Towards Sustainable AI for Continuous Integration Quality Gates

By Doriane Olewicki

A thesis submitted to the Graduate Program in School of Computing in conformity with the requirements for the Degree of Doctor of Philosophy

Queen's University
Kingston, Ontario, Canada
January, 2025

Copyright © Doriane Olewicki, 2025

Abstract

Continuous Integration (CI) is the concept of continuously incorporating new code changes in the project's main codebase. To ensure the quality of the integrated changes, two types of quality gates are typically considered by most software development companies: automated gates, leveraging the build process, which invokes compilation commands and runs tests, and manual gates, based on peer reviews.

With projects scaling up in size, the cost required for maintaining CI quality gates can be considerable (in terms of computational/manual effort). To assist or improve CI quality gates, the Software Engineering (SE) community has proposed a wide range of software analytics models. However, most of prior work focuses on the initial deployment of such models, without considering the different factors involved in their long-term sustainability: automated gates require smart retraining processes to maintain performance, while manual gates require synergy with reviewers' workflows.

In this thesis, we study sustainable AI for CI quality gates. For automated gates, we explore how to improve the learning process of software analytics models using (1) dynamic training scheduling strategies, where we evaluate different ways to dynamically adapt the training time for Retraining-From-Scratch (RFS) setups, and (2) Lifelong Learning (LL) training setups, where we change the fundamental learning algorithms toward an incremental setup. For the manual CI quality gates, we perform

user studies to evaluate the impact on reviewers' productivity of (3) hotspot-based file-ordering, where problematic changes are shown first to the reviewers, and of (4) reviewers' interaction with generated review comments.

We were able to optimize the trade-off between model performance and the computational effort of model retraining, with our best heuristics recommending retraining only once every 5-6 weeks without losing performance to weekly retrained models. Yet, LL substantially reduces computational effort even further by 2-40x compared to RFS. Through large-scale industrial user studies, we also observe that reviewers benefit from ML-based assistance, as file-reordering led to more (+23%) and better (precision +13%, recall +8%) review comments compared to the default alphanumeric file-ordering, while generative AI-based review comment generation obtains promising results regarding acceptance (8.1% and 7.2%) and appreciation (23% and 28.3%).

Co-authorship

For all thesis chapters and the publications they are based on, my contribution comprises designing and refining the research plan, conducting background literature reviews and related work, collecting datasets, performing and leading data analysis and experiments, and finally writing the associated manuscripts.

The first three papers were written in collaboration with Ubisoft, with me as the main researcher on the associated projects. The fourth paper was a collaborative effort with Ubisoft and Mozilla. I designed the ideas behind "RevMate", which was prototyped and developed in collaboration with Mozilla with me in charge of integration of the prototype in Ubisoft's review environment. Furthermore, I designed and coordinated the overall user study (across both organizations) and the analysis of the results.

The papers related to the work presented in this thesis are listed below:

1. Doriane Olewicki, Mathieu Nayrolles, and Bram Adams. 2022. *Towards Language-independent Brown Build Detection.* IEEE/ACM International Conference on Software Engineering (ICSE) [1]. Accepted. Regarding this paper, only RQ3 is addressed in this thesis.

2. Doriane Olewicki, Sarra Habchi, Mathieu Nayrolles, Mojtaba Faramarzi, Sarath Chandar, and Brams Adams. 2024. *On the Costs and Benefits of Adopting*

Lifelong Learning for Software Analytics - Empirical Study on Brown Build and Risk Prediction. IEEE/ACM International Conference on Software Engineering - Software Engineering in Practice (ICSE - SEIP) [2]. Accepted and awarded with distinguished award.

3. Doriane Olewicki, Sarra Habchi, and Bram Adams. 2024. *An Empirical Study on Code Review Activity Prediction and Its Impact in Practice.* ACM International Conference on the Foundations of Software Engineering (FSE) [3]. Accepted.

4. Doriane Olewicki, Leuson Da Silva, Suhaib Mujahid, Arezou Amini, Benjamin Mah, Marco Castelluccio, Sarra Habchi, Foutse Khomh, and Bram Adams. 2024. *Impact of LLM-based Review Comment Generation in Practice: A Mixed Open-/Closed-source User Study.* IEEE Transactions on Software Engineering (TSE). Submitted, under review.

Acknowledgments

First, I want to thank my advisor, Professor Bram Adams. I've had the chance to work with Bram since 2018 when I started my Master's. Bram has always been an amazing mentor and an inspiration regarding work, but also the way he shows respect to everyone he works with. I'm forever grateful for everything I'm continuously learning from him and I will continue applying his ways throughout my career.

Thanks to Bram, I also got the chance to work with Ubisoft La Forge. There are so many people I got the chance to work or exchange with throughout this collaboration. The list could be pages long, but I need to give a special thanks to my mentors Sarra Habchi and Mathieu Nayrolles, who have been supervising and encouraging my research, to Arezou Amini, who has contributed in many ways, and to Andrea Feder, who has given me support and opportunities to represent students at Ubisoft.

To my mother and father, my sister and sister-in-law, I'm so grateful for your support and love, thanks for being there for me. Thanks to my support system: my whole family and my friends from Belgium, Montreal, Ubisoft, Queen's, Polytechnique and Waterloo. You have all contributed to those incredible years. Timothée, my forever +1, you know I could not have done this without you. We did it. I hope you enjoyed learning so much about CI, reviews and LLMs. Thank you so much.

Finally, thanks to my jury for taking the time to read and evaluate my thesis.

Statement Of Originality

The following works are my own and I hereby certify the intellectual content of this thesis is the product of my own work. All references and contributions of other individuals has been cited and sourced appropriately, as defined by the IEEE Citation Reference manual.

