Model History with GMean for the Current Window (used by Algorithms 2 and 3)

Input:
C /* Current Time Window */
1 H_{all} /* Model Version History */
2 Y_{pred} /* Historical Models' Predicted Labels */
3 Y_{actual} /* Historical, Actual Labels */
Output: V /* Model Version with Best G-Mean Score */
4
5 **Function** `findBestOldModel`$(C, H_{all}, Y_{pred}, Y_{actual})$**:**
6 $maxScore \leftarrow 0, V \leftarrow -1$
7
/* Evaluate each model version one window at a time */
8 **foreach** $modelVersion \in H_{all}[C-15 : C-1]$ **do**
/* Note: ground truth (Y_{actual}) is same for all models */
9 $score = gMean(Y_{pred}[C-15 : C][modelVersion], Y_{actual}[C-15 : C])$
10 **if** $score > maxScore$ **then**
11 $maxScore = score$
12 $V = modelVersion$
13 **return** V
14 **End Function**

As major drawbacks, the model selection strategy adds a layer of complexity to the inference process either by requiring training a new *recycling* model or by requiring

Algorithm 2: Training a Model Selection *Recycling* Model for the Current Window

Input:
C /* Current Time Window */
1 K /* Past Knowledge of Models' Performance */
2 D /* Seen Training Data */
3 H_{all} /* Model Version History */
Output: M /* Model Selection Model */
4
5 **Function** `modelSelection`(C, K, D, H)**:**
6 $Y_{pred} \leftarrow K[C-15:C]['predictedLabel']$
7 $Y_{actual} \leftarrow D[C-15:C]['actualLabel']$
8 $V \leftarrow$ `findBestOldModel`$(C, H_{all}, Y_{pred}, Y_{actual})$
9
/* Prepare oracle Y_{train} for *model selection* recycling */
10 **foreach** $i \in D[C-15:C]['commitID']$ **do**
11 $Y_{train}[i] \leftarrow \arg\min(K[i]['error'][C], K[i]['error'][V])$
/* Prepare features X_{train} as per Section 3.1.2.5 */
12 $X_{train} \leftarrow prepareFeatures(D[C-15:C])$
13
/* Train RF to predict which model version to use for each commit */
14 $M = RandomForestClassifier().fit(X_{train}, Y_{train})$
15 **return** M
16 **End Function**

inference from both old and new models to aggregate predicted probabilities. However, it can be an effective strategy if the *recycling* model can identify changes in data distribution (drift) and choose an appropriate model that performed better on similar data samples in the past.

3.2.2 Single Model Reuse

As depicted in Figure 3.8b, this strategy is designed to identify the best old model from a given model version history during pre-inference time and use the selected best old model during inference. Algorithm 3 shows the process involved in this strategy, and again leverages Algorithm 1 to find the best old model from the model history.

pre-inference – As shown in Algorithm 3, for each window C, the pre-inference step involves finding the best old model from the available model history. In addition, we also train a RFS model, to use it as a historical model in the next window, although this model is ignored for the current window C.

during-inference – At inference time, unlike the model selection strategy, the single model reuse strategy exclusively utilizes the old model to detect bug-inducing commit IDs as they arrive.

The single model reuse strategy is a simple approach that does not require any additional training or inference time. Yet, as this strategy only uses a historical model, it may not be able to always find the best historical model for the latest time window. For example, when the data distribution changes significantly, the historical model may not be able to make accurate predictions.

Algorithm 3: Finding the Best Old Model Version for the Current Window using Single Model Reuse

Input:
C /* Current Time Window */
1 K /* Past Knowledge of Models' Performance */
2 D /* Seen Training Data */
3 H_{all} /* Model Version History */
Output: V /* Best Old Model */
4 **Function** `modelReuse`(C, K, H_{all})**:**
5 $Y_{pred} \leftarrow K[C-15:C]['predictedLabel']$
6 $Y_{actual} \leftarrow D[C-15:C]['actualLabel']$
7 $V \leftarrow$ `findBestOldModel`$(C, H_{all}, Y_{pred}, Y_{actual})$
8 **return** V
9 **End Function**

3.2.3 Model Stacking

As its name implies, model stacking utilizes stacked MLMs for inference. Those initially were proposed by Wolpert (1992), and found applications in various domains. The fundamental idea is to employ multiple models to generate predictions, then utilize a meta-model to combine the predictions from these base models, as illustrated in Figure 3.8c. In the case of JIT, the meta-model is trained to identify bug-inducing commits using the probabilities generated by the base models as features. The meta-model can be a simple linear regression or a more complex model such as a neural network.

The two main differences between traditional stacking and our stacking recycling approach are that traditional stacking re-trains both the stacked models and the meta-model simultaneously; however, we treat the stacked models as black boxes, thereby avoiding their re-training. Second, in practice, stacking combines models trained on the same data whereas we combine historical models trained on different data from different time windows up to and including the latest model.

The process can be split into two stages: *pre-inference* and *at-inference*. The pre-inference stage involves training the meta-model, and the at-inference stage involves using the meta-model to make predictions. The pre-inference stage can be further split into two stages: *filtering* of model version history and *training meta-model*.

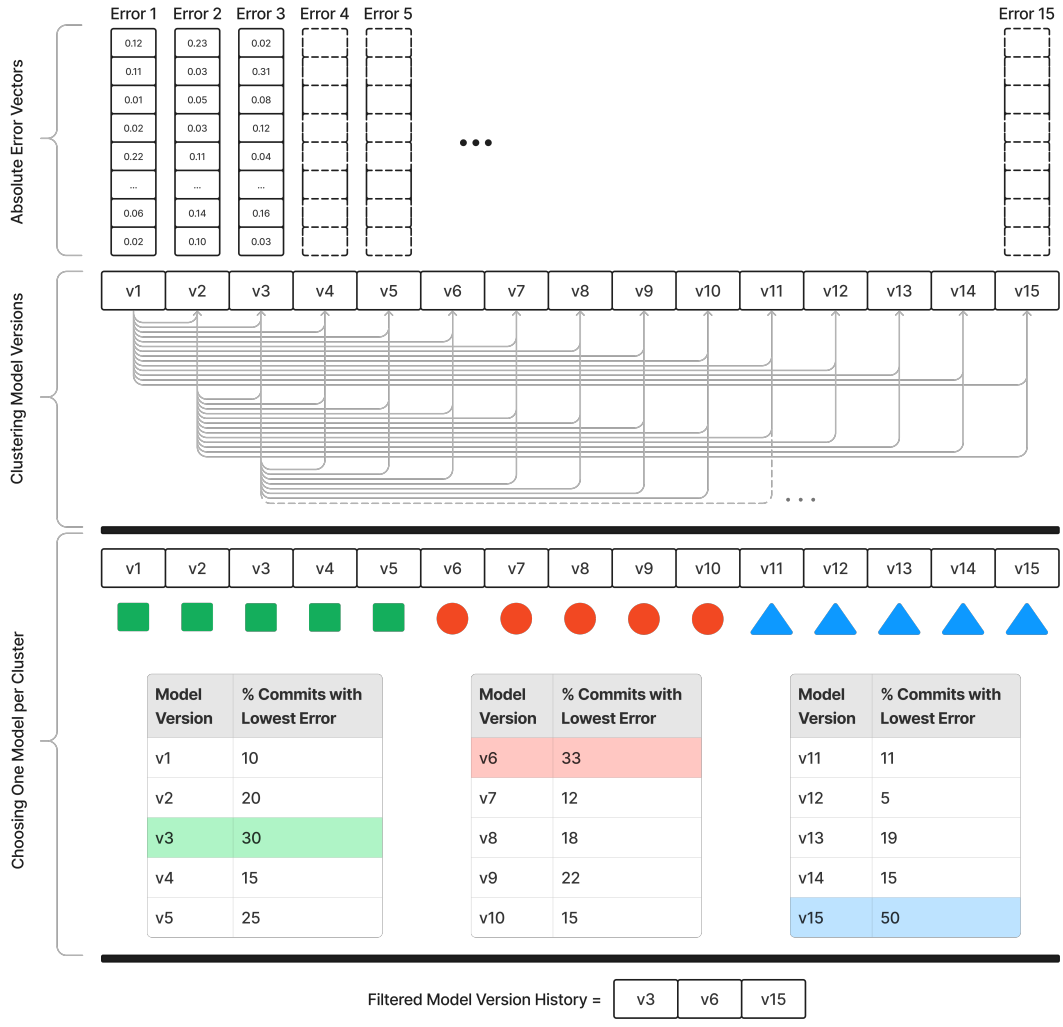


Figure 3.9: Filtering Model History using Statistical Tests

pre-inference/filtering – Using the entire model history might not be ideal for this

strategy, given that it employs all models for predictions. Hence, we employ statistical tests to identify the best models for stacking from history. As illustrated by Algorithm 4 and Figure 3.9, to find the most distinct set of models, we conduct a statistical test on the absolute error vectors ($Y_{error}(m_i)$ and $Y_{error}(m_j)$) of each pair of models. The error vectors are retrieved by finding the absolute error between actual and predicted class probabilities for both models m_i and m_j as defined earlier in Section 3.1.2.7, for each commit available in the model version history. The statistical tests on the error vectors help us to identify the most similar models in the model history, cluster those and select the best model from each cluster. We then use the selected models for stacking. We evaluate both the Wilcoxon signed-rank test (Woolson, 2007) and the Scott-Knott Effect Size Difference (ESD) test (Tantithamthavorn et al., 2018) to reduce the size of the model history. We apply Bonferroni correction (Weisstein, 2004) to adjust the p-value for multiple comparisons.

pre-inference/training meta-model – As explained in Algorithm 5, for each window C, we first prepare the training features (X_{train}) using the method explained in Section 3.1.2.5. Here, the oracle (Y_{actual}) includes labels for each commit indicating whether it is bug-inducing or not. Finally, we train the meta-model using the prepared features (X_{train}) and oracle (Y_{actual}).

at-inference – As new commits arrive, all stacked models predict probabilities for a commit being bug-inducing. We then use the meta-model to combine the probabilities from the stacked models to make predictions for the new commit.

Our historical stacking approaches leverage a simple logistic regression (LR) model or a more complex random forest (RF) model as our meta-models. We also evaluate two different feature space configurations for our meta-model. One uses only base

Algorithm 4: Filter Model Search Space using Statistical Tests for the Current Window (used by Algorithms 5 and 6)

Input:
C /* Current Time Window */
H_{all} /* Model Version History */
Y_{prob} /* Models' Predicted Probabilities */
Y_{actual} /* Actual Labels */
Output: $H_{filtered}$ /* Filtered Model Version History */
Function `filterModelHistory`$(C, H_{all}, Y_{prob}, Y_{actual})$**:**
  $modelsToRemove \leftarrow []$
  /* Create all pairs of chronologically ordered model versions for comparison e.g. (1, 2), (1, 3), (2, 3) for models 1, 2, 3 */
  $versionPairs \leftarrow combinations(H_{all})$

  foreach $(modelX, modelY) \in versionPairs$ **do**
    /* If a model is already set to be filtered out, skip checking for statistical significance */
    if $modelX \in modelsToRemove$
      or
      $modelY \in modelsToRemove$
    then
      continue

    /* Evaluate statistical similarity between the error vectors of the two models */
    $Y_{err}^{modelX} \leftarrow absolute(Y_{actual}[C-15:C] - Y_{prob}[C-15:C][modelX])$
    $Y_{err}^{modelY} \leftarrow absolute(Y_{actual}[C-15:C] - Y_{prob}[C-15:C][modelY])$
    $pValue = wilcoxon(Y_{err}^{modelX}, Y_{err}^{modelY})$

    /* If the models have statistically similar error vectors, remove one of the models in comparison, otherwise keep both since they both are distinct */
    if $pValue \geq ALPHA$ **then**
      /* Find the model version that more frequently has better absolute error than the other */
      $modelVersion \leftarrow findLosingModelVersion(Y_{err}^{modelX}, Y_{err}^{modelY})$
      $modelsToRemove.remove(modelVersion)$

  $H_{filtered} \leftarrow H_{all}[C-15:C].remove(modelsToRemove)$
  return $H_{filtered}$
End Function

Algorithm 5: Training Model Stacking *Recycling* Model for the Current Window

Input:
C /* Current Time Window */
K /* Past Knowledge of Models' Performance */
D /* Seen Training Data */
H_{all} /* Model Version History */
Output: M /* Meta Classifier Model */

Function `modelStacking`(C, K, D, H_{all})**:**
    $Y_{prob} \leftarrow K[C-15:C]['predictedProbability']$
    $Y_{actual} \leftarrow D[C-15:C]['actualLabel']$
    $H_{filtered} \leftarrow$ `filterModelHistory`$(C, H_{all}, Y_{prob}, Y_{actual})$

    /* Prepare features X_{train} as per Section 3.1.2.5 */
    $X_{train} \leftarrow prepareFeatures(D[C-15:C])$

    /* Train meta-model */
    $M = StackingClassifier($
        /* Use filtered model versions as base estimators */
        $baseEstimators = H_{filtered}$,
        /* Use Random Forest or Logistic Regression (Section 3.2.3) as meta classifier */
        $metaEstimator = RandomForestClassifier()$,
        /* Treat the base estimators as black boxes */
        $fitBaseEstimators = false$,
        /* Use commit features for meta classifier */
        $useCommitFeatures = true$,
)
    $M.fit(X_{train}, Y_{actual})$
    return M

End Function

(stacked) models' prediction probabilities as features, while the other configuration adds the commit features (i.e., commit metrics and code changes) in conjunction with the base models' prediction probabilities as features. When training the meta-model with only stacked models' probabilities, we utilize LR, as this configuration involves a smaller feature space. However, we use RF when training the meta-model with stacked model probabilities and original model features. The scikit-learn library's *StackingClassifier* (Pedregosa et al., 2011) is used to implement the model stacking strategy.

Furthermore, we evaluate this strategy using different data sampling configurations, which are as follows:

1. **all** – Utilize the full aggregated training data of the model history, including the training data of the RFS model.

2. **random** – Randomly sample training data (equal to the window size) from the aggregated training data of the model history.

3. **randomly resampled** – Randomly resampled training data (equal to the window size) from the aggregated training data of the model history based on class labels. Since the source dataset is imbalanced, this configuration ensures that the data points from each class are represented equally (50%-50%) in the training sample.

Model stacking is a more complex strategy than the other two strategies, as it requires training a meta-model and there are more than two models required for inference. However, it can be an effective strategy since involving more models trained on different data from different time windows can help the meta-model to generalize better.

3.2.4 Model Voting

The Model Voting strategy, similar to Model Stacking, utilizes stacked MLMs for inference. However, the key distinction lies in the approach used to combine the predictions from the base models. Figure 3.8d illustrates that Model Voting employs a straightforward voting mechanism. Unlike Model Stacking, there is no meta-model involved in this strategy. The process can be split into two stages: *pre-inference* and *at-inference*. The pre-inference stage involves filtering the model history and the at-inference stage involves using the filtered model history to make predictions.

pre-inference – As shown in Algorithm 6, for each window C, using the same filtering algorithm 4 used in Model Stacking, we filter the model history using statistical tests, similar to model stacking. The statistical tests help us to identify the most similar models (in terms of model predictions) in the model history as explained earlier in Figure 3.9. Finally, we create the voting classifier by creating an ensemble of the filtered model versions as base estimators.

at-inference – As new commits arrive, the base estimators predict probabilities for each commit. We then use the voting mechanism to combine the probabilities from the base estimators to make the final prediction for the new commit.

We implement the Model Voting strategy using mlxtend's *EnsembleVoteClassifier* (Raschka, 2018). By default, the *EnsembleVoteClassifier* re-trains the base estimators, but we configure it to treat them as black boxes. So, there is no training involved in this strategy. Additionally, instead of creating an ensemble of different models trained at the same time on the same data, we create an ensemble of historical models, including the latest model, trained on different data from different time windows.

Algorithm 6: Creating Model Voting *Recycling* Model for the Current Window

Input:
C /* Current Time Window */
1 K /* Past Knowledge of Models' Performance */
2 D /* Seen Training Data */
3 H_{all} /* Model Version History */
Output: M /* Meta Classifier Model */
4 **Function** `modelVoting`(C, K, D, H_{all})**:**
5 $Y_{prob} \leftarrow K[C-15:C]['predictedProbability']$
6 $Y_{actual} \leftarrow D[C-15:C]['actualLabel']$
7 $H_{filtered} \leftarrow$ `filterModelHistory`$(C, H_{all}, Y_{prob}, Y_{actual})$
8
9 $M = EnsembleVoteClassifier($
/* Use the filtered model versions as base estimators */
10 $baseEstimators = H_{filtered}$,
/* Use soft voting to combine predictions */
11 $voting = 'soft'$,
/* Treat the base estimators as black boxes */
12 $fitBaseEstimators = false$,
13 $)$
14 **return** M
15 **End Function**

We experiment with both hard and soft voting mechanisms. Hard voting is the simplest form of majority voting, where the class label ($\hat{y}$) is predicted based on majority (plurality) voting across all classifiers C_j ($\hat{y} = \textbf{mode}\{C_1(x), C_2(x), ..., C_m(x)\}$). On the other hand, soft voting chooses the class label with the highest cumulative probability across all models, retrieved by aggregating predicted probabilities p from each classifier C_j, i.e., $\hat{y} = \arg\max_i \sum_{j=1}^{m} w_j p_{ij}$, where w_j represents the weight assigned to the jth classifier and p_{ij} represents the predicted probabilities of each class label i by jth classifier. We do not explicitly provide weights to the $EnsembleVoteClassifier$ so it assigns uniform weights to each base estimator giving equal importance to predictions from each classifier.