Contents

Abstract i

Co-authorship iii

Acknowledgments v

Statement Of Originality vi

Table of Contents vii

List of Tables xiv

List of Figures xviii

Chapter 1: Introduction 1

1.1 Context 1

1.1.1 Automated CI Quality Gates 2

1.1.2 Manual CI Quality Gates 3

1.1.3 Sustainable CI Quality Gates 4

1.2 Hypothesis 5

1.2.1 Part 1: Towards sustainable training of ML-based approaches in automated CI quality gates 7

1.2.2 Part 2: Towards sustainable adoption of advanced ML models in code reviewers' workflows . . . 9
1.3 Main findings . . . 11
1.3.1 Study 1: Dynamic re-training scheduling strategies for brown build software analytics models . . . 11
1.3.2 Study 2: Lifelong learning setup for software analytics models 11
1.3.3 Study 3: Hot-spot prediction-based file ordering for code review 12
1.3.4 Study 4: Code review comment suggestion . . . 13

Chapter 2: Related work **14**
2.1 Importance of Continuous Integration for Software development . . . 14
2.2 Software Analytics Models for CI quality gates . . . 15
2.2.1 Software Analytics Models for automated CI gates . . . 15
2.2.2 Software Analytics Models for manual CI gates . . . 17
2.3 Background on Machine Learning . . . 21
2.3.1 Concept drift . . . 21
2.3.2 Online Learning for software analytics . . . 22
2.3.3 Lifelong Learning in the ML community . . . 24
2.3.4 LLMs and Prompt Engineering . . . 26

Chapter 3: STUDY 1: Dynamic re-training scheduling strategies - Concept Drift detection for Brown Build Prediction **28**
3.1 Introduction . . . 28
3.2 Methodology . . . 31
3.2.1 Recurrent retraining setup . . . 31

3.2.2 Training dataset reduction approach 32
3.2.3 Feature survival ratio analysis 33
3.2.4 Model switching heuristics 34
3.2.5 Use case: Brown Build detection 37
3.2.6 Design of Empirical Evaluation 39
3.3 Baselines 40
3.4 RQ1: What is the most relevant training set for Retraining-From-Scratch of brown build detection models at a given training time? . . 41
3.5 RQ2: How is the continuity between models impacted by concept drift? 43
3.6 RQ3: What is the most effective retraining schedule for Retraining-From-Scratch of brown build detection models? 44
3.7 Threats to Validity 46
3.8 Conclusion 47
3.9 Future work 48

Chapter 4: STUDY 2: Lifelong learning setup for software analytics models 50

4.1 Introduction 50
4.2 Lifelong Learning for Software analytics 53
4.2.1 Multi-layer perceptron neural networks 53
4.2.2 Data splitting 54
4.2.3 Lifelong learning training process 55
4.3 Empirical study methodology 58
4.3.1 Use cases 58
4.3.2 Baselines 62

4.4 RQ1: How does an LL setup perform compared to an RFS setup for software classification tasks? . . . 64
4.4.1 Motivation . . . 64
4.4.2 Approach . . . 64
4.4.3 Results . . . 65
4.5 RQ2: How essential is the use of a memory-based LL setup? . . . 68
4.5.1 Motivation . . . 68
4.5.2 Approach . . . 68
4.5.3 Results . . . 69
4.6 RQ3: How time-efficient is an LL setup compared to an RFS setup? . . . 69
4.6.1 Motivation . . . 69
4.6.2 Approach . . . 70
4.6.3 Results . . . 71
4.7 Discussion . . . 74
4.8 Threats to Validity . . . 77
4.9 Conclusion . . . 79
4.10 Future work . . . 80

Chapter 5: STUDY 3: Hot-spot prediction-based file ordering for code review 82

5.1 Introduction . . . 82
5.2 Motivational Use Case . . . 86
5.2.1 User study set-up . . . 87
5.2.2 Results . . . 89
5.3 Methodology for Empirical Study of Hot-Spot Prediction . . . 92

5.3.1 Labeling . . . 93
5.3.2 Datasets . . . 94
5.3.3 Data splitting . . . 96
5.3.4 Embeddings . . . 97
5.3.5 Classification . . . 101
5.3.6 Evaluation . . . 102
5.3.7 Statistical tests . . . 103
5.4 RQ1: Which text embeddings of code diffs perform best for review activity prediction? . . . 104
5.4.1 Motivation . . . 104
5.4.2 Approach . . . 105
5.4.3 Results . . . 105
5.5 RQ2: Which review process features perform best for review activity prediction? . . . 107
5.5.1 Motivation . . . 107
5.5.2 Approach . . . 108
5.5.3 Results . . . 108
5.6 RQ3: How does the combination of review process features with text embeddings perform for review activity prediction? . . . 111
5.6.1 Motivation . . . 111
5.6.2 Approach . . . 112
5.6.3 Results . . . 112
5.7 Discussion . . . 114
5.7.1 Hot-spot detection for file re-ordering . . . 114

5.7.2 Computation time . . . 116
5.8 Threats to Validity . . . 117
5.9 Conclusion . . . 119
5.10 Future work . . . 120

Chapter 6: STUDY 4: Impact of code review comment generation 121

6.1 Introduction . . . 121
6.2 User Study Design . . . 125
6.2.1 RevMate's LLM-based approach . . . 125
6.2.2 UI for RevMate's generated comments . . . 132
6.2.3 User study setup . . . 134
6.2.4 Data Collection . . . 136
6.2.5 Metrics and statistical tests . . . 136
6.3 RQ1: How frequently do reviewers accept comments generated by RevMate? . . . 137
6.3.1 Approach . . . 137
6.3.2 Results . . . 138
6.4 RQ2: Do acceptance ratios depend on the type of generated review comment? . . . 141
6.4.1 Approach . . . 141
6.4.2 Results . . . 144
6.5 RQ3: How does the adoption of LLM-generated comments impact the code review workflow? . . . 145
6.5.1 Approach . . . 145
6.5.2 Results . . . 147