The Model Voting strategy requires multiple JIT models for inference. It can be an effective strategy because the base models are trained on different data from different time windows, providing diverse knowledge to the ensemble, which can be beneficial. However, it can be challenging due to drift, since voting relies only on base model prediction probabilities and there is no meta-model to calibrate the predictions from the base models like the model stacking approach does.

3.2.5 Clustering

The clustering strategy is depicted in Figure 3.8e. Intuitively, this strategy leverages the *ball tree* clustering algorithm to learn the similarity between commits from the current model history based on features retrieved from their commit metrics and code changes. Then for inferencing new commits, it utilizes the prepared *ball tree* to identify the closest neighbours and selects the best-performing model on these neighbouring samples for inference. The process can be split into two stages: *pre-inference* and

at-inference. The pre-inference stage involves training the *ball tree* algorithm responsible for finding neighbouring samples, and the at-inference stage involves using the learned *ball tree* algorithm to make predictions.

pre-inference – As shown in Algorithm 7, for each window C, we first prepare the training features (X_{train}) using the same process as before. Second, we train the ball tree algorithm using the prepared features (X_{train}). We chose the *ball tree* algorithm because it is faster than both *brute force* and the *KD tree* algorithm for high-dimensional data (with ≥ 100 features).

Algorithm 7: Training Ball Tree Algorithm for the Current Window

Input:
C `/* Current Time Window */`
1 D `/* Seen Training Data */`
Output: M `/* Nearest Neighbor Model */`
2 **Function** `createNNModel(`C, D`):`
3 $X_{train} \leftarrow D[C-15:C]['features']$
4
`/* Create NN model using commit metrics and code changes as features */`
5 $M = NearestNeighbors(algorithm='ballTree')$
6 $M.fit(X_{train})$
7 **return** M
8 **End Function**

at-inference – At inference time, we query the *ball tree* algorithm for each new commit to retrieve the k neighbouring (i.e., most similar in terms of feature vector) commits observed in the past. The primary objective is to identify the best model from model history based on the models' performance on these neighbouring commits. To achieve this, we evaluate the mean-squared error (MSE) of all models from the model history at the current window on these neighbouring commits. Afterwards, we find the model

Algorithm 8: Inference with Clustering for the Current Window

Input:
C /* Current Time Window */
1 K /* Past Knowledge of Models' Performance */
2 N_C /* Nearest Neighbor Model for C */
3 H_{all} /* Model Version History */
4 kN /* # neighboring commits to consider in model selection */
5 X_{new} /* Unseen commit ID */
Output: P_{new} /* Predicted Label for X_{new} */
6 **Function** `modelVoting`$(C, K, N_C, H_{all}, kN, X_{new})$**:**
7 $commitIDs \leftarrow getKNearestCommitIDs(N_C, X_{new}, kN)$
8 $modelVersion \leftarrow getBestModelVersionByMSE(K[C-15, C], H_{all}[C-15:C], commitIDs)$
9 $P_{new} \leftarrow predict(modelVersion, X_{new})$
10 **return** P_{new}
11 **End Function**

with the best MSE from the evaluated model history on these neighbours, and use it for inference on the new commit as outlined in Algorithm 8. The implementation of the clustering strategy incorporates scikit-learn's *NearestNeighbors* (Pedregosa et al., 2011).

The process involved in inference can be seen as costly, as it requires evaluating the available model history on k neighbouring commits at inference time. To reduce the computation cost at-inference, in turn reducing delay in inference, we propose to use *absolute error lookup* by evaluating the models at the pre-inference time. At the pre-inference time, we evaluate the models on each commit in the training data seen by the available model history. We then store the absolute error of each model for each commit from the training data. At inference time, we retrieve the pre-evaluated absolute error of models for the k-neighbouring commits of the new commit. We then aggregate the absolute errors on k-neighbouring commits into mean-squared-error (MSE)

for each model and use MSE to identify the best model for the new commit.

Clustering is an intuitive strategy that selects the best model based on the similarity of the new commit to the commits observed in the past. Since this strategy identifies the best model at runtime, it requires evaluating multiple models at runtime, which can be computationally expensive. These computation costs can be reduced by using the *absolute error lookup* as discussed earlier. Clustering can be an effective strategy since it selects the best model based on its local performance on the most similar commits in the past.

3.3 Empirical Evaluation of Recycling Strategies

In this section, we evaluate the recycling strategies proposed in Section 3.2 to answer RQ2 and RQ3. For each research question, we first describe the approach we use to evaluate the recycling strategies. Then, we present the results of our case study and discuss the findings.

3.3.1 RQ2: What variant works best for each recycling strategy?

3.3.1.1 Motivation

This research question aims to find out what variant of each recycling strategy exploits the most value out of old models when compared to exclusively using a model that is retrained from scratch (RFS). We do not yet compare across recycling strategies.

3.3.1.2 Approach

Each proposed recycling strategy can be implemented with various configurations, which we refer to as a **variant**. These variants are created based on the different configuration dimensions discussed in Section 3.2 as listed in Table 3.6. For each recycling strategy, we empirically evaluate its multiple variants to find the best variant for this research question. The data used for this evaluation is the same as in RQ1 (Section 3.1). The baseline (RFS) is created by using the same setup as described in Sections 3.1.2.1 to 3.1.2.6.

For each variant, we collect performance vectors by evaluating the variant on the metrics discussed in Section 3.1.2.6 for each time window. As a baseline, we use the respective model that is retrained from scratch (RFS) for each window, e.g., RFS Random Forest model for Random Forest model variants and RFS Logistic Regression model for Logistic Regression model variants. We use the Wilcoxon signed-rank test (Woolson, 2007) to find the variants that statistically significantly perform better than the baseline, using Bonferroni correction (Weisstein, 2004) to adjust the p-value for multiple comparisons. Then, to evaluate the effect size for each variant with significant change, we use Cliff's delta effect size (Macbeth et al., 2011). Cliff's delta measures the observable effect of significant differences. We followed the values provided by Hess and Kromrey (2004) for interpreting the result of this value. Values smaller than 0.147 are negligible, values in the range [0.147, 0.33) are small, values in the range [0.33, 0.474) are medium, and values greater than 0.474 are large. For detailed p-value statistics and the effect size of each variant, please refer to the online replication package (Patel, 2023).

To identify whether a significant change is positive or negative, we calculate the

difference in mean metric performance between recycling and baseline performance vectors. If the value is negative, we mark that as performance degradation, while, if positive, we mark that as performance improvement compared to the baseline. For detailed differences in the mean metric performance of each variant, we again refer to the online replication package (Patel, 2023).

To summarize the statistical results for each variant, we show for each performance metric the number of projects having significant improvement or degradation out of the eight projects evaluated. To highlight whether the change is positive or negative, we show superscript "+" for improvement and "−" for degradation. We also highlight the cells based on the summarized effect size of the change. For a single variant, the summarized effect size is the most frequent effect size across all projects. In case of a tie, we superscript the cell with a "$*$" and highlight the cell with the greater effect size out of the two. For example, if a variant improves a performance metric in 4/8 projects by a small effect size and in 4/8 projects by a large effect size, we highlight the cell with a large effect size and superscript the cell with a "$*$".

Table 3.6: List of variants evaluated for each recycling strategy

Recycling Strategy	**Variant Dimension**	**Description**	**Evaluated Options**
Model Selection	Model Search Space	Length of model version history	Limited (=15), All

Continued on next page

Table 3.6 – continued from previous page

Recycling Strategy	**Variant Dimension**	**Description**	**Evaluated Options**
	Filtering Metric	Metric used for filtering the model search space	f1-score, g-mean
	Post Inference	Post inference method used to combine old and new model's prediction probabilities	average-probabilities
Single Model Reuse	Model Search Space	Length of model version history	Limited (=15), All
	Filtering Metric	Metric used for filtering the model search space	f1-score, g-mean
Model Stacking	Model Search Space	Length of model version history	Limited (=15), All
	Filtering Test	Statistical test used for filtering the model search space	Wilcoxon, ScottKnott

Continued on next page

Table 3.6 – continued from previous page

Recycling Strategy	**Variant Dimension**	**Description**	**Evaluated Options**
	With Commit Features	Flag indicating whether commit features (i.e. commit metrics and code changes) were used in training meta-model	yes/no
	Data Sample	Data sampling technique used to sample data from seen data	all, random (N=1000), randomly resampled (N=1000)
Model Voting	Model Search Space	Length of model version history	Limited (=15), All
	Filtering Test	Statistical test used for filtering the model search space	Wilcoxon, ScottKnott
	Voting Method	Voting method used to combine probabilities or votes from ensemble of classifiers	soft, hard

Continued on next page

Table 3.6 – continued from previous page

Recycling Strategy	Variant Dimension	Description	Evaluated Options
Clustering	k-Neighbors	# neighboring samples used to evaluate and select the best model for unseen data sample	20, 40

3.3.1.3 Results

3.3.1.3.1 Model Selection

For Random Forest (RF) and Logistic Regression (LR) models, the model selection recycling variants were able to significantly improve recall and g-mean compared to RFS for at least 25% of the projects, while they suffered significant performance degradation for Neural Network (NN) models. Tables 3.7, 3.8 and 3.9 show the number of projects that experienced significant change per metric for the RF, LR and NN models, respectively. The RF variants show similar improvements in recall, g-mean, AUC and F1 for 2-3 projects with effect sizes ranging from negligible to medium. However, there is a noticeable decline in precision for 1-2 projects, with negligible to small effect sizes. Determining a clear winner is challenging, as all variants display similar performance, ± one project (effect size). Among the LR variants, specifically #2, #4, #6, and #8, stand out as the top performers. They demonstrate improvements (up to a median of 2.6-4%) in precision, AUC, and F1, without any performance degradation. Other LR variants result in decreased recall (up to a median of 5.5%) while improving

other metrics but only for one or two projects. In contrast, NN variants consistently exhibit performance degradation across almost all projects, with large effect sizes observed for all metrics. This can be because of the high variance in the performance of NN models, which makes it difficult to find a good old model for inference. We will not discuss NN models further in this section since it seems that the model selection strategy does not work well for NN models.

For both RF and LR, variants #5-8 were able to perform similarly as variants #1-4, despite using a simpler configuration, since they only require a *limited* model search space, in turn requiring less computations. While for RF, variants #2, #4, #6 and #8 show improvements in more projects than the other four variants, these improvements are not as significant as the other four variants because of the negligible effect size in AUC, g-mean and F1. However, variants #4 and #8 can be argued to be slightly better choices in practice due to no significant degradation in precision. Similarly, for LR, variants #2, #4, #6 and #8 show improvements in more projects than the other four variants, with variants #4 and #8 showing slightly better performance in terms of g-mean, precision and F1.

filtering metric – Out of all variants, one can see that using *g-mean* as the filtering metric resulted in slightly more projects experiencing significant improvements in precision, g-mean and F1. This can be because the g-mean gives equal importance to both classes (Section 3.1.2.6). This characteristic helps the filtering process in finding the best old model that performs well on both class labels, in turn helping the strategy to pick the best old model for inference.

post-inference – We also observe that using the post-inference method (i.e., *averaging probabilities of the old and new models*) instead of using a *recycling* model, improved AUC in > 25% projects for all variants (i.e., #2, #4, #6 and #8), with negligible to medium effect size and resulted in lower recall improvements. This can be because using post-inference averages the probabilities of both old and new models, which favours the prediction towards a more confident model (i.e., class label with higher probability). Since confident predictions improve the ROC curve, it results in better AUC but this does not mean the model was able to make more correct predictions. This can happen if the model adjusts its predicted probabilities across the board, making them more extreme, but still maintaining the same decision boundary for classification.

Table 3.7: Model Selection (RF): # of projects (out of 8) experiencing significant change per evaluation metric

Model Search Space	#	Filtering Metric	Post Inference	Precision	Recall	G-Mean	AUC	F1
all	1	f1 score	n/a	2^{-*}	2^{+*}	0	0	0
	2	f1 score	avg. prob.	1^{-}	2^{+*}	1^{+}	2^{+}	1^{+}
	3	g-mean	n/a	1^{-}	2^{+*}	0	0	0
	4	g-mean	avg. prob.	0	2^{+*}	2^{+}	3^{+}	1^{+}
limited	5	f1 score	n/a	2^{-*}	2^{+*}	0	0	0
	6	f1 score	avg. prob.	1^{-}	2^{+*}	1^{+}	2^{+}	1^{+}
	7	g-mean	n/a	1^{-}	2^{+*}	0	0	0
	8	g-mean	avg. prob.	0	2^{+*}	2^{+}	3^{+}	1^{+}

Cliff's Delta Effect Size) Small Medium Large

Table 3.8: Model Selection (LR): # of projects (out of 8) experiencing significant change per evaluation metric

Model Search Space	#	Filtering Metric	Post Inference	Precision	Recall	G-Mean	AUC	F1
all	1	f1 score	n/a	3^{+}	$2^{+}2^{-*}$	2^{+}	1^{+}	1^{+}
	2	f1 score	avg. prob.	3^{+}	2^{+}	1^{+}	6^{+}	2^{+*}
	3	g-mean	n/a	5^{+}	2^{-*}	1^{+}	0	2^{+*}
	4	g-mean	avg. prob.	4^{+}	0	4^{+*}	5^{+}	3^{+}
limited	5	f1 score	n/a	3^{+}	$2^{+}2^{-*}$	2^{+}	1^{+}	1^{+}
	6	f1 score	avg. prob.	3^{+}	2^{+}	1^{+}	6^{+}	3^{+}
	7	g-mean	n/a	5^{+}	2^{-*}	2^{+*}	0	2^{+*}
	8	g-mean	avg. prob.	4^{+}	0	4^{+}	5^{+}	3^{+}

Cliff's Delta Effect Size) Small Medium Large

3.3.1.3.2 Model Reuse

The best model reuse recycling variants were able to significantly improve recall, g-mean and F1 for at least one project in case of RF, did not improve or degrade significantly in case of LR and degrade significantly in case of NN. Table 3.10 shows that the RF variants had very similar performance, with improvements in 2-4 (small-large effect, up to a median of 7.4-8%), 1 (small-medium effect, up to a median of 4.5-5.8%) and 1 (negligible-small effect, up to a median of 3.8-4%) project(s) for recall, g-mean and F1, respectively. These improvements come at the cost of degradation in 3 (small effect, up to a median of 4.5-4.8%), 0-1 (medium effect, up to a median of 3.7) and 0-1 (negligible effect, up to a median of 1%) project(s) for precision, g-mean and AUC metric performance, respectively. While variants #1 and #3 show higher effect size improvements than the other two variants, they lack in the overall number of projects

Table 3.9: Model Selection (NN): # of projects (out of 8) experiencing significant change per evaluation metric

Model Search Space	#	Filtering Metric	Post Inference	Precision	Recall	G-Mean	AUC	F1
all	1	f1 score	n/a	7^{-}	5^{-}	6^{-}	7^{-}	7^{-}
	2	f1 score	avg. prob.	7^{-}	5^{-}	6^{-}	7^{-}	7^{-}
	3	g-mean	n/a	7^{-}	6^{-}	6^{-}	7^{-}	7^{-}
	4	g-mean	avg. prob.	7^{-}	6^{-}	6^{-}	7^{-}	7^{-}
limited	5	f1 score	n/a	7^{-}	5^{-}	6^{-}	7^{-}	7^{-}
	6	f1 score	avg. prob.	7^{-}	6^{-}	6^{-}	7^{-}	7^{-}
	7	g-mean	n/a	7^{-}	6^{-}	6^{-}	7^{-}	7^{-}
	8	g-mean	avg. prob.	7^{-}	6^{-}	6^{-}	7^{-}	7^{-}

Cliff's Delta Effect Size) Small Medium Large

experiencing improvements.

Table 3.11 shows that LR variants did not make enough significant changes in the performance of any metrics to say that recycling the old model is better than retraining from scratch. Worse, Table 3.12 shows that the NN variants resulted in significant degradation in all projects for all metrics. Both model selection and model reuse strategies depend significantly on the selection of the best old NN model and result in significant degradation in performance due to its over-reliance on the old NN model. This suggests that old NN models are not able to generalize to other windows. This can again be because of the high variance in the performance of NN models over different windows, which makes it difficult to find a good old model for inference.

model search space – Although there is no clear winner, variants #3 and #4 use a simpler configuration, since they use a *limited* model search space, which requires less

computation.

filtering metric – Similar to model selection, the model reuse recycling strategy results in slightly more projects experiencing improvements in recall and g-mean when using *g-mean* as a filtering metric, although one project had a larger effect size improvement in recall for *f1-score* filtering. This can be because of the g-mean's characteristic of giving equal importance to both classes in the score (3.1.2.6). Since this recycling strategy reuses the old model as is, using g-mean as a filtering metric helps in selecting the model with the best performance on both class labels.