6.6 RQ4: How does the adoption of LLM-generated comments impact the review outcome? . . . 148
6.6.1 Approach . . . 148
6.6.2 Results . . . 150
6.7 Discussion . . . 151
6.8 Threats to Validity . . . 152
6.9 Conclusion . . . 154
6.10 Future work . . . 155

Chapter 7: Conclusion **157**
7.1 Sustainable training of software analytics models for CI . . . 158
7.1.1 Main findings . . . 159
7.1.2 Future work . . . 160
7.2 Sustainable synergy with human processes . . . 160
7.2.1 Main findings . . . 161
7.2.2 Future work . . . 162
7.3 Final thoughts . . . 163

Bibliography **164**

Appendix A: Research Ethics Approval **192**

List of Tables

3.1 Information on the full history of the studied projects for brown build detection. 39

3.2 Information on the scraped part of the studied projects for brown build detection. 39

3.3 Intuitive baselines (with BFR the brown failure ratio by project). . 40

3.4 Comparison of heuristics with a win_{train} size of 30. The table is split vertically to fit two projects per row, then each section is split vertically again into a posteriori heuristics, default heuristics (first and weekly), a priori heuristics and baselines. Bold indicates the line with the best F1-score of a project per category. 45

4.1 Studied projects for Brown build and JIT risk prediction. The Brown failure rate represents the percentage of brown jobs among the failed build jobs and the risky rate the percentage of risky commits. 60

4.2 Features used for Just-In-Time risk prediction. 62

4.3 Training setup for LL and baselines. 63

4.4 Performance after hyper-parameter optimization for the LL setups and baselines. We report the average test results out of 5 runs. The **bold blue** values indicate the best F1-score (F1) and G-mean (GM) per project. For each project, Kruskal-Wallis tests ($\alpha = .05$) between the F1-score/G-mean of runs of the LL setup and of the other setups are performed. If significant, the Cliff's delta effect size is shown as "+" (large effect), otherwise "/" is shown. 65

4.5 Optimal LL hyperparameter values used in our study. 66

4.6 Median life expectation of models in #days. 70

4.7 Size of training sets for model training/updates in terms of *GS* for all datasets, and average model computation time across 5 runs for the Risk datasets (training with 50 epochs and 10 sample batches). . . . 72

4.8 %Features with higher variance of importance for RFS or LL compared to the other for the Risk use case. The variance is computed on the feature importance value before discretization (first 3 rows) and after (last 3 rows). 77

5.1 User study's patch run-through analysis: depending on the ordering approach, we report the number of patches reviewed, the median (Med.) and Average (Ave.) duration of the review, and the median and average ranking position of the files' initial opening during the reviews. 90

5.2 User study's review comment analysis: depending on the ordering approach, we report the total number of comments, the comments' position (files F1 to F5 or general Gen), the number of comments that found hot-spots, and the number of hot-spot files that were commented on. 90
5.3 Label descriptions (file-level). 94
5.4 The revision activity in the studied datasets. 95
5.5 Description of review process features generated at the file-level granularity. <LAB> refers to the different labels we consider. 100
5.6 Results for RQ1: text embeddings for the code review activity prediction task at file-level. The results shown are returned for the best model ("mod") across each variation of text embedding. Blue cells indicate the maximum values for each metric per dataset and per label. 106
5.7 Results for RQ2: software review process features embeddings on the code review activity prediction tasks at file-level. The results are returned for the best model ("mod") for each setting of text embedding. Blue cells indicate the maximum values for each metric per dataset and per label. 109
5.8 Results for RQ3: combining the text embeddings with software meta-features embeddings for the code review activity prediction task at file-level. The results are returned for the best model ("mod") for each setting of text embedding. Blue cells indicate the maximum values for each metric per dataset and per label. 115

5.9 Ordering based on hot-spot prediction. We report the "Alph" (alphabetic) ordering, and "Ours" our ordering, based on hot-spot prediction. "%patch" reports the percentage of patches with the range of file changes. The blue cells indicate the best results by dataset and by metric. 116

6.1 Prompts 126

6.2 End of study survey questions 136

6.3 RQ1 - Evaluation counts. "Total" refers to the union of all the generated comments, disregarding the approach used. 138

6.4 RQ1- Evaluation ratios. "Total" refers to the union of all the generated comments, disregarding the approach used. 138

6.5 RQ2 - For each automatically generated comment category, we list the number of comments from each Organization (Moz and Ubi) that were labeled as such. "*" indicates that the label was an example in the few-shot prompt [4]. 142

6.6 RQ2 - Categorization of generated comment and associated appreciation and acceptance. 144

6.7 RQ3 - Editing of accepted comments 147

6.8 RQ4 - Impact of generated comments on finished patch reviews. "Generated" refers to accepted generated comment. "Human" refers to manually written comments on patches without generated comments. 150

List of Figures

1.1 Overview of Thesis. The blue regions indicate the main code integration process. The yellow regions indicate the Machine Learning use cases that we considered in this thesis. The green regions indicate the contributions of this thesis. 6

3.1 Switching heuristics. We suppose that the current Week is I and the model selected used at $I-1$ is m_j. Switch heuristics use the colored zones as criteria for model selection. 33

3.2 Optimization of the drifting window size, windows varying from 5 to 50 (step of 5). The darker the line, the larger the window size is. . . . 41

3.3 Features surviving ratio per project with a win_{train} size of 30. 43

4.1 LL setup with validation set, initial training set and replay buffer of size $VWin = ITWin = RBWin = 3$, resp. 56

4.2 Relative coefficient $coef_{rel}$ of training setups compared to the LL setup updates. 73

4.3 Discretized feature importance evolution for Risk_1. 76

5.1 Data split for open- and closed-source datasets. 96

5.2 Embedding approach. The file change is composed of lines added, lines removed and meta-information about the submission (e.g., timestamp, author and reviewer ids, patch id). The combination will concatenate the targeted embeddings, i.e., the text embedding, features, or both. The final embedding for each file is 1D. 97

6.1 UI for generated comments' evaluation boxes 133

6.2 UI for choice of reasons for ignoring generated comments 133

6.3 RQ3 - Evaluation duration. Outliers are hidden for visibility. 146

Chapter 1

Introduction

1.1 Context

Continuous Integration (CI) is the concept of continuously incorporating new code and other changes back in the project's main version repository [5]. To ensure the quality of the integrated changes, CI requires reliable and swift quality gates to validate developers' changes and provide feedback such that they can update their changes and resubmit until code acceptance [6]. Two main quality gates types are typically considered by most software development companies: automated gates leveraging the build process [7], which invokes compilation commands and runs tests, and manual gates based on peer reviews of the submitted changes [8].

With companies and projects scaling up in size, the effort and cost required for maintaining CI quality gates can be considerable [9]. Build processes are costly due to execution and hosting costs. Furthermore, in medium-to-large organizations, delays tend to pop up between code submission and build outcome delivery due to queued-up builds in the CI pipelines caused by the high volume of patches continuously being submitted [10, 11]. Regarding code reviews, reviewers have the difficult task

of understanding and evaluating code changes to validate their quality and prevent introducing faults to the codebase [12]. This is a tedious process where the effort needed is highly dependent on the code submitted, as well as the author's and the reviewer's experience and availability, leading, for instance, as reported by Hong et al., to median wait times for review feedback of 15-64 hours in three open source projects (i.e., Openstack Nova, Openstack Ironic and Qt Base) [13].

1.1.1 Automated CI Quality Gates

To improve **automated CI quality gates** over the years, the Software Engineering (SE) community has proposed a wide range of Machine Learning (ML)-based approaches [14]. The community has addressed tasks such as defect prediction [15–17], bug assignment [18], build outcome prediction [10, 11, 19], test selection and prioritization [20]. Such ML approaches in the context of SE are often referred to as software analytics models [21].

Unfortunately, software analytics models tend to be evaluated as traditional ML tasks, i.e., a static model is trained from scratch then used for future predictions, ignoring the challenges specific to keeping up these models' performance throughout the software lifecycle, which intrinsically is an evolving environment. At a minimum, software analytics approaches need to be evaluated with train/test set respecting the temporality of changes (i.e., training set needs to be anterior to test set) to avoid data contamination, which is now the state-of-practice in most papers [22, 23]. To truly understand and improve the sustainability of ML-based approaches for automated quality gates, it is also essential to consider the phenomena of concept drift and performance decay due to the evolving nature of SE activities, especially in long-lived

systems [24].

The state of the practice for sustainable software analytics models consists of Retraining-From-Scratch (RFS) new models every fixed time-frame. Yet, RFS is sub-optimal, as the optimal retraining time-frame should dynamically adapt to a project's evolution, implying that one should vary the retraining time based on the data and model performance evolution instead of retraining only at scheduled times. Furthermore, retraining a new model from scratch is costly in terms of resources and loses the continuity between models, as the model previously used is discarded. MLOps should thus limit as much as possible the amount of retraining while considering the trade-off with performance, model discontinuities and computational effort.

Emerging approaches to maintain software analytics models, referred to as Online and Lifelong Learning [25], can be used to sustain performance over time while reducing the computational effort. Approaches applied to the SE domain include updatable parameters (e.g., probabilities, NLP pre-trained weights, clustering) [22, 26] and model bagging approaches [23]. Lifelong learning in particular uses incremental models, which can be incrementally updated instead of discarded, thus reducing the computation time of models' updates and reducing model discontinuity [27]. However, no prior work has addressed the question of evaluating the trade-off between performance and cost of maintaining such an LL approach compared to tradition RFS learning setups for software analytics.

1.1.2 Manual CI Quality Gates

To improve **manual CI gates**, i.e., peer reviews, in the last 10 years, a variety of software analytics models have been proposed, most of them using more traditional

ML classification algorithms, for instance, for reviewer recommendation [28–30], code change ordering [31], patch quality prediction [13, 32], and review comment evaluation [33–35]. Only recently, more advanced generative AI-based approaches have been considered in the SE domain, either as vector embeddings to represent the data used by the above traditional ML models, or as ML approaches for new tasks such as review comment generation [12, 32, 36–38] and code refinement (i.e., automatic bug fix) [12, 32, 36].

As manual CI gates largely involve human effort, it is essential to thoroughly evaluate the impact of ML approaches on the humans involved and to compare the models' predictions to what humans would have done [39], in order to validate the potential adoption and long-term usage of such approaches by the human users. Prior work primarily has evaluated LLMs in the context of review comment generation tasks by textually comparing their generated output to a ground truth [12, 32, 40]. Such comparison is usually done with similarity metrics such as BLEU score and Exact Match, which are often considered as imperfect, since they do not value rephrasing or alternate options [36]. Thus, to properly evaluate the impact of those tasks, user studies are required, yet, currently, these are lacking in the literature because of their high cost.