Table 3.10: Model Reuse (RF): # of projects (out of 8) experiencing significant change per evaluation metric

Model Search Space	#	Filtering Metric	Precision	Recall	G-Mean	AUC	F1
all	1	f1 score	3^{-}	2^{+*}	$1^{+}1^{-*}$	1^{-}	1^{+}
	2	g-mean	3^{-}	4^{+}	1^{+}	0	1^{+}
limited	3	f1 score	3^{-}	2^{+*}	$1^{+}1^{-*}$	1^{-}	1^{+}
	4	g-mean	3^{-}	4^{+}	1^{+}	0	1^{+}
		Cell Highlights (Cliff's Delta Effect Size)	Small	Medium	Large		

3.3.1.3.3 Model Stacking

The best model stacking variant for RF uses the *Wilcoxon* test for filtering model version history, *randomly resampled* or *all* samples for training, and both stacked model probabilities and *commit features*. In Table 3.13, RF variants #2, #3, #5, #6, #8, #9, #10, and #11 stand out with the highest number of projects experiencing significant improvements. They demonstrate improvements in recall (5-6 projects, medium

Table 3.11: Model Reuse (LR): # of projects (out of 8) experiencing significant change per evaluation metric

Model Search Space	#	Filtering Metric	Precision	Recall	G-Mean	AUC	F1
all	1	f1 score	2^-	1^+	0	0	0
	2	g-mean	1^+	0	0	1^+	0
limited	3	f1 score	1^+2^-	1^+	0	0	0
	4	g-mean	0	0	0	0	0
Cell Highlights (Cliff's Delta Effect Size)			Small	Medium	Large		

Table 3.12: Model Reuse (NN): # of projects (out of 8) experiencing significant change per evaluation metric

Model Search Space	#	Filtering Metric	Precision	Recall	G-Mean	AUC	F1
all	1	f1 score	7^-	5^-	6^-	7^-	7^-
	2	g-mean	7^-	6^-	7^-	7^-	7^-
limited	3	f1 score	7^-	5^-	6^-	7^-	7^-
	4	g-mean	7^-	6^-	7^-	7^-	7^-
Cell Highlights (Cliff's Delta Effect Size)			Small	Medium	Large		

to large effect, up to a median of 17-35.8%) and g-mean (2-4 projects, large effect, up to a median of 11.8-21.3%). However, there is a trade-off with a decline in precision (up to a median of 9%) across 2-6 projects with small effect sizes. Among these variants, #6 and #8 win by minimizing the number of projects with small effect size degradation in precision while significantly improving recall and g-mean in the majority of projects (up to 75% and 50% of projects, respectively).

The most effective LR model stacking variant adopts the *Wilcoxon* test for filtering model version history, *randomly resampled* or *all* samples for training, and both stacked model probabilities and *commit features*. Alternatively, variants using the *ScottKnott* test with *all* samples for training, combined with both stacked model probabilities and *commit features*, also prove effective. Table 3.14 shows that LR variants #1, #5, #6, #8, #9, #10, and #11 have the most projects with significant improvements in precision (4-6 projects, small to large effect, up to a median 5.4-14%), g-mean (4-6 projects, medium to large effect, up to a median 5.7-8%), AUC (6-7 projects, large effect, up to a median 5.8-11%), and F1 (5-7 projects, small to medium effect, up to a median 5-8.7%). However, there is a decrease in recall (up to a median of 9%) in one project with a medium effect size. Variants #1, #6, and #8 stand out by having only one or no project with medium effect size degradation in recall while significantly improving other metrics for at least 60% of projects with significantly small to large effect sizes.

The optimal NN model stacking variant uses either the *Wilcoxon* or *ScottKnott* test for filtering model version history, *randomly resampled* or *all* samples for training, and both stacked model probabilities and *commit features*. Finally, Table 3.15 shows that NN variants #1, #5, #6, #8, and #11 exhibit the most projects with significant improvements in g-mean (4-6 projects, large effect, up to a median 7.9-9.8%), AUC (6-7 projects, large effect, up to a median 7.6-10%), and F1 (6-7 projects, small to large effect, up to a median 9.8-12.4%). However, they show a decline in precision (up to a median 4.3-8.8%) and recall (up to a median 22-23%) in 1-2 projects with medium to large effect sizes. Variants #1, #5, #6, and #8 excel by using simpler configuration (*limited* model search space) and providing similar improvements in g-mean and AUC for

at least 50% and > 75% of projects with significantly large effect sizes, respectively.

filtering test – When comparing *ScottKnott* and *Wilcoxon* filtering test variants side-by-side, overall *Wilcoxon* performed better than *ScottKnott* by having improvements in slightly more projects with higher effect size. This can be because *Wilcoxon* resulted in more models remaining for stacking than *ScottKnott*, which means that more statistically different models provide input to the meta-model for training. These models are trained on different time windows, which can help the meta-model to generalize better on unseen commits.

data sampling technique – Out of all data sampling variants, *randomly resampled* worked best, followed by using *all* data, and using a *random sample*. This can be because the *random sample* variant outputs training data with skewed label distribution due to label imbalance in the source dataset. This label bias in data propagates to the model and negatively impacts the results. Compared to the *all* data variants, *randomly resampled* performed better while using less training data, in turn requiring less computation.

model search space – Using the full model search space ("*all*") results in a high number of models used for stacking and can increase the feature space, in turn increasing the compute time with no or negative effect on the improvements of g-mean, AUC and F1 results as shown in Tables 3.13, 3.14 and 3.15. This is why we did not try all possible variants under the *all* model search space variants. Between the *all* and *limited* model search space, *limited* seemed to provide improvements in more projects than using *all*.

commit features – Overall, using *commit features* along with stacked model probabilities resulted in fewer projects with degradation in precision than using only stacked

model probabilities, while giving a similar performance in other metrics, or resulted in more projects with improvements across multiple metrics. This can be because *commit features* can provide additional information to the model including information about underlying feature distribution changes between time windows, which can help the model to improve on classification mistakes made by base models. However, using *commit features* can increase the feature space and hence the training time.

3.3.1.3.4 Model Voting

The best model voting recycling variant for RF and LR uses the *Wilcoxon* test for filtering model version history, *limited* model version history and *soft* voting, while no significant improvements were observed for NN. Table 3.16 shows that RF variants #1 and #2 demonstrate the highest number of projects with significant improvements and the fewest instances of precision degradation (up to a median of 3.2%). Both variants show similar improvements in recall (1 project with a medium effect, up to a median of 7.5%) and AUC (1 project with a negligible effect, up to a median of 1.5%). However, variant #2 slightly outperforms by having a negligible effect size degradation in precision.

In Table 3.17, LR variants #1, #2, and #3 lead in projects with improvements across various metrics: precision (2 projects with a small effect, up to a median of 6%), recall (1 project with a medium-large effect, up to a median of 10.7%), AUC (2-3 projects with a medium effect, up to a median of 5.8%), and F1 (1-3 projects with a small effect, up to a median of 4.6%). While variant #2 shows no degradation and more projects with improvements, it only marginally outperforms the others.

Finally, Table 3.18 reveals that none of the NN variants succeed in improving model performance. This might be attributed to a generalization issue, where the models

Table 3.13: Model Stacking (RF): # of projects (out of 8) experiencing significant change per evaluation metric

Model Search Space	#	Filtering Test	With Commit Features	Data Sample	Precision	Recall	G-Mean	AUC	F1
limited	1	ScottKnott	yes	all	1^{-}	3^{+*}	2^{+*}	1^{+}	1^{+}
	2	ScottKnott	no	all	6^{-}	6^{+*}	3^{+*}	1^{+}	1^{+}
	3	ScottKnott	no	randomly resampled	6^{-}	5^{+}	2^{+*}	1^{+}	1^{+}
	4	ScottKnott	yes	random	$2^{+}1^{-}$	$2^{+}2^{-*}$	2^{-}	0	1^{-}
	5	ScottKnott	yes	randomly resampled	5^{-}	6^{+}	4^{+*}	1^{+}	1^{+}
	6	Wilcoxon	yes	all	2^{-*}	6^{+*}	4^{+*}	1^{+}	1^{+}
	7	Wilcoxon	yes	random	1^{+}	$2^{+}2^{-*}$	2^{-}	0	0
	8	Wilcoxon	yes	randomly resampled	4^{-}	6^{+*}	4^{+*}	1^{+}	1^{+}
	9	Wilcoxon	no	all	6^{-}	5^{+}	3^{+}	1^{+}	1^{+}
	10	Wilcoxon	no	randomly resampled	6^{-}	5^{+}	3^{+}	1^{+}	1^{+}
all	11	Wilcoxon	yes	randomly resampled	4^{-}	5^{+}	3^{+}	1^{+}	1^{+}

Cell Highlights (Cliff's Delta Effect Size) Small Medium Large

struggle to generalize well on unseen commits. Alternatively, it could be that filtered models were not different enough to be useful for voting.

Table 3.14: Model Stacking (LR): # of projects (out of 8) experiencing significant change per evaluation metric

Model Search Space	#	Filtering Test	With Commit Features	Data Sample	Precision	Recall	G-Mean	AUC	F1
limited	**1**	ScottKnott	yes	all	6^{+*}	$1^{+}1^{-*}$	5^{+}	7^{+}	7^{+*}
	2	ScottKnott	no	all	6^{+}	0	2^{+*}	4^{+}	2^{+}
	3	ScottKnott	no	randomly resampled	6^{+}	0	2^{+*}	4^{+}	2^{+}
	4	ScottKnott	yes	random	6^{+}	$1^{+}4^{-}$	$2^{+}1^{-}$	6^{+}	$3^{+}1^{-}$
	5	ScottKnott	yes	randomly resampled	4^{+}	5^{+}	5^{+}	6^{+}	6^{+}
	6	Wilcoxon	yes	all	6^{+}	$1^{+}1^{-*}$	5^{+}	7^{+}	6^{+}
	7	Wilcoxon	yes	random	6^{+}	$1^{+}4^{-}$	$2^{+}1^{-}$	6^{+}	$3^{+}1^{-*}$
	8	Wilcoxon	yes	randomly resampled	5^{+}	3^{+}	6^{+}	7^{+}	6^{+}
	9	Wilcoxon	no	all	6^{+}	0	4^{+*}	6^{+}	5^{+*}
	10	Wilcoxon	no	randomly resampled	6^{+}	0	5^{+}	6^{+}	5^{+}
all	11	Wilcoxon	yes	randomly resampled	5^{+}	4^{+}	5^{+}	6^{+}	6^{+*}

Cell Highlights (Cliff's Delta Effect Size): Small, Medium, Large

model search space – Similar to model stacking, using *all* model version history impacted performance negatively, so we did not test other variants with the *all* model search space option.

Table 3.15: Model Stacking (NN): # of projects (out of 8) experiencing significant change per evaluation metric

Model Search Space	#	Filtering Test	With Commit Features	Data Sample	Precision	Recall	G-Mean	AUC	F1
limited	**1**	ScottKnott	yes	all	$5^{+}1^{-}$	$2^{+}2^{-}$	$4^{+}1^{-}$	7^{+}	7^{+}
	2	ScottKnott	no	all	0	1^{+}	0	0	0
	3	ScottKnott	no	randomly resampled	0	1^{+}	0	0	0
	4	ScottKnott	yes	random	$5^{+}1^{-}$	$2^{+}2^{-}$	$2^{+}2^{-}$	6^{+}	$2^{+}1^{-*}$
	5	ScottKnott	yes	randomly resampled	1^{+}	4^{+}	6^{+*}	6^{+}	6^{+}
	6	Wilcoxon	yes	all	$5^{+}1^{-}$	$2^{+}2^{-}$	$4^{+}1^{-}$	7^{+}	7^{+}
	7	Wilcoxon	yes	random	$5^{+}1^{-}$	$2^{+}2^{-}$	$2^{+}2^{-}$	7^{+}	$2^{+}1^{-}$
	8	Wilcoxon	yes	randomly resampled	1^{+}	4^{+}	6^{+*}	6^{+}	6^{+}
	9	Wilcoxon	no	all	0	2^{+*}	0	1^{+}	0
	10	Wilcoxon	no	randomly resampled	0	1^{+}	0	0	0
all	11	Wilcoxon	yes	randomly resampled	1^{+}	4^{+}	6^{+*}	6^{+}	6^{+}

Cell Highlights (Cliff's Delta Effect Size) — Small — Medium — Large

voting method – Using *hard* voting resulted in performance degradation for RF, improved performance for LR models and negligible improvements for NN. The reason behind this variance in performance can be that the correct predictions have a higher

chance of getting overshadowed by other models in the ensemble, as the *hard* voting variant relies on majority voting using models' predicted labels.

filtering test – Overall, using the *Wilcoxon* test for filtering model version history works slightly better for the model voting recycling strategy. This can be because the *Wilcoxon* filtering again results in more models than *ScottKnott*, i.e., more models are used in the ensemble than *ScottKnott*. This can lead to better performance, since voting relies only on the predicted probabilities of the models as input.

Table 3.16: Model Voting (RF): # of projects (out of 8) experiencing significant change per evaluation metric

Model Search Space	#	Filtering Test	Voting Method	Precision	Recall	G-Mean	AUC	F1
limited	1	ScottKnott	soft	1^{-}	1^{+}	0	1^{+}	0
	2	Wilcoxon	soft	2^{-}	1^{+}	0	1^{+}	0
	3	Wilcoxon	hard	1^{-}	2^{+}	1^{-}	1^{+}	1^{-}
all	4	Wilcoxon	soft	2^{+*}	$1^{+}4^{-}$	4^{-}	1^{+}	2^{-}
			Cell Highlights (Cliff's Delta Effect Size)	Small	Medium	Large		

3.3.1.3.5 Clustering

Although model clustering sounds like an intuitive way for recycling, it does not result in any performance gain; instead, it degrades the performance by a large margin for all models, i.e., RF, LR and NN. Tables 3.19, 3.20 and 3.21 show that all variants showed a significant reduction in metric performance for almost all projects, without significant improvement.

Table 3.17: Model Voting (LR): # of projects (out of 8) experiencing significant change per evaluation metric

Model Search Space	#	Filtering Test	Voting Method	Precision	Recall	G-Mean	AUC	F1
limited	1	ScottKnott	soft	2^{+*}	$1^{+}1^{-*}$	2^{+*}	2^{+}	2^{+}
	2	Wilcoxon	soft	2^{+*}	1^{+}	0	3^{+}	3^{+}
	3	Wilcoxon	hard	2^{+*}	1^{+}	0	3^{+}	1^{+}
all	4	Wilcoxon	soft	4^{+}	$1^{+}4^{-}$	1^{-}	3^{+}	1^{+}
Cell Highlights (Cliff's Delta Effect Size)				Small	Medium	Large		

Table 3.18: Model Voting (NN): # of projects (out of 8) experiencing significant change per evaluation metric

Model Search Space	#	Filtering Test	Voting Method	Precision	Recall	G-Mean	AUC	F1
limited	1	ScottKnott	soft	0	0	0	0	0
	2	Wilcoxon	soft	0	1^{-}	0	0	0
	3	Wilcoxon	hard	1^{+}	0	0	0	0
all	4	Wilcoxon	soft	2^{-*}	1^{+}	2^{-*}	2^{-*}	2^{-*}
Cell Highlights (Cliff's Delta Effect Size)				Small	Medium	Large		

k-neighbors – Increasing the # of neighbours in variant #2 to twice that of variant #1 did not make any improvements in performance metrics. Given the inclusion of 15 different models in the search space, relying solely on the model with the lowest MSE may not always be the optimal approach. This is because the top-performing model in terms of MSE could potentially be overly specialized to the training data, leading to

overfitting issues. As a result, the selection process tends to favour models that have been trained on a time window containing most of the neighbouring commits. This inherent bias may help explain why the clustering strategy did not yield favourable results in our experimental setup.

Table 3.19: Clustering (RF): # of projects (out of 8) experiencing significant change per evaluation metric

#	k-Neighbors	Precision	Recall	G-Mean	AUC	F1
1	20	7⁻	7⁻	7⁻	7⁻	7⁻
2	40	7⁻	7⁻	7⁻	7⁻	7⁻

Cell Highlights (Cliff's Delta Effect Size): Small Medium Large

Table 3.20: Clustering (LR): # of projects (out of 8) experiencing significant change per evaluation metric

#	k-Neighbors	Precision	Recall	G-Mean	AUC	F1
1	20	7⁻	7⁻	7⁻	7⁻	7⁻
2	40	7⁻	7⁻	7⁻	7⁻	7⁻

Cell Highlights (Cliff's Delta Effect Size): Small Medium Large

Table 3.21: Clustering (NN): # of projects (out of 8) experiencing significant change per evaluation metric

#	k-Neighbors	Precision	Recall	G-Mean	AUC	F1
1	20	7^{-}	6^{-}	6^{-}	7^{-}	7^{-}
2	40	7^{-}	7^{-}	6^{-}	7^{-}	7^{-}

Cell Highlights (Cliff's Delta Effect Size) Small Medium Large

Summary for Research Question 2

Model Selection and Model Reuse strategy variants performed very similarly to each other in terms of prediction performance, while Model Stacking and Model Voting strategy variants performed better when the *Wilcoxon* test was used for filtering model history. Although Clustering sounds like an intuitive way for recycling, it did not result in any performance gain; instead, it degraded the performance by a large margin. All strategies except Clustering showed significant improvements in at least one project for Random Forest (RF) and Logistic Regression (LR) models. However, none of the strategies except Model Stacking showed significant improvements in any project for Neural Network (NN) models.

3.3.2 RQ3: Which recycling strategy performs the best?

3.3.2.1 Motivation

After evaluating the variants of each recycling strategy in isolation in RQ2, we want to determine which strategy works best overall. Additionally, while a recycling strategy may work best in terms of the model's performance, we want to understand the trade-off between inference latency and model performance for each recycling strategy, because some strategies require one extra model inference to determine the best historical model, while others require inference of more than two historical models.

3.3.2.2 Approach

This research question builds on RQ2. Having found the best variant(s) for each recycling strategy, we compare each recycling strategy using its best variant. We reuse the performance results of RQ2, this time to determine the best variant across all recycling strategies.