1.1.3 Sustainable CI Quality Gates

In this thesis, we consider the overall **sustainability of software analytics models** introduced into CI quality gates, both in terms of the way the models are maintained over time (automated quality gates) and how the models are integrated into the CI workflow (manual quality gates). This work was done in collaboration

with our industrial partner, Ubisoft Montréal. A particularity of the game development industry is that the evolving aspect of the CI process is more exacerbated, as games are developed over 3-5 years with clear development periods and intermediate objectives. Similar to typical software development processes, game development leverages software analytics models in the context of CI quality gates, and would benefit from sustainability of such approaches in terms of (1) having models updated in an optimal way in terms of both performance and computational effort, and (2) having models integrated seamlessly in human workflows.

1.2 Hypothesis

Thesis statement: *Sustainable usage of ML techniques for continuous integration quality gates requires (1) intelligent decisions with respect to model retraining and (2) synergy with human reviewers' workflows.*

To address this statement, Part 1 of this thesis focuses on the challenges of maintaining software analytics models for automated CI quality gates, with long-term usage. Then, Part 2 focuses on evaluating the potential impact of LLM-based assistants for reviewers in the context of manual quality gates. An overview of the thesis contribution can be found in Figure 1.1.

Overall, we empirically evaluated the sustainability of ML approaches in the context of CI gates, both in terms of ensuring that up-to-date models are being leveraged for automated CI gates, as well as in terms of understanding the impact on human productivity for manual CI gates. All empirical evaluations involved a mixture of open-source and industry contexts, comprising quantitative case studies on large datasets covering up to 9 years of development data, as well as large-scale user

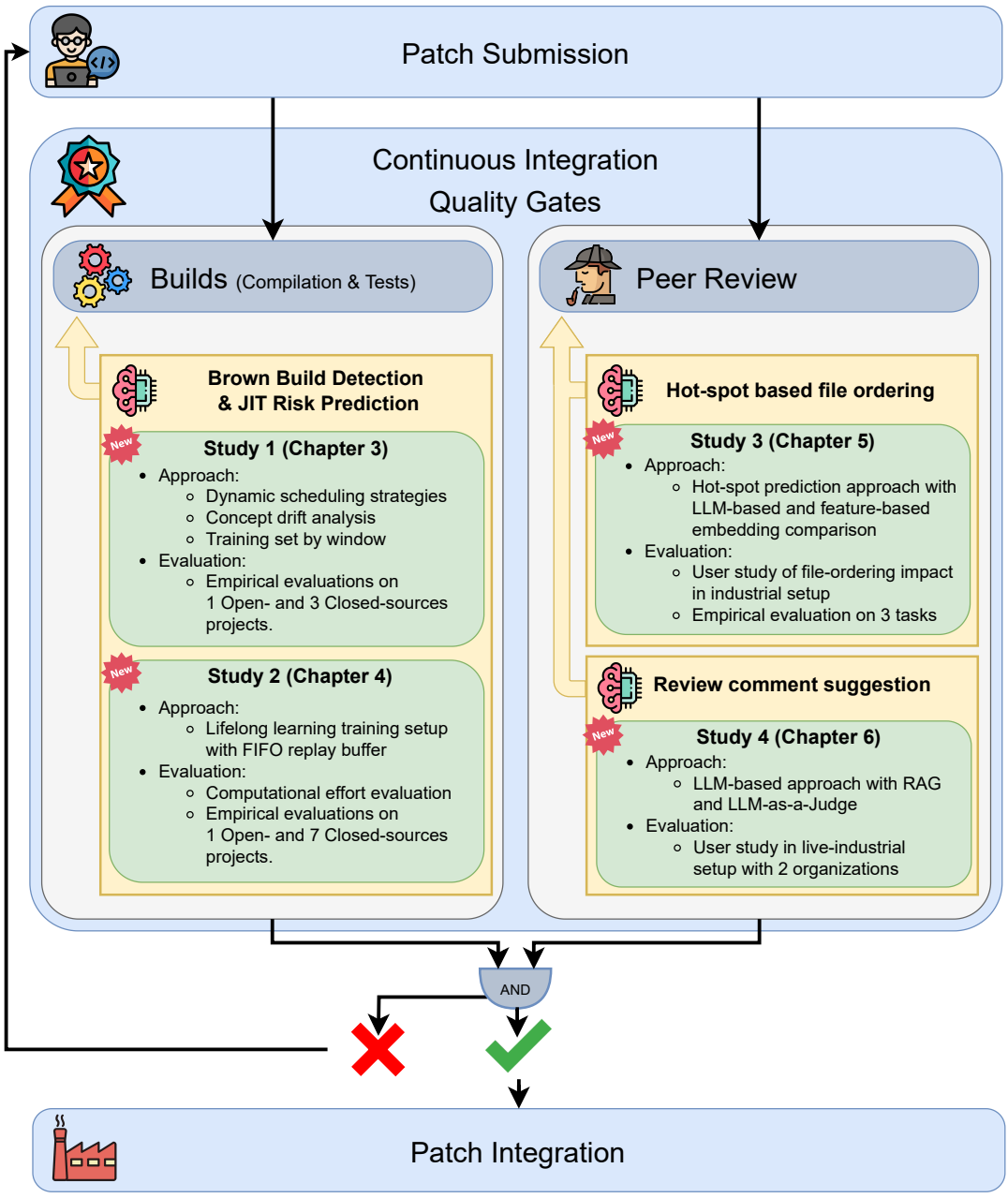


Figure 1.1: Overview of Thesis. The blue regions indicate the main code integration process. The yellow regions indicate the Machine Learning use cases that we considered in this thesis. The green regions indicate the contributions of this thesis.

studies involving 88 expert reviewers across two studies at Ubisoft and Mozilla.