As the best RF variant for each recycling strategy, we use variant #8 from model selection, variant #4 from model reuse, variant #8 from model stacking and variant #2 from model voting, respectively, because these showed the best performance in terms of # projects across evaluation metrics while using simpler configurations. We do not include any variant from clustering in this comparison because it did not show any significant improvements in any of the projects. Using the same criteria, we selected LR variants #8, #8 and #2 for model selection, model stacking and model voting, respectively. We do not include any variant from model reuse and clustering in this comparison, because they did not show any significant improvements in any of the projects. For NN, we selected #8 for model stacking only, since other strategies did not show any

significant improvements in any of the projects.

Furthermore, we evaluate the inference latency of the selected best variant for each recycling strategy. As a baseline, we measure the inference latency of the RFS model on a given window. The inference latency is retrieved by measuring the time to inference on each test sample for that window, and doing this across all windows yielding $\sum_{project=1}^{project=8}(\#\text{ Evaluated Windows})_{project} * \text{Test Size}(=200)$ measurements. We only collect one time measurement per test sample since there are enough test samples for each project to generalize the latency. For example, the minimum # test samples for the project with the least # windows (9 for ActiveMQ) is 1,800. We then compare each recycling strategy's best variant based on its metric performance and on its inference latency.

We perform statistical tests on the latency measurements of each recycling strategy's best variant against the baseline (no recycling). We use the Wilcoxon signed-rank test to determine if the latency measurements of each recycling strategy's best variant are significantly different from the baseline and we use the Cliff's Delta effect size to determine the magnitude of the difference between them. Additionally, to help understand latency differences, we measure the median latency ratio of the recycling strategy's best variant to the baseline.

3.3.2.3 Results

The model stacking recycling strategy gives the best improvements in terms of prediction performance in recall, g-mean and F1 among all recycling strategies tested. Tables 3.22, 3.23 and 3.24 show the significant performance metric changes for each best variant of RQ2 for all models (RF, LR and NN). For RF, out of all recycling strategies

evaluated in this study, model stacking showed a significant increase in 4/8 and 6/8 projects for g-mean and recall respectively (compared to RFS), with large effect sizes, despite a decrease in precision in 4/8 projects with small effect sizes. For LR, model stacking showed a significant increase in all metrics, while improving performance for $\geq 75\%$ of projects for g-mean, AUC and F1, with medium-large effect sizes. Finally, for NN, model stacking was the only strategy that improved performance with 6/8 projects in g-mean, AUC and F1. Model stacking is an ensemble approach, and when used with multiple generalizers trained on different time windows, it seems to get the most out of old estimators across all models.

Model selection and model voting are second-best recycling strategies in terms of prediction performance. There is no clear second-best recycling strategy between model selection and model voting. For RF and LR, The model selection recycling strategy shows improvements across more projects with a negligible-medium effect size. On the other hand, for RF, model voting shows slight improvements in recall and AUC, while for LR, it shows higher effect size improvements in Recall, AUC and F1 for comparatively more projects. Finally, for NN, both model selection and model voting showed no significant improvements as discussed earlier. Model selection uses the *recycling* model's prediction to decide which model to reuse for inference, whereas model voting is an ensemble approach that uses the predictions of multiple models to decide the final prediction. In other words, model selection, while using fewer models than model voting, can provide similar performance improvements if not better than model voting.

The model reuse recycling strategy comes last in terms of prediction performance. While it was able to boost the performance of recall and g-mean in some projects, it

led to precision degradation in more projects than it was able to improve g-mean or F1. Additionally, it was only able to significantly improve performance for RF models, not for LR and NN. This can be because the model reuse recycling strategy's focus on reusing only the best old model can sometimes result in a model that is not well suited for the current data distribution.

Table 3.22: # of projects (out of 8) experiencing significant change for the best recycling strategy variants (RF)

Recycling Strategy	Precision	Recall	GMean	AUC	F1
Model Selection	0	2^{+*}	2^{+}	3^{+}	1^{+}
Single Model Reuse	3^{-}	4^{+}	1^{+}	0	1^{+}
Model Stacking	4^{-}	6^{+}	4^{+}	1^{+}	1^{+}
Model Voting	2^{-}	1^{+}	0	1^{+}	0

Cell Highlights (Cliff's Delta Effect Size): Small, Medium, Large

Table 3.23: # of projects (out of 8) experiencing significant change for the best recycling strategy variants (LR)

Recycling Strategy	Precision	Recall	GMean	AUC	F1
Model Selection	4^{+}	0	4^{+}	5^{+}	3^{+}
Model Stacking	5^{+}	3^{+}	6^{+}	7^{+}	6^{+}
Model Voting	2^{+*}	1^{+}	0	3^{+}	3^{+}

Cell Highlights (Cliff's Delta Effect Size): Small, Medium, Large

Inference latency increases by 6-11x for model voting, 7-17x for model stacking, 2x for model selection, and remains the same for model reuse compared to the baseline. Figures 3.10, 3.11 and 3.12 show the inference latency results for each best variant under each recycling strategy tested for each model, while Tables 3.25, 3.26 and 3.27

Table 3.24: # of projects (out of 8) experiencing significant change for the best recycling strategy variants (NN)

Recycling Strategy	Precision	Recall	GMean	AUC	F1
Model Stacking	1^{+}	4^{+}	6^{+*}	6^{+}	6^{+}

Cell Highlights (Cliff's Delta Effect Size): Small, Medium, Large

show the difference factor between inference latency of baseline and recycling for each project and model alongside statistical test results. The baseline shows results when only using the latest model for every inference request.

Both model voting and model stacking result in significantly higher inference latency than the baseline, i.e., six to seventeen times. Between the two, model stacking requires higher inference times because of the additional computations required for meta-model inference. Tables 3.25, 3.26 and 3.27 and Figures 3.10c 3.11b and 3.12a show that these slowdowns are statistically significant in all projects and models with large effect sizes, with the median inference time ranging from 6-11x baseline for model voting and 7-17x baseline for model stacking. This is due to the ensemble nature of these two recycling strategies, i.e., we are doing inferencing on multiple models stacked together. Note that this may differ with the choice of model history filtering approach, because, in our study, we saw that the Wilcoxon test-based variant resulted in generally more than three models and sometimes up to ten or more after filtering. In contrast, the number of models stacked is significantly lower with the Scott Knott test-based variants. Even though both recycling strategy variants in the table used the Wilcoxon test for filtering, model stacking has a higher inference delay than model voting. This is because the selected model stacking recycling variant requires using the

meta-model inference as well as *commit features* alongside stacked model probabilities, which increases the feature space and inference time.

Model selection results in twice the inference latency of the baseline because of the additional inference required for each inference request. This is because the best variant tested for model selection averages probabilities of two predictions, meaning it always requires inference of two models before it can provide a final prediction. Hence, as expected, Tables 3.25 and 3.26 show that model selection has statistically significant differences in latency in all projects, with large effect sizes and with a median inference time of $\sim$ 2x baseline.

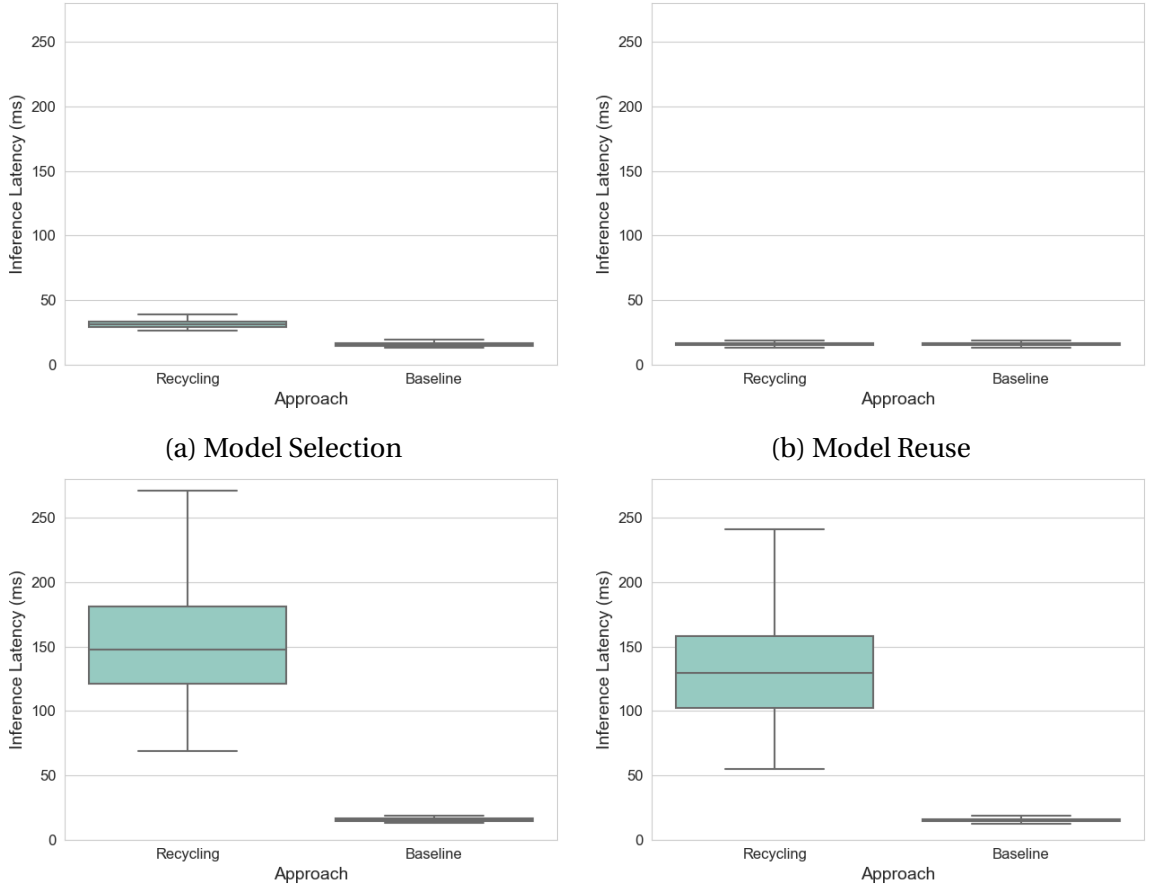
The model reuse recycling strategy results in the same inference latency as the baseline because it requires a single model inference for each inference request. For model reuse strategy, in both cases we are only doing inference with one model, i.e., inference with the old recycled model or baseline inference with the RFS model. Table 3.25 shows that the model reuse recycling strategy has a statistically significant difference in latency in only five projects, but with negligible effect size and median inference time being similar to baseline (1x).

The model selection and model stacking variants provide a good trade-off between inference latency and model performance. In practice, depending on the importance of inference latency or performance, an appropriate strategy can be chosen. Both model selection and model stacking are good strategies because of the trade-off between their metric performance and inference latency. Model selection, while giving performance improvements in at least 25% of projects, results in inference latency of 2x baseline. On the other hand, model stacking, while providing performance improvements in at least 50% of projects, results in inference latency of 7-17x baseline.

Looking at model voting, it improves the performance in only $12-37\%$ of projects, while still resulting in inference latency of 6-11x baseline. Lastly, model reuse, while resulting in the same inference latency as the baseline and improving performance in $12-50\%$ of projects, also degrades precision for 37% of the projects.

The decision on a model recycling strategy is ultimately left to practitioners, guided by our analysis of the trade-off between predictive performance and inference latency. While inference latency remains a crucial consideration, it is not the sole determinant when opting for a recycling strategy. In scenarios where the model operates in real-time systems, inference latency takes precedence over model performance. Conversely, in batch systems, model performance outweighs inference latency.

Ensemble approaches can leverage parallel inference, diminishing the significance of inference latency in the selection of a recycling strategy. Our latency analysis reveals that some recycling strategies exhibit superior predictive performance but incur a substantial increase, ranging from 6 to 17 times, in inference latency compared to the baseline. Notably, this practical limitation can be mitigated by adopting parallel inference for stacked base models instead of a sequential approach. Furthermore, it is essential to stress that the observed increase in inference latency, although noteworthy, consistently remains within the acceptable fraction-of-a-second range (< 300 milli-seconds), aligning precisely with the demands of the intended application scenarios for just-in-time software defect prediction applications. Take, for example, pull request review systems, where predictions occur while developers await the review's completion, which could take hours or even days.



(a) Model Selection

(b) Model Reuse

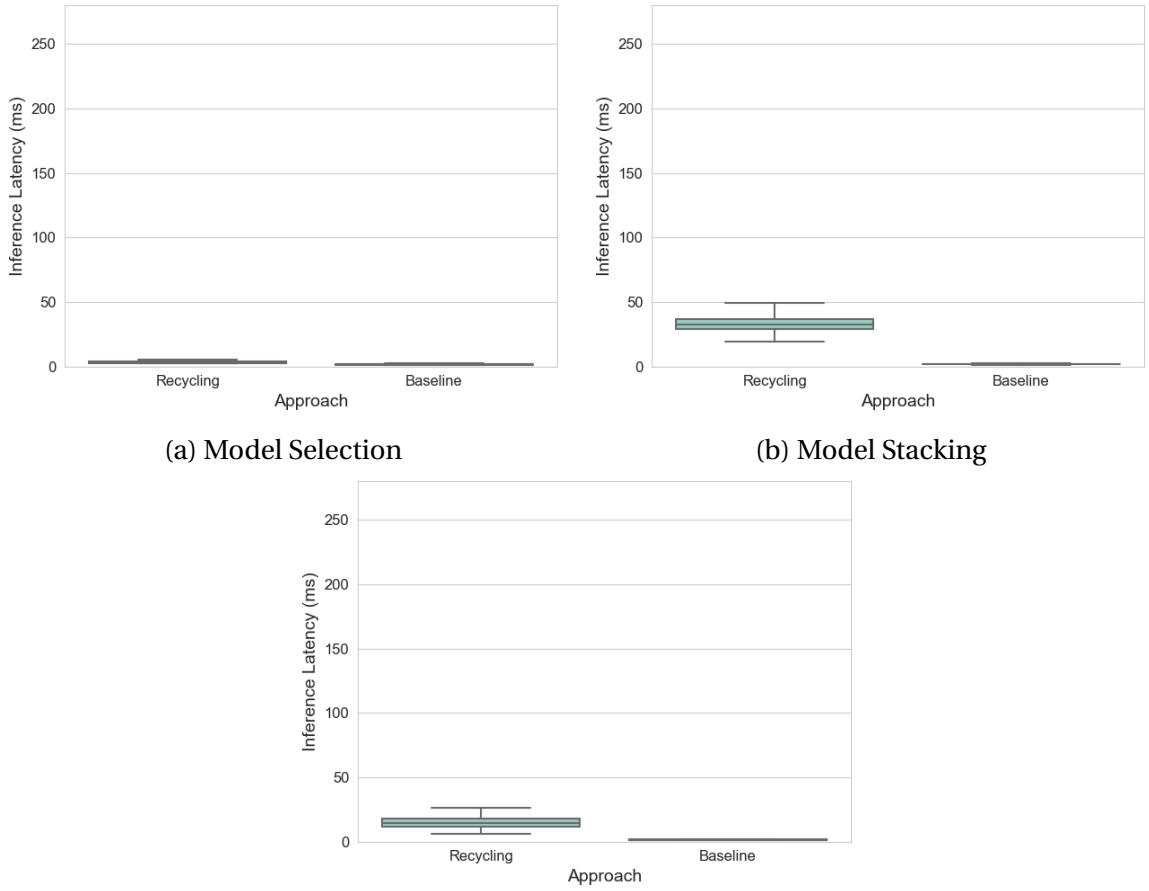
(c) Model Stacking

(d) Model Voting

Figure 3.10: Inference Latency per Recycling Strategy (RF)

Summary for Research Question 3

Model selection provides improvements in at least ∼25% of projects with 2x inference latency for RF and LR models, model stacking provides improvements in at least ∼50% of projects with 7-17x inference latency for all models, model voting provides improvements in ∼12-37% of projects with 6-11x inference latency for RF and LR, and model reuse provides improvements in ∼12-50% of projects with no inference latency increase for only RF. Based on the trade-off between inference latency and model performance, model selection and model stacking seem to be the best recycling strategies.



(a) Model Selection

(b) Model Stacking

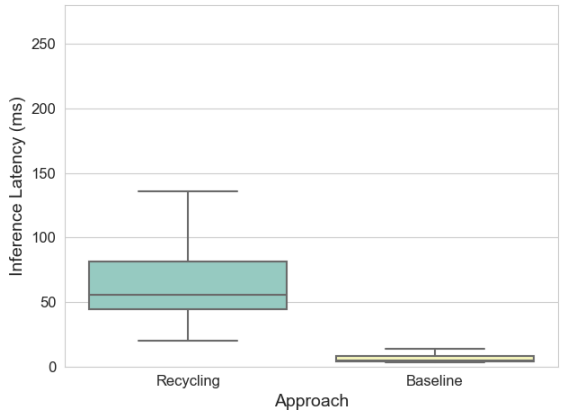
(c) Model Voting

Figure 3.11: Inference Latency per Recycling Strategy (LR)

3.4 Threats to Validity

3.4.1 Internal Validity

In our experimental setup, we employed a sliding window technique to create data splits, utilizing a window size of one thousand samples. The choice of window size significantly impacts the number of samples available for training a single model. Due to the infeasibility of evaluating all possible window sizes, we addressed this potential



(a) Model Stacking

Figure 3.12: Inference Latency per Recycling Strategy (NN)

Table 3.25: Median latency ratio of recycling to baseline and statistical test results for inference latency of the best variant of each recycling strategy ($\alpha = 0.05$) (RF)

Recycling Strategy	Project							
	ActiveMQ	Camel	Cassandra	Flink	Groovy	HBase	Hive	Ignite
Model Selection	2.016	2.017	2.017	2.029	1.996	1.990	1.988	2.023
Model Reuse	1.001	1.004		1.012	1.003		1.004	
Model Stacking	7.598	10.356	8.889	10.676	9.287	9.001	8.598	9.325
Model Voting	6.757	9.232	7.811	9.453	8.369	8.230	7.652	7.994

Cell Highlights (Cliff's Delta Effect Size): Small Medium Large

Note: *blank cells above indicate that there was no statistically significant difference between the latency of the baseline and the recycling strategy for that project.*

threat to validity by conducting additional analysis using a window size of two thousand samples alongside our primary window size of one thousand samples. However, this did not show any significant improvements over the original window size. The results of these additional study results can be found in the replication package (Patel, 2023).

Table 3.26: Median latency ratio of recycling to baseline and statistical test results for inference latency of the best variant of each recycling strategy ($\alpha = 0.05$) (LR)

Recycling Strategy	Project							
	ActiveMQ	Camel	Cassandra	Flink	Groovy	HBase	Hive	Ignite
Model Selection	1.962	1.982	1.925	2.046	2.002	1.918	1.956	1.960
Model Stacking	16.409	17.811	15.246	16.935	16.052	15.566	12.900	17.015
Model Voting	10.105	11.612	8.141	11.723	8.974	9.856	6.921	11.057

Table 3.27: Median latency ratio of recycling to baseline and statistical test results for inference latency of the best variant of each recycling strategy ($\alpha = 0.05$) (NN)

Recycling Strategy	Project							
	ActiveMQ	Camel	Cassandra	Flink	Groovy	HBase	Hive	Ignite
Model Stacking	10.617	12.001	10.723	11.460	12.072	8.644	9.766	13.377

Similarly, we maintained a window shift of two hundred samples for all analyses, which controls the overlap of data samples between consecutive windows. To explore the impact of different window shifts, we further tested our best variants with a window shift of three hundred samples. The results showed some improvements in the baseline but they are not significantly better than the original configuration settings used in this thesis. The results of these additional study results can be found in the replication package (Patel, 2023).

In Section 3.1, we propose several hypotheses to account for instances where older models outperform newer ones. While we hypothesize that such occurrences may stem from changes in project evolution, as previously discussed, our work does not provide definitive evidence to support this claim. We acknowledge the need for further investigation into this and consider it as a potential future work.

In the limited model history approaches, we set the model history size to fifteen, considering only the fifteen most recent model versions for recycling strategies. This limitation was a reasonable choice to avoid delving too far back into the history of models. Given that the cost of computing resources and time required to train and evaluate models increases with the model history size, we decided not to continue testing other variants with a larger model history size. However, to ensure a thorough investigation, we also conducted experiments without limit on the model history size, enabling us to explore potential improvements beyond the limited model history setting. Again, the results do not show improvements over the limited model history setting. This has been discussed in the results section of this chapter.

We adopted hyperparameters from the JITLine model (Pornprasit and Tantithamthavorn, 2021) as a baseline for our RF model training, hyperparameters from the DeepJIT model (Hoang et al., 2019) for our NN models and the default configuration for LR provided by scikit-learn (Pedregosa et al., 2011). Although hyperparameter tuning could potentially enhance the performance of our MLM, we deliberately avoided this process to maintain consistency across our experiments. Similar to Falessi et al. (2021), our focus does not lie in improving the performance of the defect prediction MLM. But rather on exploring the possibility of recycling old models to enhance the performance of existing MLMs, treating them as black boxes. By conducting these supplementary experiments and maintaining specific choices in hyperparameter tuning, we aim to provide a robust and comprehensive evaluation of our approach while mitigating potential internal threats to the validity of our study.

In our DeepJIT experiments, we do not use commit messages as features. This is because we want consistency in the features used across all models (RF, LR and NN).

However, one can argue that models like DeepJIT can benefit additionally from textual features like commit messages. Therefore, we conducted additional experiments with DeepJIT using commit messages as features along with both commit metrics and code change features. The results when compared with DeepJIT without commit messages showed statistical significance, but with negligible effect size, in almost all metrics and projects. The results of these additional study results can be found in the replication package (Patel, 2023).

3.4.2 External Validity

Our research used the ApacheJIT dataset (Keshavarz and Nagappan, 2022) and Just-In-Time defect prediction models to conduct our experiments. It is essential to acknowledge that this dataset primarily comprises popular Apache projects and their associated known bugs, which may limit its representativeness compared to all software projects. Furthermore, since the projects within the dataset are written in the Java programming language, there may be concerns about the generalizability of our results to other programming languages. However, the commits level features used in this thesis are language-independent and can be extracted from any software project. Since we use JIT defect prediction models to conduct our experiments, our results may not apply to other software analytics models.

However, despite these considerations, we firmly believe that our approach's fundamental principles and methodologies are applicable and generalizable beyond the confines of the ApacheJIT dataset or JIT defect prediction models. This adaptability extends to other software analytics techniques, further enhancing its potential utility. Therefore, while recognizing the specific context of our case studies, we are optimistic

that our findings can serve as a valuable foundation for future research and practical applications across diverse software projects and programming languages.

Our study uses JIT defect prediction models leveraging random forest, logistic regression and neural network models. These models fall under traditional machine learning models, and are widely used for defect prediction (Zhao et al., 2023). Recently, Large Language Models (LLMs) have started to revolutionize many fields. However, even today, many companies are still training and deploying the majority of their own bread-and-butter non-LLM machine learning models (deep learning, traditional classification models, etc.). McKinsey's report[4] emphasizes that despite the rapid progress of generative AI, traditional AI applications (like advanced analytics and machine learning) still contribute more to the overall potential value (11.0-17.7 trillion dollars) compared to the incremental impact of new generative AI use cases (2.6-4.4 trillion dollars). As such model recycling techniques will remain relevant in the foreseeable future.

As to the applicability of model recycling to LLMs, the emergence of techniques like frugalGPT (Chen et al., 2023) offer an efficient and cost-effective approach to using LLMs by sequentially cascading multiple LLMs, from cheapest to most expensive. With various LLM APIs available, each with unique performance and costs, strategic selection based on query requirements can reduce expenses and enhance performance. The LLM cascade strategy sequentially sends a query to multiple LLM APIs, returning the first reliable (cheapest) response and avoiding more expensive queries, showing how multiple inferences with LLM models are not unrealistic. As such, one could imagine our model recycling strategies to be applied to LLMs as well, either to improve model performance (focus of the current thesis) or to reduce inference costs (Chen

[4]https://www.mckinsey.com/capabilities/mckinsey-digital/our-insights/the-economic-potential-of-generative-ai-the-next-productivity-frontier#key-insights

et al., 2023). We leave this follow-up work as a potential future work.

3.4.3 Construct Validity

ApacheJIT uses the SZZ algorithm to find bug-inducing changes. The SZZ algorithm tends to label many clean commits as bug-inducing. However, Keshavarz and Nagappan (2022) use additional filtering steps that are shown to be effective in removing safe changes that are identified as bug-inducing by SZZ, and they utilized GumTreeDiff (Falleri et al., 2014) to remove trivial changes. Therefore, we believe that the bug-inducing changes in ApacheJIT are of high quality.

We use the sliding window technique to create data splits to mimic release cycles in real-world ML applications. This technique may not exactly represent every nuance of the real-world scenario, but it provides a structured and controlled way to simulate the progression of time and evolving data distributions. Despite its slight deviations from reality, we firmly believe that the sliding window technique is a valuable approximation.

3.5 Discussion

The reuse of historical models in general is not new. There have been several studies done in the past to exploit historical models to some extent for concept drift adaptation. We have listed them in Table 3.28 along with other related work. There are multiple distinctive features of our study compared to these studies, which we discuss in detail in this section.

Black-box – A major distinctive feature of our study is that we treat the historical

models as black boxes. This is important, because retraining historical models can be costly and time-consuming, and would change the known, possibly certified behaviour of an existing model. Furthermore, training data or scripts of old models might no longer be available, while the historical, black-box model could still be valuable.

Time Window – Another distinctive feature of our study is that we limit the number of historical models to be reused, which in turn limits the historical data to a specific time window. This is because, as we add models to the ensemble, it increases complexity and it is impractical to load all historical models in memory at inference. However, we also evaluated variants without limiting the number of models where filtering is applied, which not only reduces historical models but also selects models trained on diverse data distributions. We believe that filtering can help in selecting the model trained on distinct data trends which can help the ensemble generalize better on unseen data.

Filtering History – The third distinctive feature of our study is that we filter the historical models before reusing them. This is because we want to avoid reusing models that are too similar to other models in the ensemble and might reduce the ensemble's performance. This is important because if we reuse similar models, it can lead to overfitting and reduce the diversity of the ensemble. Another advantage of filtering historical models is that it reduces the number of models to be reused, which reduces the number of models to be inferenced per commit in turn reducing the inference time.

While many studies do limit the historical models using a time window, they do not filter the models. They use weighted voting, which adjusts the weights of the models based on their performance on the latest data. This is done to avoid reusing models that are not performing well on the latest data. However, this approach does not

consider the diversity of the models, which is important to avoid overfitting. This also means that all models in the ensemble are inferenced every time even if their weight in prediction is negligible, leading to higher latency.

Filtering Method – Filtering can be done in multiple ways. On the one hand, one could use statistical tests such as the Wilcoxon or ScottKnott test to cluster similar models and select models with statistically different error performances. On the other hand, one could also use clustering techniques or simply use performance metrics such as g-mean or f1-score to find the best historical model. We believe that filtering can be done in multiple ways and can be further explored in future work.

Domain – While prior research has explored historical model reuse for concept drift adaptation, our study is the first to assess its applicability in the context of software analytics models. It is important to note that the outcomes of black-box model reuse can vary across domains. For instance, in tasks like news article classification, as seen with TIX (Forman, 2006), it consistently achieved over 10 points of F-measure improvement. In contrast, experiments by Tsymbal et al. (2008) using synthetic data sets emulating abrupt and gradual concept drifts, as well as a real-world antibiotic resistance data set, revealed that dynamically integrating classifiers built over small time intervals or fixed-sized data blocks can outperform methods like majority voting and weighted voting. As such, it is important to evaluate the performance of model recycling in the domain of software analytics.

In our study, we evaluate the performance of five model recycling strategies on Just-In-Time defect prediction models. When we compare the performance of recycling strategies across projects, we see a variance in performance. This is not only a

problem with our study but also with other studies in the literature. For example, JIT-Line's (Pornprasit and Tantithamthavorn, 2021) F1-score between OpenStack and QT differs by 10%. Similarly, in the study by Kamei et al. (2016) on JIT defect prediction, within-project JIT defect prediction F1-score varied up to 47% between eleven different projects studied. This variance in performance can be due to the difference in data distributions between projects. This is also the reason why we see a variance in performance across projects in our study. However, we believe that our study is still valuable because we see a consistent improvement in performance across projects for the best recycling strategy (Model Stacking). This also holds for performance degradation variants from the worst recycling strategy (Clustering).

Furthermore, we perform the first large-scale comparative study of multiple recycling strategies. We also evaluate the trade-offs between each strategy in terms of time-to-inference and metric performance. We believe that our study can be a valuable foundation for future research and practical applications across diverse software projects and programming languages.

Table 3.28: Distinctive Features: Prior Work vs. Our Approaches

Strategy	Related	Description	Domain	UHM[1]	Black-box	FH[2]	FM[3]	TW[4]
Model Selection	**Our work**	**See Section 3.2.1**	**Software Analytics**	✓	✓	✓	**G-mean, F1-score**	✓
Model Reuse	**Our work**	**See Section 3.2.2**	**Software Analytics**	✓	✓	✓	**G-mean, F1-score**	✓
Model Stacking	TIX (Forman, 2006)	New model is trained using augmented features and with historical model output	News Article Classification	✓	✓			✓
	Our work	**See Section 3.2.3**	**Software Analytics**	✓	✓	✓	**ScottKnott, Wilcoxon**	✓

Continued on next page

Table 3.28 – continued from previous page

Strategy	Related	Description	Domain	UHM[1]	Black-box	FH[2]	FM[3]	TW[4]
Model Voting	SEA (Street and Kim, 2001)	Combines historical model output with new model output using majority voting	Salary Prediction, Cancer Detection, Web User Behaviour Prediction	✓	✓			✓
	Learn++ (Polikar et al., 2001)	Ensemble of weak learners trained on different data distributions combined using majority voting	OCR, Vehicle Classification, Gas Sensing, Synthetic (Numeric)	✓	✓			

Continued on next page

Table 3.28 – continued from previous page

Strategy	Related	Description	Domain	UHM[1]	Black-box	FH[2]	FM[3]	TW[4]
	Learn++.NSE (Elwell and Polikar, 2011)	Combines historical model output with new model output using weighted voting where weights are calculated based on models' MSE on latest and previous data	Weather Prediction, Synthetic (Numeric, Image)	✓	✓			

Continued on next page

Table 3.28 – continued from previous page

Strategy	Related	Description	Domain	UHM[1]	Black-box	FH[2]	FM[3]	TW[4]
	AUE2 (Brzezinski and Stefanowski, 2013)	Combines historical model output with new model output using weighted voting where weights are calculated based on models' MSE on latest data only	Synthetic (Numeric), Energy Price Prediction, Forest Cover Type Prediction, Poker Hand Classification, Airline Delay Prediction	✓				✓

Continued on next page

Table 3.28 – continued from previous page

Strategy	Related	Description	Domain	UHM[1]	Black-box	FH[2]	FM[3]	TW[4]
	DTEL (Sun et al., 2018)	Preserves a diverse set of historical models and retrains them on new data. Employs weighted voting scheme used by AUE2	Synthetic (Numeric), Energy Price Prediction, Forest Cover Type Prediction, Poker Hand Classification, Web User Behaviour Prediction, Tweet Classification	✓		✓	Yules Q-statistic	✓
	Our work	**See Section 3.2.4**	**Software Analytics**	✓	✓	✓	**ScottKnott, Wilcoxon**	✓

Continued on next page

Table 3.28 – continued from previous page

Strategy	Related	Description	Domain	UHM[1]	Black-box	FH[2]	FM[3]	TW[4]
Clustering	DIC (Tsymbal et al., 2008)	Combines historical models output with new model output • Dynamic Selection (DS): model with the best local performance is selected • Dynamic Voting (DV): combines models output using majority voting • DV with Selection (DVS): half of the models are selected using DS and then combined using DV	Synthetic (Numeric), Antibiotic Resistance in Patients	✓	✓			✓

Continued on next page

Table 3.28 – continued from previous page

Strategy	Related	Description	Domain	UHM[1]	Black-box	FH[2]	FM[3]	TW[4]
	two-step (Cruz et al., 2022)	A two-step approach for model selection that involves choosing the most suitable model from the model pool	Micro-milling of aerospace components					
	Our work	**See Section 3.2.5**	**Software Analytics**	✓	✓			✓

[1]UHM – Using Historical Models

[2]FH – Filtering History

[3]FM – Filtering Method

[4]TW - Time Window

3.6 Chapter Summary

In summary, this study challenges the conventional practice of discarding older Machine Learning (ML) models once newer ones are deployed. It highlights the substantial value that older models still hold compared to freshly trained RFS counterparts. This chapter introduces novel post-deployment model recycling techniques, offering a more informed approach to deciding which old models to reuse and when to do so. In other words, in traditional ML, release-readiness of a model is not a static decision made at deployment time, but a finer-grained decision that should be revisited per inference request.

Through an empirical investigation across eight long-lived Apache projects, encompassing a substantial number of commits, this study analyzes the performance of five model recycling strategies on three different types of models in the domain of Just-In-Time defect prediction, a popular subdomain of software analytics. Our findings indicate that our approach significantly outperforms the traditional RFS approach in terms of recall, g-mean, AUC and F1 performance metrics. Our best recycling strategy (Model Stacking) outperforms the baseline in over 50% of the projects across all models. These results may differ based on the choice of the project and recycling strategy.

While these recycling strategies showcase improved model performance, it is important to consider that they come at the cost of negligible to low effect size degradation in precision and increased time-to-inference compared to RFS. Nevertheless, this research underscores the potential of thoughtfully retaining and reusing older models, enhancing the effectiveness of ML models in real-world settings.

CHAPTER 4

A State-of-the-practice Release-readiness Checklist for Generative AI-based Software Products

As software products evolve to incorporate Generative AI and Large Language Models (LLMs) (Hou et al., 2023), the complexity of ensuring their release-readiness escalates, highlighting the need for revised evaluation criteria that match their advanced capabilities. This shift is underscored by the potential economic impact of generative AI, which is estimated to contribute $2.6 trillion to $4.4 trillion annually to the global economy as per McKinsey's report[1], emphasizing the importance of efficient and effective LLM integration.

However, determining when these generative AI-integrated products are ready for

[1]https://www.mckinsey.com/capabilities/mckinsey-digital/our-insights/the-economic-potential-of-generative-ai-the-next-productivity-frontier

release poses a complex challenge. Alongside their capabilities, generative AI bears the risk of generating inappropriate or misleading content, necessitating robust control measures to mitigate such risks (Achiam et al., 2023). Furthermore, the concept of release-readiness in software development is multi-faceted, involving a product's compliance with all specified requirements and successful passage through various tests (functionality, performance, security, usability, etc.). When it comes to generative AI-incorporating products, release-readiness extends to evaluations of the model's performance, ethical implications, and the potential effects of its outputs on users. Additionally, ensuring compliance with legislation, which is still evolving and varies from country to country, is essential, adding another layer of complexity to determining release-readiness.