1.2.1 Part 1: Towards sustainable training of ML-based approaches in automated CI quality gates

In the context of automated CI quality gate models, we explore different approaches to reduce the computational effort of model (re-)training, both in terms of frequency and training duration, while maintaining, if not improving, performance.

The thesis starts with **(STUDY 1) dynamic retraining scheduling strategies** where we determine the optimal training time for RFS, to minimize performance decay. In the extreme case, each individual prediction could require a new model to be re-trained, with the latest data included. However, this is not realistic, since training a model takes time and would increase the prediction time. If many predictions need to be made in a day, which definitely is the case in the context of CI [41], this could jam the project's pipeline.

Hence, usually, a model is used for several predictions, but this brings a new problem: how long does the model stay relevant? This corresponds to determining the model's TTL (time-to-live). The longer the TTL, the lower the frequency of model retraining, but the higher the risk of outdated predictions.

A related issue is that we also need to identify how long CI data stays relevant before training a new model version and when old data should be discarded from training sets. Data can get outdated because of the project changing over time. Also, some periods and events during the development might not be relevant to keep in the dataset, depending on the current situation. Finally, depending on the complexity of the learning algorithm, large training sets require more resources for the training

process.

Hence, in Study 1, we propose a methodology to identify when a model in a RFS setup should be retrained and which part of the data should be considered for the training set, to optimize the model prediction performance. We empirically evaluate this methodology on a brown build detection prototype, applied to four datasets, to understand factors impacting TTL. Furthermore, we explore the continuity between models by analyzing the differences in model versions in terms of feature survival ratios, as the selected features differ over time from one trained model to another.

While determining the TTL of models and of data would allow MLOps engineers to automatically identify when to retrain the model and with what data, such a process still involves RFS and might still be expensive to maintain. With **(STUDY 2) lifelong learning setups**, we change the fundamental learning setup of the automated quality gate models towards a model learning algorithm that can incrementally update a trained model with new data points, while taking care not to forget important knowledge from prior training sets, instead of using the more traditional RFS strategy.

Lifelong Learning (LL) is the domain studying how the performance of a model evolves across time while new data points/tasks are added to the training over time [27]. Lifelong learning performs incremental retraining when the model needs to be updated, which reduces the computation time of model updating and maintains the continuity of the model between updates [27]. To avoid so-called "catastrophic forgetting" of important older data points, we adopt a replay buffer of older data, which still allows us to drastically reduce the size of the overall training dataset, and hence the model training time.

The large-scale empirical evaluations in part 1 consider two software analytics

use cases, i.e., brown/flaky build prediction (for both studies) and Just-In-Time risk prediction (for the second study). For the first use case, we study four projects, among which one open-source and three closed-source ones from our industrial partner, Ubisoft, covering between 17 and 43 months of data and representing from 22k to 47k build jobs. In the context of the second use case, we study four closed-source projects, representing between 49k and 82k commits over a span of 4-9 years.

1.2.2 Part 2: Towards sustainable adoption of advanced ML models in code reviewers' workflows

The code review process comes with many challenges: understanding code, navigating the code, finding problematic changes and properly communicating with the patch's author to ensure the needed revisions are made to the code. As providing constructive reviews on a submitted patch is a challenging and error-prone task, the advances of Large Language Models (LLMs) in performing natural language processing (NLP) tasks in a human-like fashion have prompted researchers to evaluate their ability to automate the code review process. However, outside lab settings, it is still unclear how receptive reviewers are to accept comments generated by LLMs in a real development workflow. We contribute to the sustained integration of advanced ML models in review processes through large-scale user studies of new LLM-based techniques in mixed closed-/open-source settings, for two major tasks.

The first task is **(STUDY 3) hot-spot prediction-based file ordering for code review**, where a "hot-spot" refers to the locations in a submitted patch that most likely require focused review activity (i.e., review comments or revisions). Previous work has introduced approaches to identify problematic parts in reviews, i.e.,

parts of the code that need to be commented on and/or revised [13, 32]. Henley et al. argue that such approaches are useful to warn about the parts of the code that need to be focused on [42]. Baum et al. argue that reviewers might benefit from seeing those changes first and avoid review fatigue, which would otherwise lead to the hot-spots being missed [43]. However, most review environments display changes in alpha-numeric order, which is suboptimal [31].

We propose a new approach for ordering files in patches based on the likelihood of the change being a hot-spot. In the context of this approach, we empirically evaluate on three open-source (representing 1k to 5k patches) and two closed-source projects (representing 34k and 40k patches) the impact on the prediction performance of using LLM embeddings compared to traditional software analytics features. Using a user study involving 29 industrial reviewers, we also evaluate the potential impact of file ordering on reviews.

The final study focuses on **(STUDY 4) code review comment generation**, in which we explored the reviewers' acceptance of review comments generated by off-the-shelf LLM models using chain-of-thoughts, Retrieval-Augmented Generation (to add code review context and relevant comment examples) and LLM-as-a-Judge (for the LLM to self evaluate its generated comments and filter out the wrong comments). Through a mixed open-/closed-source user study involving 59 expert reviewers, we explore the impact of such models on the reviewers' interaction with the comment suggestions and on the actual adoption of the accepted generated comments in follow-up revisions.