To some extent, some of the challenges of ensuring the production-readiness of *generative* AI-based products in real-world settings are still the same as for systems leveraging *traditional* machine learning (ML) systems. The latter introduced strong dependencies between a code base and both data and models. These challenges include non-determinism, where outputs are not predictable due to the inherent variability in AI's decision-making processes, and the difficulty of testing, since there are no explicit requirements for models that can cover all potential outputs or behaviors. These problems led to the need for release checklists, i.e., formal lists of evaluation criteria that organizations should check either manually or (semi)automatically, to determine whether their product is ready for release to end users (Breck et al., 2017; ISE, 2023; Zinkevich, 2017).

However, LLMs do not only inherit the typical ML concerns around data dependency and model unpredictability, but they also face unique issues such as ensuring

contextually accurate and unbiased language understanding, managing the vast and evolving scope of human language, and addressing the ethical implications of their responses in a wider range of real-world scenarios. Companies have already begun to share their processes and experiences through blogs and industry conferences, contributing to a growing body of "grey literature" on the subject.

This thesis takes the first step towards understanding and synthesizing these disparate sources, with an eye toward future automation of many of these release-readiness evaluation steps. While existing checklists for traditional AI typically were limited to the experiences of a single organization, our systematic review of 65 grey literature sources on generative AI release-readiness considers views of 44 different organizations. In addition to identifying the major release challenges faced by the latter organizations, we derive a first release-readiness checklist aimed both at practitioners and researchers.

4.1 Methodology

Our systematic grey literature review follows a similar approach for search construction as used by earlier work (Raulamo-Jurvanen et al., 2017), and aims at capturing the breadth of discussions surrounding the deployment of LLMs in production environments. We started from a base query, i.e., "generative ai release checklist", which covers the seminal resources in the field such as the "LLMs in Production Conference" (Community, 2023) and the influential blog posts of Matt Bornstein (2023) and Huyen (2023). We then split the query into three parts (1. "generative ai", 2. "release" and 3. "checklist") and included synonyms or related words to improve the coverage of grey literature, yielding the resulting search query:

```
("generative ai" | "large language models" | llms | "foundation
models" | chatgpt | "conversational ai") AND (release | production
| deployment | monitoring | observability | operations) AND (checklist
| checks | guardrails | considerations | requirements | practices |
patterns | challenges | methods | risks)
```

The timeframe for our search spanned from June 1, 2018, coinciding with the release of GPT-1 by OpenAI, to September 1, 2023, marking the commencement of our survey.

Inclusion criteria of our search:

- Content must be authored by or associated with organizations focused on the field of generative AI.
- Content should specifically discuss LLMs in production settings.
- Accessibility to the full text without restrictions.

The exclusion criteria:

- Product announcements, advertisements, and demos.
- News articles.
- Non-English literature.
- Duplicates.
- Literature that mentions LLM production without detailing the techniques used.
- Articles behind paywalls.

Performing a Google search using our resulting query, yielded 522 million results, however, the results became irrelevant after the top 100 results. Therefore, we collect only the top 100 results for the search query. To augment the results with grey publications from leading industries in this area, an additional search was conducted by using the same search string introduced earlier but including the company name, for each of the top 10 companies recognized for their work in generative AI and LLMs, as listed by resources such as from Benaich and Capital (2023) and Gartner (2023). Across both types of queries, we collected a total of 1,100 results, and after applying our in/exclusion criteria, 35 were left. An additional 30 articles were included through snowball sampling on those 35, where references from the initial set led to further relevant grey literature. These results comprise blog posts, videos and industry papers.

I, along with my postdoc, then started developing a taxonomy of release-readiness challenges and solutions from a sample of the obtained grey literature. After independently coding 20% of the sample ($\sim$ 12), they compared and converged their taxonomies to reach a consensus. This established taxonomy was then applied to recode the initial 20%, after which both authors' codings were compared to assess agreement and resolve any conflicts. We computed Krippendorff's Kappa to evaluate the inter-rater reliability of the coding done using the taxonomy, yielding a Kappa score of 0.917, satisfying the minimum acceptable level of reliability (> 0.8) (Krippendorff, 2009). High agreement on this coding allowed the first author to complete the coding of challenges and solutions for the remaining 80% of grey sources. All results were documented in a mindmap comprising nodes representing challenges of LLM release-readiness or solution practices, along with a list of references. Following the creation of the mindmap, our industry collaborator and I organized several workshops with their

team to transform the mindmap into a checklist using the team's feedback.

Our replication package[2] includes the resulting mind map to visually organize the findings, the collected notes for each grey resource we reviewed and an online repository of references for further exploration. We refer to online references in this thesis with the notation "O«number»".

4.2 Results

4.2.1 Challenges

Figure 4.1 lists the 40 identified release challenges related to generative AI, and their prevalence in the grey sources. Note that the latter measures the number of "mentions" of these challenges in the grey sources, which is not necessarily equal to the actual frequency of challenges in practice (companies may use specific strategies or face challenges without necessarily discussing them in public media). The top five challenges discussed are:

1. **Reliability** concerns the consistency of LLMs in delivering the same result for the same input and ensuring accurate and coherent responses without making things up (hallucinations).

2. **Deployment Resource Management** deals with the optimization of computational resources necessary for LLMs in production, addressing factors like cost, latency, and overall system performance.

[2]https://github.com/SAILResearch/replication-24-harsh-generative-ai-release-readiness-checklist

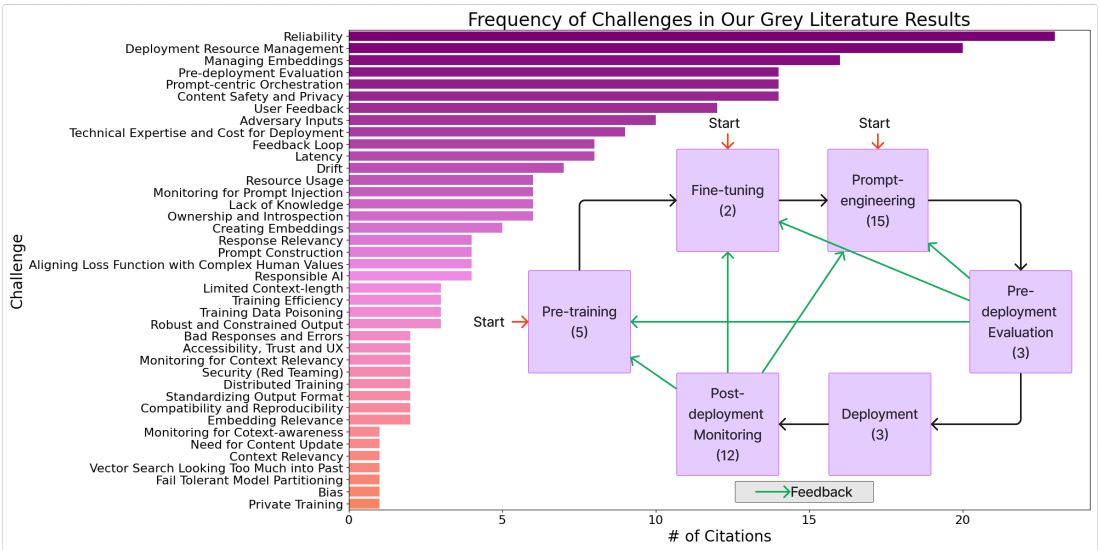


Figure 4.1: Frequency of challenges in our grey literature results (The numbers in the flowchart indicate the number of challenges associated with each stage of the LLM lifecycle, as identified through our survey)

3. **Managing Embeddings** involves creating, storing, and updating vector representations of language data to enhance the model's semantic understanding, facilitating more accurate and contextually relevant responses.

4. **Pre-deployment Evaluation** ensures that before an LLM is released, it is tested thoroughly to guarantee it is consistent and unbiased across various demographic groups, providing relevant and safe responses.

5. **Prompt-centric Orchestration** requires designing and managing prompts effectively to fully leverage LLMs' capabilities, focusing on decomposing complex tasks, integrating external tools, and avoiding vendor lock-in, while also ensuring the protection of sensitive information to maintain privacy and security.

4.2.2 Release-readiness Checklist

This section presents a release-readiness checklist for LLM-based software products, based on the challenges and practices identified from the grey literature. The checklist is organized according to the lifecycle of LLM-based software products shown in the flowchart of Figure 4.1, which is based on the product release stages identified through our survey and the number of associated challenges. For each lifecycle stage, the checklist includes a set of checkpoints that practitioners can use to evaluate the release readiness of their LLM-based software products and to identify areas for improvement. Each checkpoint in the checklist is accompanied by references to the grey literature and by a brief explanation of proven practices that should be used to address the challenges associated with the question. It is essential that for each challenge applicable to a given product, one of the proposed practices or an appropriate adaptation must be adopted.

C1 When pretraining a new LLM:

C1.1 Check if your LLM needs to be trained using sensitive/private data provided by others. [O43]

def.: This involves determining whether the training of your LLM requires access to data that is sensitive or private, belonging to third parties. This consideration is crucial for ensuring privacy compliance and ethical use of data, especially in scenarios where data sensitivity may pose risks to individuals' privacy or lead to regulatory issues.

sol.: Federated learning (FL) for LLMs (e.g., using FedLLM or Nvidia Flare) addresses privacy and data sharing concerns by sharing models instead of raw data,

adapting the traditional FL approach to the unique demands of LLMs. In this FL variant, clients train models on their private data locally. Then, through an iterative process, a central, trusted party aggregates the updated parameters from these clients multiple times ensuring that clients benefit from collective training insights without compromising their data privacy. After several iterations of this process, a final model is obtained that incorporates the collective insights of all participating clients.

C1.2 Partition the training process across multiple accelerators (i.e., GPUs or Devices) to speed up training. [O28] [O42]
sol.: Distributed platforms for LLM workloads, such as Ray, play a crucial role in enabling parallelization, particularly when incorporating advanced techniques like FL.

C1.3 Employ anonymization techniques. [O43]

- ***def.:*** Anonymization in LLM training is the process of altering or removing identifiable information from data sets to protect individual privacy. It focuses on making it impossible to associate data with specific individuals, differing from federated learning, which avoids sharing raw data but does not necessarily anonymize it.
- ***sol.:*** Tools like Gretel.ai, Private AI, and Tonic.ai automate the process of removing sensitive data and generating synthetic alternatives. However, for optimal privacy protection, this automation should be complemented with manual checks to catch any nuances automation might miss, ensuring thorough data privacy.

C1.4 Consider potential ways to deal with training data poisoning. [O26] [O27] [O29]

- ***def.:*** LLMs, such as T5 and GPT-3, rely on extensive datasets from sources like Common Crawl, WebText, OpenWebText (covering news and Wikipedia), and books — which alone constitute 16% of GPT-3's training material — to learn and generate content. However, these LLMs are vulnerable to "training data poisoning", a tactic where attackers deliberately introduce misleading information into these training datasets. As a result, the LLMs may inadvertently learn and replicate these inaccuracies or biases in their outputs, thus undermining the models' reliability and integrity. Recognizing this risk, the OWASP List advises caution regarding over-dependence on AI-generated content, highlighting the importance of maintaining high standards for data quality and model training processes to safeguard against such vulnerabilities.
- ***sol.:*** Check the source of external training data used by your organization, and keep records of its origins, similar to the methods used in the Software Bill of Materials (SBOM). Please see the SPDX specs[3] for details about emerging AI and data SBOM scenarios.

C2 When fine-tuning an LLM:

C2.1 Minimize unsafe behaviours or align loss function to complex human values. [O2] [O25] [O29] [O58]

[3]https://fossa.com/blog/spdx-3-0/,2023

- ***sol.:*** Reinforcement Learning with Human Feedback (RLHF) merges instruction fine-tuning with reinforcement learning, using human preferences (like pairwise comparisons) to train a reward model. This technique is crucial for aligning AI behavior with human values and safety, as it directly incorporates human judgments into training. By doing so, RLHF aims to reduce the occurrence of unsafe behaviors, ensuring that AI actions are ethical, appropriate, and aligned with human expectations, thus enhancing the safety and reliability of AI applications.

C2.2 Utilize efficient fine-tuning processes to improve training efficiency. [O2] [O6] [O9]

- ***def.:*** With models growing in size, full fine-tuning is impractical on prosumer hardware, and storing and deploying separate fine-tuned models for each task is costly, as they are as large as the original model.
- ***sol.:*** Parameter Efficient Fine-Tuning (PEFT) methods fine-tune only a few additional parameters, freezing most of the pre-trained LLM's parameters, significantly reducing computational and storage costs, and addressing catastrophic forgetting seen in full fine-tuning of LLMs.

C3 When doing prompt engineering:

C3.1 Consider using different prompt construction methods. [O4] [O7] [O10] [O47]

sol.: Employing techniques like Zero-shot, Few-shot, and Chain-of-Thought (CoT) prompting at the start can significantly affect effectiveness. While simple prompts can yield good results, their success often depends on the level of detail and

craftsmanship involved. A well-crafted prompt should include clear instructions or questions and may also incorporate context, inputs, or examples for more precise guidance, leading to better outcomes.

C3.2 Utilize known practices to improve reliability and hallucinations. [O1] [O2] [O3] [O4] [O7] [O9] [O13] [O14] [O18] [O19] [O22] [O24] [O26] [O34] [O36] [O38] [O40] [O44] [O46] [O47] [O57] [O60] [O63]

- ***def.:*** Users expect consistency, like receiving the same insurance quote on each visit to a website. However, LLMs, being stochastic, may not always give the same output for identical inputs.
- ***sol.:*** Model-based evaluation / self-evaluation – Conduct real-time evaluation of outputs being returned (as done by OpenAI Evals). It can include simple yet impactful checks, such as asking "Are you sure?" to prompt a final review or adjustment of the output. This mechanism enables the LLM to critically assess its own responses, ensuring they are consistent, reliable, and ethically sound, while drawing on methods that allow for such thorough evaluation.
- ***sol.:*** Utilizing self-consistency and employing multiple prompting (Wang et al., 2022) – Create multiple responses for the same input, selecting the final output through a majority vote or by having the LLM choose the best one. This can be done using popular LLM APIs like OpenAI's.
- ***sol.:*** Guardrails – Guardrails ensure LLM outputs are not only coherent but also syntactically accurate, factual, and free from harmful content, while also protecting against adversarial inputs. Examples include Guardrails.ai,

NeMo, Guidance, and rellm.

C3.3 Bridge knowledge gaps by enriching prompts with relevant context. [O38] [O47] [O55] [O56] [O63]

- ***def.:*** LLMs lack awareness of context beyond their training data, often covering up knowledge gaps with assumptions, termed hallucinations.
- ***sol.:*** Retrieval-Augmented Generation (RAG) enhances inputs by fetching relevant external data, offering richer context and reducing hallucinations by grounding models in factual content. Moreover, updating retrieval indices is more cost-effective than continuous fine-tuning of an LLM.

C3.4 Utilize vector stores and embedding models to store, search and update vector representations of language data. [O2] [O5] [O7] [O9] [O10] [O11] [O14] [O17] [O33] [O36] [O38] [O44] [O55] [O60] [O61] [O63]

- ***def.:*** Vector stores and embedding models are essential for managing and querying vector representations of language data, enabling efficient retrieval of relevant information (see C3.3) and reducing hallucinations.
- ***sol.:*** Vector stores like Weaviate, Pinecone, and pgvector, and embedding models by OpenAI, Huggingface and Cohere, are crucial for creating, storing, and updating vector representations of language data, facilitating more accurate and contextually relevant responses.

C3.5 Employ efficient context-retrieval techniques to overcome LLM context length limit. [O1] [O4] [O38]

- ***def.:*** The largest GPT-4 model processes up to 50 pages of input, with its performance, both in terms of inference time and accuracy, significantly declining near this context window size.
- ***sol.:*** Serve relevant chunks – To boost RAG (see C3.3) efficiency, only the most relevant documents are selected and sent for each LLM prompt, utilizing a method called chunking. This technique divides large texts into smaller, manageable pieces, enhancing the LLM's information retrieval accuracy and effectiveness by focusing on the most pertinent text segments.
- ***sol.:*** Contextual compression – Condense and outline essential facts for addressing the question, doubling or tripling information density.

C3.6 If your application is used by untrusted users, you need an approach to deal with adversarial input. [O2] [O4] [O18] [O19] [O34] [O26] [O29] [O40] [O54] [O57]

- ***def.:*** Prompt injection seeks to manipulate model behavior by using crafty prompts to alter outputs. Prompt Injection Vulnerabilities arise when attackers craft inputs to exploit a trusted large language model (LLM) across various channels, often remaining undetected due to the inherent trust in the LLM's output.
- ***sol.:*** Defence in Instruction – A basic defensive strategy is to guide the model's behavior through clear instructions in the prompt. While not foolproof, this underscores the impact of effectively crafted prompts.
- ***sol.:*** Guardrails (see C3.2)
- ***sol.:*** Parameterized Prompt Components – Prompt injections, similar to SQL injections, imply that defense techniques from SQL could be adapted.

One approach is parameterizing prompt components, treating instructions and inputs separately, inspired by SQL's methods.

C3.7 Ensure prompts' compatibility across changes to the underlying LLM (e.g., migrating from GPT3.5 to GPT4) and reproducibility across changes to the prompt itself. [O1] [O19]

- ***def.:*** In prompt engineering, unlike traditional software engineering where third-party library updates aim to ensure backward and/or forward compatibility after updates, there is no guarantee that prompts for an older model will work with a new version, often necessitating prompt rewrites. Therefore, if model changes are anticipated, it is vital to unit-test all prompts with evaluation examples.
- ***sol.:*** Temperature – Guarantees consistent output for identical input with a given model at the expense of less creativity.
- ***sol.:*** Prompt Versioning – Minor prompt modifications can significantly alter results, making it crucial to version and monitor each prompt's performance.
- ***sol.:*** Prompt templates – Prompt templates offer a scalable, reproducible method for generating prompts, encompassing instructions, few-shot examples, or various action sequences for language models.

C3.8 Incorporate practices to ensure content safety and privacy. [O1] [O2] [O4] [O5] [O9] [O18] [O24] [O26] [O27] [O38] [O40] [O57] [O60]

- ***def.:*** This check ensures safety and privacy in content generated by third-party LLMs, focusing on issues that can persist even after RLHF (see C2.1)

fine-tuning. It involves proactive prompt review and adjustment to prevent the generation of harmful or sensitive information, addressing inherent model concerns and ensuring outputs remain within ethical and privacy guidelines.

- ***sol.:*** Models for detecting profanity, such as the PyPI module "profanity-check".
- ***sol.:*** LLM-based evaluation – Employ LLMs to assess responses for inappropriate content.
- ***sol.:*** Use Personal Identifiable Information (PII) masking solutions with LangChain or LLamaIndex.
- ***sol.:*** Utilize PII detection tools, such as Microsoft/presidio or Azure services.

C3.9 Consider reusing prompts, decomposing a large task into smaller tasks, utilizing external tools or chaining prompts. [O7] [O10] [O11] [O17] [O36] [O38] [O40] [O44] [O55] [O61] [O62]

- ***def.:*** Prompt orchestration refers to the strategic combination of techniques to enhance the effectiveness of LLMs. This includes reusing successful prompts for similar tasks, breaking down complex problems into smaller, more manageable tasks, integrating external tools to extend capabilities, and chaining prompts in a sequence to tackle multi-step challenges.

- ***sol.:*** Agent and chain orchestration frameworks like LangChain and LlamaIndex excel in simplifying prompt chaining, handling external API interactions, managing contextual data retrieval from vector stores, and preserving memory across multiple LLM calls.
- ***sol.:*** Additionally, these frameworks leverage thought decomposition frameworks such as CoT, Tree of Thought, or ReAct to break down complex tasks into more manageable sub-tasks. This method not only simplifies the problem-solving process but also significantly enhances the effectiveness of the LLM in handling intricate tasks by guiding it through a structured approach to reasoning and response generation.

C3.10 Ensure responses are highly constrained using types, templates and constraints. [O61] [O62] [O63]

sol.: When integrating LLMs into larger systems with non-LLM components, ensuring the comprehensibility and usability of the generated text is essential. Constrained decoding reduces the risk of prompt injection attacks by enforcing input constraints, enhancing overall security (LMQL). Compared to guardrails, constrained decoding aims to restrict LLM outputs at the token level, ensuring that each step of the model's output is valid and controlled.

C4 Pre-deployment evaluation ensuring that a model's performance matches safety criteria for deployment.

C4.1 Evaluate the performance of your application pre-deployment. [O2] [O5] [O6] [O9] [O12] [O15] [O17] [O19] [O35] [O36] [O38] [O55] [O56] [O58]

- ***def.:*** Assessments involve measurements to gauge a model's task performance, including benchmark data and metrics. Evaluations help measure system or product performance and identify regressions. Representative evaluations are crucial for monitoring system changes at scale, eliminating the need for manual inspection of LLM outputs with each update.
- ***sol.:*** A/B Testing for Models and Prompts – Organizations often use A/B testing to determine response quality when evaluating interactions with generative AI models.
- ***sol.:*** Similarity metrics – When using LLMs for classification tasks (e.g., toxicity detection, document categorization) or extractive QA without dialogue, leverage standard classification metrics like recall, precision, PRAUC, etc. For tasks without correct answers but references (e.g., machine translation, extractive summarization), use reference metrics based on matching (BLEU, ROUGE) or semantic similarity (BERTScore, MoverScore).
- ***sol.:*** Human evaluation – Traditional machine learning metrics like precision, recall, and F1 score may not completely capture language nuances. Evaluations must be approached with care, considering various aspects. Human evaluation, involving human assessors to judge LLM outputs, offers valuable insights into response quality.
- ***sol.:*** Penetration Tests and Red Teaming use adversarial tactics to test cybersecurity, recognizing the potential for control failures and shadow IT. Red Teaming in the context of LLMs involves ethical hackers identifying and exploiting vulnerabilities specific to LLM systems. For example, they might test an LLM's resilience to generating biased or inappropriate content. It

is crucial to include LLM-specific test cases in security practices like pen tests, code reviews, and threat modeling. Anthropic's red-teaming dataset provides a practical resource for integrating these adversarial simulations, enhancing defenses against real-world threats[4].

- ***sol.:*** Task-specific benchmarks – For complex tasks like abstractive summarization or dialogue, traditional metrics like BLEU and ROUGE fall short, and broad academic benchmarks such as MMLU may not fully align with specific application needs. Eval Driven Development (EDD) requires the creation of custom benchmarks that directly reflect the unique requirements of each task. For example, in abstractive summarization, rather than relying on generic accuracy metrics, EDD suggests benchmarks focused on measuring the coherence and relevance of generated summaries to the original text. This method ensures evaluations are directly tied to the specific challenges of the task at hand. Automated evaluation with a strong LLM enhances this process, making the feedback loop more efficient.

- ***sol.:*** Leveraging LLMs for evaluation, such as the G-Eval framework, incorporates Chain-of-Thought (CoT)reasoning to assess LLM outputs. CoT is a technique where the model explains its reasoning process step by step, leading to a final judgment, enhancing transparency and interpretability. This method begins with providing the LLM task-specific instructions and criteria. For instance, in evaluating news summarization, the process involves

[4]https://github.com/anthropics/hh-rlhf

concatenating the prompt, the CoT (which outlines the evaluative reasoning), the news article, and its summary. The LLM then evaluates these elements for coherence and relevance, assigning a score, such as on a 1 to 5 scale.

C4.2 Consider practices to ensure accessibility and to build end-user trust in your application. [O2] [O42]

sol.: For comprehensive guidance on ensuring human-centric AI interaction, refer to the AI UI/UX guidelines provided by industry leaders such as Microsoft[5], Google[6] and Apple[7]

C5 During deployment:

C5.1 Check if you need a commercial or open-source LLM deployment. [O9] [O19] [O26] [O27] [O29] [O33] [O36] [O37] [O41]

- ***def.:*** Choosing between commercial and open-source LLMs mirrors decisions for traditional software libraries but with potentially larger impacts due to the complexities of deploying AI models. This choice hinges on balancing the need for deployment ease, maintenance support, and cost against the desire for flexibility, control, and the opportunity to innovate.
- ***sol.:*** Commercial LLMs offer streamlined deployment and maintenance, reducing the need for extensive technical expertise. They provide reliable

[5]https://www.microsoft.com/en-us/research/publication/guidelines-for-human-ai-interaction/
[6]https://pair.withgoogle.com/guidebook/
[7]https://developer.apple.com/design/human-interface-guidelines/machine-learning

support, simplify compliance and integration issues, and focus organizational resources on application development rather than infrastructure management. The trade-off typically includes higher licensing costs and less flexibility but less operational overhead.

- ***sol.:*** Open-source LLMs demand more from the deploying organization in terms of technical skills and infrastructure but offer significant advantages in flexibility and customization. They allow for adaptation to specific needs and encourage innovation, with the added benefit of potential cost savings on licensing fees. The open-source community can be a valuable resource for troubleshooting and development, making this option attractive for organizations willing to invest in the necessary expertise.

C5.2 Use optimization techniques to meet latency, cost and resource requirements. [O2] [O3] [O4] [O5] [O9] [O17] [O19] [O20] [O24] [O27] [O28] [O32] [O33] [O36] [O38] [O41] [O47] [O60] [O63] [O64]

- ***def.:*** It is important to establish cost requirements when considering 3rd party LLMs since their pricing continuously evolves and can go up quickly, especially with token length variations. Additionally, you can choose between cloud-based or on-premise deployment. Cloud offers scalability, while on-premise is favored for enhanced data security.
- ***sol.:*** Semantic caching – Caching can greatly cut response latency for previously served responses. It also reduces the need for redundant LLM requests, saving costs, yet the catch is to find a good similarity algorithm (and data retention mechanism).

- ***sol.:*** Methods like model compression, quantization, pruning, and distillation are used for model size reduction. Use them to decrease memory usage, improve computational efficiency, and lower latency.
- ***sol.:*** GPUs / TPUs / CPUs – Select hardware that matches your requirements for processing power, memory, and storage capacity.
- ***sol.:*** Utilize smaller, specialized LLMs that offer similar performance – Despite common perception, larger does not necessarily equate to better in the realm of LLMs. Smaller, task-specific models can be equally or more efficient. They tend to be cost-effective, demand fewer computing resources, and suit resource-limited projects.
- ***sol.:*** Memory optimization and distributed inference – To efficiently manage LLMs' high memory requirements, strategies like data, tensor (model sharding), pipeline, and hybrid parallelism are crucial. Data parallelism involves duplicating the model across GPUs to handle multiple requests simultaneously. Tensor parallelism, or model sharding, divides the model's layers horizontally across GPUs, distributing the computational load. Pipeline parallelism segments the model vertically, with each GPU processing different layers sequentially. Hybrid parallelism combines these techniques, optimizing performance and scalability for effective deployment.
- ***sol.:*** Attention layer optimization – Techniques like FlashAttention and PagedAttention optimize attention memory requirements, reducing data movement and improving latency and throughput.
- ***sol.:*** Scheduling optimization – Efficient request scheduling on model servers

is crucial due to the high, yet variable latency of LLMs during inference, ensuring a good user experience with LLM applications.

C6 Post-deployment monitoring to set up feedback loops and improve application performance.

C6.1 Monitor for prompt injection attacks. [O7] [O16] [O18] [O26] [O38] [O40]

- ***def.:*** Detecting prompt injection involves identifying adversarial prompts before they manipulate the LLM's responses. This proactive approach aims to flag and assess suspicious inputs, preventing harmful or unintended interactions.
- ***sol.:*** Utilize an Adversarial Prompt Detector, such as "rebuff." – While LLMs excel in diverse tasks, they can also be fine-tuned for specific roles, such as knowledge generation (Liu et al., 2021) and self-verification (Weng et al., 2022). Similarly, an LLM can serve to identify and filter out adversarial prompts.
- ***sol.:*** Similarity with known attacking prompts – This method involves gauging text similarity between prompts and responses, comparing them to known instances of jailbreak attempts, prompt injections, and LLM service refusals.

C6.2 Monitor for resource usage. [O3] [O9] [O26] [O32] [O35] [O63]

- ***def.:*** After deployment, continuously assess the model's performance, resource usage, and cost efficiency to optimize operations.
- ***sol.:*** LLM Token Count and Token Utilization – LLMs, like GPT models, utilize tokens for input and output, which can vary from one character to a

whole word. Efficient token usage monitoring is vital for cost control and staying within the model's token limit.

- ***sol.:*** Resource Usage – Regularly track CPU, GPU, memory usage, latency, and throughput as part of standard monitoring.

C6.3 If your application is publicly exposed, consider solutions to prevent DDoS attacks. [O3] [O8] [O25]

- ***sol.:*** API Rate Limiting and Captcha – Implement rate limiting to ensure a positive user experience while deterring misuse.

C6.4 Monitor for drift. [O1] [O10] [O16] [O35] [O38] [O42] [O43]

- ***def.:*** "Drift" refers to the decline in a model's performance due to changes in data distribution over time. To promptly identify any shifts in user behavior or environmental changes, it is essential to monitor a model's inputs and outputs continuously.
- ***sol.:*** Comparing Model Performance: Monitor the model's current performance against historical data to detect shifts in data distribution. Measuring drift involves tracking changes in data over time by comparing current and historical model performance.
- ***sol.:*** Expected prompts vs. Actual prompts – Prompt Variability assesses discrepancies between anticipated and actual user prompts, highlighting deviations in usage that may affect system performance. This comparison reveals how the system's real-world performance might diverge from initial expectations, emphasizing the need for continuous adaptation and refinement based on actual user interactions to improve effectiveness.

- ***sol.:*** Concept Drift Detection: Observe if there are changes in the model's response patterns.
- ***sol.:*** Clustering prompts and responses using embeddings (see C3.4) – Identify problematic responses by clustering prompts and responses through embeddings. Analyze cluster-specific metrics, drift, or user feedback to pinpoint and address potential issues.

C6.5 Check if your context-retrieval system can serve the "relevant" documents. [O14] [O38]

- ***def.:*** A challenge occurs when a user's query poorly aligns with the data in the vector store. In these cases, the system, struggling to find an exact match, resorts to using the closest available data, which may not be completely relevant. This issue is often triggered by unique or very specific queries.
- ***sol.:*** Query density – Query density measures the extent to which the vector store encompasses user queries. Significant drift in query density indicates that the vector store might not be adequately representing the range of user queries, leading to a lack of closely related data.
- ***sol.:*** Ranking metrics – These metrics assess the effectiveness of the search and retrieval system in choosing the most pertinent data. A decrease in ranking metrics suggests the system may require improvements in differentiating between relevant and irrelevant data.

C6.6 Evaluate the context served to the LLM. [O58]

- ***def.:*** A key necessity for production-level LLM applications is ensuring they do not fabricate facts, and return responses that are well-grounded in the given context. "Context" here refers to the combination of the prompt, any additional information given to guide the LLM's understanding, and the background knowledge it draws upon to construct responses. It is important to monitor this aspect diligently because inaccuracies or fabrications can significantly undermine the credibility and usefulness of the LLM.
- ***sol.:*** Use other LLM – Employ a secondary LLM to evaluate the context's relevance.
- ***sol.:*** Context alignment and factual accuracy – This involves critically evaluating how the LLM retrieves, interprets and applies context in its responses. The goal is to verify that the LLM correctly understands the topic, checks the accuracy of the information it references or provides, and incorporates this information in a way that is both correct and relevant. Ensuring the LLM's responses are not only factually accurate but also appropriately contextualized is key to maintaining its reliability and effectiveness.

C6.7 Check LLM responses for relevance. [O16] [O35] [O56] [O58]

- ***def.:*** Is the tone of your LLM's responses appropriate? Are there sudden or gradual changes in the sentiment of your prompts? Are the topics diverging from what you expected?
- ***sol.:*** Is your LLM providing relevant content and sticking to the expected topics for this application?
- ***sol.:*** Sentiment score – Monitor for any sudden or gradual changes in the

sentiment of the LLM's responses, checking if they stray from the expected topics. This evaluation ensures the LLM's outputs remain aligned with the intended tone and relevance, providing consistent and appropriate interactions.

- ***sol.:*** Classification into topics (politics, etc) – Ensure that the responses consistently align with the application's designated topics.
- ***sol.:*** Similarity between Q&A – Measure the alignment between queries and LLM responses to ensure they accurately address the query's intent, indicating the LLM's comprehension and relevance in its answers. High semantic similarity reflects the LLM's effective understanding and precision.

C6.8 Continuously evaluate the need for content updates for the context-retrieval system. [O38]

- ***def.:*** Inability of the LLM to answer some queries may signal a need to update the vector store (see C3.4).
- ***sol.:*** # Refusals – Count the LLM's generic non-answer responses to develop a metric.

C6.9 Ensure fairness. [O35] [O40] [O42] [O43]

- ***def.:*** LLMs can inherit and propagate biases from their training data. Organizations need to track and measure these biases in outputs using fairness metrics, which may vary by domain, including gender, race, or other unintentional biases.
- ***sol.:*** Sentiment score (see C6.7)

- ***sol.:*** Assess the model's performance across various demographic groups, e.g., Gender Bias can be measured by comparing the model's performance across different genders.
- ***sol.:*** Toxicity analysis – Alongside sentiment analysis (see C6.7), employ toxicity classifiers to detect and address harmful or inappropriate content in LLM outputs, ensuring outputs are not only unbiased but also safe and respectful across all user interactions.

C6.10 Monitor for latency. [O10] [O11] [O32] [O35] [O36] [O42] [O57] [O63]

- ***def.:*** Latency is a significant concern in LLMOps, significantly influenced by the length of outputs generated by LLMs. This issue is more pronounced in LLMOps compared to MLOps, affecting both development speed and user experience in production.
- ***sol.:*** Prompt causing high latency – Utilize observability solutions to analyze prompt and response pairs through structured metadata, like API latency, enabling the identification of high-latency instances for targeted optimization.

C6.11 Monitor for bad responses. [O35] [O40]

- ***def.:*** LLMs can occasionally generate unwanted responses.
- ***sol.:*** Implicit and explicit user feedback – User feedback is invaluable for monitoring; negative or confused reactions should be taken as significant indicators.

4.3 Threats to Validity

4.3.1 External Validity

Our checklist might not fully capture all the nuances of generative AI readiness for release. The field of generative AI is advancing rapidly since the time our survey was performed, meaning our guidelines may miss some aspects. However, the fundamental challenges and solutions still hold. This checklist should be viewed as a foundation upon which practitioners can build, adapt and expand to align with the evolving landscape of generative AI technology and its applications.

4.3.2 Internal Validity

Our checklist was created by a team of human coders, which may introduce bias. We attempted to mitigate this by using an established empirical methodology.

4.4 Chapter Summary

This chapter investigates the release readiness of software products integrated with generative AI, synthesizing a comprehensive checklist to guide practitioners in evaluating their LLM products' readiness for release. Unlike traditional ML checklists, which focus on performance metrics and data quality, the checklist for generative AI also emphasizes ethical considerations, the impact of generated content, and the dynamic nature of generative models. This distinction addresses the unique challenges of ensuring that generative AI products not only perform as intended but also align with broader societal and ethical standards. As the generative AI landscape rapidly evolves,

this checklist underscores the need for ongoing adaptation and community engagement to ensure the responsible and effective use of LLMs in software development.