1.3 Main findings

1.3.1 Study 1: Dynamic re-training scheduling strategies for brown build software analytics models

By identifying the TTL of the data and the model, MLOps Engineers can select more wisely what data to use for the next training and when to perform the update. Based on our empirical study on four projects covering 17 to 43 months of data, we found that our proposed scheduling heuristics based on performance decay and model aging were able to keep the model up longer, i.e., 5-6 weeks compared to weekly training, while maintaining F1-score performance. We also show that 30 weeks of data was sufficient when retraining models, which reduces the size of the training set, allowing us to discard old data points, while maintaining prediction performance. Furthermore, for one of the datasets, we found that adding older data even harmed the performance.

Finally, we found that, when using RFS, continuity between models is of concern, as most of the features selected at a given re-training time will not be selected by follow-up re-trainings. We observe that the feature survival ratios (i.e., percentage of features re-selected by follow-up trained models compared to an initial model) drop to 58-76% after a week, then consistently decrease further over time. This suggests that switching to a newer model could improve model performance, but introduces inconsistent behavior compared to preceding models.

1.3.2 Study 2: Lifelong learning setup for software analytics models

Through our empirical evaluation of our LL approach on 8 datasets, covering 1 to 9 years of data, we show that LL in practice manages to at least match traditional RFS

performance in terms of F1-score, while using 3.3-13.7x less data at each update, thus considerably speeding up the model updating process. Considering both the computational effort of updates and the time between model updates, the LL setup needs 2-40x less computational effort than retraining-from-scratch setups, thus clearly showing the potential of LL setups in the industry.

1.3.3 Study 3: Hot-spot prediction-based file ordering for code review

Through a user study with 29 expert reviewers from Ubisoft, we found that hot-spot-based re-ordering, of the file changed by a patch within the review environment has potential to improve review quality, as more comments are written (+23%), and participants' file-level hot-spot precision and recall increases to 53% (+13%) and 28% (+8%), respectively, compared to the alpha-numeric ordering.

For this reason, we propose software analytics models to predict which files in a submitted patch need to be (1) commented, (2) revised, or (3) either (i.e., a hot-spot), which can be used to identify the optimal file-ordering. To predict hot-spotness, we evaluate two different types of code embeddings (i.e., Bag-of-Words and Large Language Models encoding) and review process features (i.e., code size-based and history-based features). Our empirical study on three open-source and two industrial datasets shows that combining the code embedding and review process features leads to better results than the state-of-the-art baseline [13]. For all tasks, F1-scores (median of 40-62%) are significantly better than the state-of-the-art (from +1 to +9%).

The hot-spot prediction based file-ordering is capable of presenting hot-spots earlier in the review file list, which would lead to a positive impact on the review process, as suggested by our user study. When using the hot-spot prediction task for

file-ordering, we found that the recall of hotspots in the top-half of reviews is between 62% and 90%, depending on the size of the review, but most notably is always better than the alpha-numeric ordering.

1.3.4 Study 4: Code review comment suggestion

Finally, apart from improving file-ordering, we explore the possibility to assist reviewers with review comment generation by conducting the first large-scale empirical user study on the acceptance of LLM-generated review comments and their impact on the review process. This user study was performed in a mixed setting of two organizations, Mozilla (which has its codebase available as open-source) and Ubisoft (fully closed-source). Inside their usual review environment, participants were given access to RevMate, an LLM-based assistive tool suggesting generated review comments to reviewers, using an off-the-shelf LLM with Retrieval-Augmented Generation to provide extra code and review context, combined with LLM-as-a-Judge, to auto-evaluate the generated comments and discard irrelevant cases.

Based on more than 550+ patch reviews provided by RevMate, we observed that 8.1% and 7.2%, respectively, of LLM-generated comments were accepted by reviewers in each organization, while 14.6% and 20.5% other comments were still marked as valuable as review or development tips. Refactoring-related comments are more likely to be accepted than functional comments (18.2% and 18.6% compared to 4.8% and 5.2%). The extra time spent by reviewers on inspecting or editing 36/119 accepted comments, yielding an overall median of 43s per patch, is reasonable. Finally, the accepted generated comments are as likely to yield future revisions of the revised patch as human-written comments (74% vs 73% at chunk-level).

Chapter 2

Related work

2.1 Importance of Continuous Integration for Software development

Continuous Integration (CI) is a major quality assurance practice used in software development where code changes are integrated frequently by the developers into the codebase to reduce risks related to merging several collaborators' work [5]. As a best practice, developers submit their code changes regularly (sometimes several times a day) to the integration pipeline for the code to be validated and eventually integrated. Such validation involves all activities required to check that the changes do not break the current codebase, nor cause any software quality issues down the line. Such activities are being referred to as so-called "CI quality gates".

CI quality gates can be simplified into two main types of quality gates: automated gates and manual gates. The automated gates refer to the build and test automation process, which prevents regressions during the merging of patches [7]. The manual gates consist of manual review of code by peers (i.e., reviewers), to identify simple logical errors, maintainability issues and coding standard violations [6]. Both types of practices aim to reduce the risk and cost of code integration and project collaboration,

to get quicker feedback, and improve software quality and team productivity [44].

2.2 Software Analytics Models for CI quality gates

To assist or improve CI gates, several types of software analytics models have been introduced by researchers and the industry [14, 21]. In the context of automated CI gates, many tasks have been introduced by the field of software analytics (mining software repositories), such as defect prediction [15–17], bug assignment [18], build outcome prediction [10, 11, 19], test selection and prioritization [20]. Several review automation actions have been studied over the years as well, such as reviewer recommendation [28–30], code change ordering [31], patch quality prediction [13, 32], review comment recommendation [12, 32, 36–38, 40], review comment evaluation [33–35] and code refinement [12, 32, 36].

In the scope of this thesis, we will focus on two software analytics tasks related to automated quality gates for CI, i.e., Just-In-Time (JIT) risk prediction and brown build detection, and two for review automation, i.e., hot-spot prediction for file ordering and review comment suggestion.