CHAPTER 5

Conclusions and Future Work

This thesis has explored innovative approaches to enhancing the release-readiness process of machine learning models, both in the realm of historical ML model reuse and generative AI-based software products. It underscores the critical role of release-readiness in ensuring that software systems, utilizing both traditional ML and generative AI, are prepared for deployment. Establishing rigorous release-readiness criteria is crucial for identifying and mitigating potential risks, ensuring software meets both technical performance and compliance standards. Existing release-readiness checklists, which support comprehensive evaluations through manual or semi-automatic checks, are foundational to a systematic release process.

However, this thesis identifies two critical areas for improvement: first, the checklists' inherent bias towards newer ML models, disregarding the value of older models, and second, the emerging need to update these checklists to effectively tackle the unique challenges introduced by generative AI technologies, such as hallucinations and prompt injections. The thesis challenges that the traditional lifecycle management practices for ML models, characterized by regular retraining and the subsequent discarding of older models, overlook the value of old models. Moreover, it claims that the advent of generative AI-based software products introduces distinct challenges that necessitate revising current ML-based software release-readiness checklists to specifically address generative AI release-readiness. This revision should also incorporate a broad perspective, drawing on the experiences of various organizations.

Based on the results, this thesis has successfully challenged a core assumption made by traditional lifecycle management practices of ML models, highlighting the overlooked value of historical models. Through empirical analysis and rigorous evaluation, we have demonstrated that the conventional approach of routinely discarding older models in favor of periodic retraining does not fully leverage the potential insights these models offer, especially in JIT defect prediction scenarios. Our findings advocate for a paradigm shift towards recycling models, thereby enhancing the predictive accuracy and operational efficiency of ML-based software systems.

Moreover, this work has underscored the necessity of revising and expanding existing release-readiness checklists to address the nuanced challenges posed by the integration of generative AI technologies into software products. By incorporating a broader set of considerations, including those derived from the experiences of multiple organizations, this thesis contributes a more comprehensive framework for assessing the

release-readiness of generative AI-based applications. This not only ensures a higher standard of reliability and performance but also fosters a more ethically and socially responsible deployment of AI technologies.

Ultimately, the contributions of this thesis lay the groundwork for future research and practice in ML model lifecycle management and the deployment of AI-based software. By advocating for the strategic reuse of historical models and the adoption of inclusive, dynamically updated release-readiness checklists, this work invites a reconsideration of how we evaluate and deploy AI technologies.

5.1 Thesis Contributions

The main contributions of this thesis are listed below:

- Through rigorous empirical analysis across several large and long-lived open-source projects, this study has substantiated the latent value of historical JIT defect prediction models, challenging the prevailing norm of discarding older models in favour of new training iterations. It highlights the need for inference-time release-readiness decision in traditional ML instead of a static, one-time decision.

- This research has conceptualized and introduced novel strategies for the recycling of machine learning models post-deployment, showcasing their potential in enhancing the efficiency and effectiveness of model deployment processes.

- A thorough evaluation of the proposed recycling strategies against traditional retraining methods provides critical insights into their effectiveness, offering a new perspective on model deployment strategies in software analytics.

- By analyzing the trade-offs between inference latency and performance metrics, this research offers a nuanced understanding of the practical implications of adopting recycling strategies, guiding practitioners in making informed decisions.

- In response to the unique challenges presented by generative AI, this thesis pioneers a specialized checklist that emphasizes the nuanced requirements for the successful deployment of generative AI-based applications, marking a significant advancement in the field.

- By drawing insights from multiple organizations, the release-readiness checklist not only addresses the technical and ethical challenges of releasing generative AI-based software products but also fosters a collaborative approach to improving AI software deployment practices.

5.2 Future Work

Potential directions for future research include:

- Additional research could explore more granular aspects of recycling strategies, such as the impact of project-specific characteristics on the effectiveness of these strategies, and the potential for their application in different domains beyond software analytics.

- There is an opportunity to further refine and expand the release-readiness checklist to cover emerging challenges and opportunities in the deployment of generative AI, incorporating insights from a broader range of stakeholders and industry

contexts.

- Developing automated tools and methodologies for the implementation and verification of checklist activities, aiming to streamline the assessment process for release-readiness, particularly in fast-paced development environments.
- Extending the concept of model recycling to generative AI models to examine their potential in improving the deployment process and exploring the unique challenges associated with recycling these types of AI models.
- Evaluating the long-term impact of adopting model recycling and dynamic release-readiness assessments on software project outcomes, including product quality, team productivity, and maintenance costs.
- Conducting case studies with industry partners to validate and refine the proposed recycling strategies and release-readiness checklist in real-world settings, facilitating the transfer of academic insights into practical applications.

Bibliography

Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F. L., Almeida, D., Altenschmidt, J., Altman, S., Anadkat, S., et al. (2023). Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Benaich, N. and Capital, A. S. (2023). State of AI Report 2023 — stateof.ai. https://www.stateof.ai/. [Accessed 14-03-2024].

Breck, E., Cai, S., Nielsen, E., Salib, M., and Sculley, D. (2017). The ml test score: A rubric for ml production readiness and technical debt reduction. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 1123–1132. IEEE.

Brzezinski, D. and Stefanowski, J. (2013). Reacting to different types of concept drift: The accuracy updated ensemble algorithm. *IEEE transactions on neural networks and learning systems*, 25(1):81–94.

Cabral, G. G. and Minku, L. L. (2022). Towards reliable online just-in-time software defect prediction. *IEEE Transactions on Software Engineering*, 49(3):1342–1358.

Cabral, G. G., Minku, L. L., Shihab, E., and Mujahid, S. (2019). Class imbalance evolution and verification latency in just-in-time software defect prediction. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 666–676. IEEE.

Chawla, N. V., Bowyer, K. W., Hall, L. O., and Kegelmeyer, W. P. (2002). Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357.

Chen, L., Zaharia, M., and Zou, J. (2023). Frugalgpt: How to use large language models while reducing cost and improving performance. *arXiv preprint arXiv:2305.05176*.

Chen, Z. and Liu, B. (2018). *Lifelong machine learning*, volume 1. Springer.

Community, M. (2023). LLMs in Production Conference - Event | MLOps Community — home.mlops.community. https://home.mlops.community/public/events/llms-in-production-conference-2023-04-13. [Accessed 14-03-2024].

Cruz, Y. J., Rivas, M., Quiza, R., Haber, R. E., Castaño, F., and Villalonga, A. (2022). A two-step machine learning approach for dynamic model selection: A case study on a micro milling process. *Computers in Industry*, 143:103764.

De Lange, M., Aljundi, R., Masana, M., Parisot, S., Jia, X., Leonardis, A., Slabaugh, G., and Tuytelaars, T. (2021). A continual learning survey: Defying forgetting in classification tasks. *IEEE transactions on pattern analysis and machine intelligence*, 44(7):3366–3385.

Diethe, T., Borchert, T., Thereska, E., Balle, B., and Lawrence, N. (2019). Continual learning in practice. *arXiv preprint arXiv:1903.05202*.

Ekanayake, J., Tappolet, J., Gall, H. C., and Bernstein, A. (2009). Tracking concept drift of software projects using defect prediction quality. In *2009 6th IEEE International Working Conference on Mining Software Repositories*, pages 51–60. IEEE.

Elwell, R. and Polikar, R. (2011). Incremental learning of concept drift in nonstationary environments. *IEEE Transactions on Neural Networks*, 22(10):1517–1531.

Falessi, D., Ahluwalia, A., and Penta, M. D. (2021). The impact of dormant defects on defect prediction: A study of 19 apache projects. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(1):1–26.

Falleri, J.-R., Morandat, F., Blanc, X., Martinez, M., and Monperrus, M. (2014). Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 313–324.

Forman, G. (2006). Tackling concept drift by temporal inductive transfer. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 252–259.

Gao, S., Zhang, H., Gao, C., and Wang, C. (2023). Keeping pace with ever-increasing data: Towards continual learning of code intelligence models. *arXiv preprint arXiv:2302.03482*.

Gartner (2023). Gartner generative ai: What is it, tools, models, applications and use cases. https://www.gartner.com/en/topics/generative-ai. [Accessed 14-03-2024].

Goncalves, V. P., Silva, L. P., Nunes, F. L., Ferreira, J. E., and Araújo, L. V. (2024). Concept drift adaptation in video surveillance: A systematic review. *Multimedia Tools and Applications*, 83(4):9997–10037.

Herraiz, I., Rodriguez, D., Robles, G., and Gonzalez-Barahona, J. M. (2013). The evolution of the laws of software evolution: A discussion based on a systematic literature review. *ACM Computing Surveys (CSUR)*, 46(2):1–28.

Hess, M. R. and Kromrey, J. D. (2004). Robust confidence intervals for effect sizes: A comparative study of cohen'sd and cliff's delta under non-normality and heterogeneous variances. In *annual meeting of the American Educational Research Association*, volume 1. Citeseer.

Hoang, T., Dam, H. K., Kamei, Y., Lo, D., and Ubayashi, N. (2019). Deepjit: an end-to-end deep learning framework for just-in-time defect prediction. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 34–45. IEEE.

Hou, X., Zhao, Y., Liu, Y., Yang, Z., Wang, K., Li, L., Luo, X., Lo, D., Grundy, J., and Wang, H. (2023). Large language models for software engineering: a systematic literature review (2023). *arXiv preprint arXiv:2308.10620*.

Huyen, C. (2023). Building LLM applications for production — huyenchip.com. https://huyenchip.com/2023/04/11/llm-engineering.html. [Accessed 14-03-2024].

ISE, M. (2023). ML model production checklist - Engineering Fundamentals Playbook — microsoft.github.io. https://microsoft.github.io/

code-with-engineering-playbook/machine-learning/ml-model-checklist/. [Accessed 14-03-2024].

James, G., Witten, D., Hastie, T., Tibshirani, R., et al. (2013). *An introduction to statistical learning*, volume 112. Springer.

Jiang, T., Tan, L., and Kim, S. (2013). Personalized defect prediction. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 279–289. Ieee.

Kamei, Y., Fukushima, T., McIntosh, S., Yamashita, K., Ubayashi, N., and Hassan, A. E. (2016). Studying just-in-time defect prediction using cross-project models. *Empirical Software Engineering*, 21:2072–2106.

Kamei, Y., Shihab, E., Adams, B., Hassan, A. E., Mockus, A., Sinha, A., and Ubayashi, N. (2012). A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 39(6):757–773.

Keshavarz, H. and Nagappan, M. (2022). Apachejit: a large dataset for just-in-time defect prediction. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 191–195.

Kim, S., Whitehead, E. J., and Zhang, Y. (2008). Classifying software changes: Clean or buggy? *IEEE Transactions on software engineering*, 34(2):181–196.

Kononenko, O., Baysal, O., Guerrouj, L., Cao, Y., and Godfrey, M. W. (2015). Investigating code review quality: Do people and participation matter? In *2015 IEEE international conference on software maintenance and evolution (ICSME)*, pages 111–120. IEEE.

Krippendorff, K. (2009). Testing the reliability of content analysis data. *The content analysis reader*, pages 350–357.

Kubat, M., Matwin, S., et al. (1997). Addressing the curse of imbalanced training sets: one-sided selection. In *Icml*, volume 97, page 179. Citeseer.

Lemaître, G., Nogueira, F., and Aridas, C. K. (2017). Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in machine learning. *Journal of Machine Learning Research*, 18(17):1–5.

Liu, J., Liu, A., Lu, X., Welleck, S., West, P., Bras, R. L., Choi, Y., and Hajishirzi, H. (2021). Generated knowledge prompting for commonsense reasoning. *arXiv preprint arXiv:2110.08387*.

Lu, J., Liu, A., Dong, F., Gu, F., Gama, J., and Zhang, G. (2018). Learning under concept drift: A review. *IEEE transactions on knowledge and data engineering*, 31(12):2346–2363.

Lundberg, S. M. and Lee, S.-I. (2017). A unified approach to interpreting model predictions. *Advances in neural information processing systems*, 30.

Macbeth, G., Razumiejczyk, E., and Ledesma, R. D. (2011). Cliff's delta calculator: A non-parametric effect size program for two groups of observations. *Universitas Psychologica*, 10(2):545–555.

Matt Bornstein, R. R. (2023). Emerging Architectures for LLM Applications | Andreessen Horowitz — a16z.com. https://a16z.com/emerging-architectures-for-llm-applications/. [Accessed 14-03-2024].

McIntosh, S. and Kamei, Y. (2018). Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction. In *Proceedings of the 40th International Conference on Software Engineering*, pages 560–560.

Mockus, A. and Weiss, D. M. (2000). Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2):169–180.

Olewicki, D., Habchi, S., Nayrolles, M., Faramarzi, M., Chandar, S., and Adams, B. (2023). Towards lifelong learning for software analytics models: Empirical study on brown build and risk prediction. *arXiv preprint arXiv:2305.09824*.

Olewicki, D., Nayrolles, M., and Adams, B. (2022). Towards language-independent brown build detection. In *Proceedings of the 44th International Conference on Software Engineering*, pages 2177–2188.

Paleyes, A., Urma, R.-G., and Lawrence, N. D. (2022). Challenges in deploying machine learning: a survey of case studies. *ACM Computing Surveys*, 55(6):1–29.

Patel, H. (2023). Post-deployment model recycling. https://github.com/SAILResearch/replication-23-harsh-model_recycling.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

Polikar, R., Upda, L., Upda, S. S., and Honavar, V. (2001). Learn++: An incremental learning algorithm for supervised neural networks. *IEEE transactions on systems, man, and cybernetics, part C (applications and reviews)*, 31(4):497–508.

Pornprasit, C. and Tantithamthavorn, C. K. (2021). Jitline: A simpler, better, faster, finer-grained just-in-time defect prediction. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 369–379. IEEE.

Quinonero-Candela, J., Sugiyama, M., Schwaighofer, A., and Lawrence, N. D. (2008). *Dataset shift in machine learning*. Mit Press.

Rajbahadur, G. K., Wang, S., Oliva, G. A., Kamei, Y., and Hassan, A. E. (2021). The impact of feature importance methods on the interpretation of defect classifiers. *IEEE Transactions on Software Engineering*, 48(7):2245–2261.

Raschka, S. (2018). Mlxtend: Providing machine learning and data science utilities and extensions to python's scientific computing stack. *Journal of open source software*, 3(24):638.

Raulamo-Jurvanen, P., Mäntylä, M., and Garousi, V. (2017). Choosing the right test automation tool: a grey literature review of practitioner sources. In *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*, pages 21–30.

Schelter, S., Biessmann, F., Januschowski, T., Salinas, D., Seufert, S., and Szarvas, G. (2015). On challenges in machine learning model management.

Singhal, S., Chawla, U., and Shorey, R. (2020). Machine learning & concept drift based approach for malicious website detection. In *2020 International Conference on COMmunication Systems & NETworkS (COMSNETS)*, pages 582–585. IEEE.

Song, L., Minku, L., and Yao, X. (2023). On the validity of retrospective predictive performance evaluation procedures in just-in-time software defect prediction. *Empirical Software Engineering*.

Street, W. N. and Kim, Y. (2001). A streaming ensemble algorithm (sea) for large-scale classification. In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 377–382.

Strubell, E., Ganesh, A., and McCallum, A. (2019). Energy and policy considerations for deep learning in nlp. *arXiv preprint arXiv:1906.02243*.

Sun, Y., Tang, K., Zhu, Z., and Yao, X. (2018). Concept drift adaptation by exploiting historical knowledge. *IEEE transactions on neural networks and learning systems*, 29(10):4822–4832.

Tabassum, S., Minku, L. L., and Feng, D. (2022). Cross-project online just-in-time software defect prediction. *IEEE Transactions on Software Engineering*, 49(1):268–287.

Tan, M., Tan, L., Dara, S., and Mayeux, C. (2015). Online defect prediction for imbalanced data. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 99–108. IEEE.

Tantithamthavorn, C., McIntosh, S., Hassan, A. E., and Matsumoto, K. (2018). The impact of automated parameter optimization on defect prediction models. *IEEE Transactions on Software Engineering*, 45(7):683–711.

Tsymbal, A., Pechenizkiy, M., Cunningham, P., and Puuronen, S. (2008). Dynamic integration of classifiers for handling concept drift. *Information fusion*, 9(1):56–68.

Wang, X., Wei, J., Schuurmans, D., Le, Q., Chi, E., Narang, S., Chowdhery, A., and Zhou, D. (2022). Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*.

Weisstein, E. W. (2004). Bonferroni correction. *https://mathworld. wolfram. com/*.

Weng, Y., Zhu, M., He, S., Liu, K., and Zhao, J. (2022). Large language models are reasoners with self-verification. *arXiv preprint arXiv:2212.09561*.

Wolpert, D. H. (1992). Stacked generalization. *Neural networks*, 5(2):241–259.

Woolson, R. F. (2007). Wilcoxon signed-rank test. *Wiley encyclopedia of clinical trials*, pages 1–3.

Zeng, Z., Zhang, Y., Zhang, H., and Zhang, L. (2021). Deep just-in-time defect prediction: how far are we? In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 427–438.

Zhao, Y., Damevski, K., and Chen, H. (2023). A systematic survey of just-in-time software defect prediction. *ACM Computing Surveys*, 55(10):1–35.

Zinkevich, M. (2017). Rules of machine learning: Best practices for ml engineering. *URL: https://developers. google. com/machine-learning/guides/rules-of-ml*.