2.2.1 Software Analytics Models for automated CI gates

Just-In-Time (JIT) risk prediction

JIT risk prediction predicts the riskiness of a patch based on metrics related to the change, such as the number of lines added/modified/removed, patch author's experience, etc. [45]. The term "just-in-time" refers to the fact that the detection is done while reviewing the code change instead of right before releasing. This has the advantage that the code change is fresh in the developers' mind. Research has

focused on making the prediction explainable, offering mitigation techniques to the developers to improve their code change [46]. Furthermore, researchers have explored both tree-based models [16] and neural network models [47].

In Ubisoft's implementation of this approach, features are fed to an XGBoost-based model [48]. The resulting tool is currently used in almost all projects of our industrial partner as a predictor for developers on the riskiness of their code changes or, in some cases, as an automated quality gate.

Brown Build Detection

The next step in automated CI gates is the build process, which automatically performs compilation and runs all required tests on the code change [44, 49]. Several builds need to be run for a given code change, due to the different environments a given project needs to support (e.g., Linux, Windows) or different library versions that need to be tested [50]. Test selection [51] and prioritization [52] are often indispensable to reduce the number of tests needed. Effort-aware approaches can be applied here for build-outcome prediction to reduce the computational effort needed to invalidate a patch rejection when applicable [53]. In theory, successful builds validate the changes, whereas failed builds indicate the need to rectify a code change before integrating it to the project.

One of the build process-related use cases we will study in this thesis is brown build detection. A brown build (or flaky build) is defined as an intermittent build job failure that would change to success just by rerunning [1, 54, 55]. Those inconsistent build failures are often attributed to the build scripts themselves [56, 57] (e.g., missing dependencies or issues with network access) or to flaky tests [58–60]. For instance,

a build could fail if the communication between the CI pipeline and physical game consoles is interrupted. A test could be failing on an under-provisioned/overused CI job worker (due to which operations are executed in a different order than the test expects), but pass otherwise. Our industrial partner, Ubisoft, observed that brown builds make developers lose trust in the CI system. Whenever a build fails, instead of verifying if the code is actually faulty, they would force a manual rerun of the build just to see if the result changes. This unnecessarily increases the build activity on the integration pipeline and the integration time.

Previous approaches to predict whether a build is brown or safe are based on flaky test detection approaches [59, 61, 62] or based on build log vocabulary extraction [1, 54]. Ubisoft developed a latter type of model. This approach takes as input build logs, processes them to extract the most representative (series of) words using KBest feature selection, then vectorizes each log trace with TF-IDF. The vectors are then used as input for an XGBoost-based model which returns the probability of brownness.

2.2.2 Software Analytics Models for manual CI gates

Hot-spot prediction-based file ordering

Through hot-spot prediction, we focus on the subject of patch quality, by prioritizing the part of a code change that most likely needs review activity (i.e., revisions and/or comments). It should be noted that Bosu et al. reported that 34.50% of code review comments from five major Microsoft projects that they studied were not "useful", i.e., did not lead to code modification, thus arguing the need for improving the code review process [33]. Review comments might indicate unclarity in the code [4]

(e.g., the developer did not follow the team's coding convention or the nomenclature is not clear), the need for changes in another part of the submission, or the need for a future code submission treated separately.

However, new review comments are not the only review activity of value. Submission of a new patch revision, i.e., a new version of a patch addressing review comments on an earlier version, indicates that the initial code had to be modified due to an issue (e.g., bug or efficiency) or did not follow some code convention of the team. As both review comments and new code revisions bring value, we aim to predict "hot-spots" as changes that require any of those review activities.

Li et al. explored patch quality prediction in order to identify parts of the code change that need review comments [32]. Their approach "CodeReviewer" uses a Text-To-Text Transfer Transformer (T5) that takes as input the code change and returns a probability that the code change needs a comment. Their T5 model is pre-trained on a dataset relevant to the task, in this case, pre-trained on code changes. Hong et al. proposed the "RevSpot" approach to predict problematic lines, i.e., lines that will be commented on by the reviewer or revised [13]. Their approach uses input based on code vocabulary and they train classification models such as Random Forest, Decision Tree and Naive Bayes. This simpler training approach has the advantage of being less costly, whereas fine-tuning a T5 model on the specific task is known to be computationally expensive. The classification-based approach reports similar results on the "commented" prediction task, indicating that the computational effort of fine-tuning an LLM might not be relevant for this task.

LLMs have been used in the context of Software Engineering for multiple tasks, such as code synthesis/generation [63, 64], review automation [12, 32, 36], and code

fuzzing [65]. In Study 3, we focus on the encoder-only type of task, as, there, we use the LLM as a text embedding approach. The embeddings are then used as input to a classifier, which are easier to train due to their smaller size compared to LLMs, thus yielding a training approach that is a more cost-effective and lower computation-time training approach.

Other past works regarding bug prediction [66], risk prediction [16,46] and quality assurance [45] have shown good performance just by using software process features. Those features range from size-based features (e.g., number of lines added, removed), contextual features (e.g., size of the patch), history-based features (e.g., number of times a file was changed, number of participants on a file), to experience-based features (e.g., experience of developers/reviewers). However, review process prediction tasks have not yet considered exploring those types of features, apart from Hong et al., who considered only two features, i.e., number of lines added and the number of lines removed [13]. Exploring review process features approaches is thus one of the goals of this thesis.

Finally, though both Li et al.'s approach [32] and Hong et al.'s approach [13] focus on identifying to some extent where in a patch the reviewer should look, they do not explore the possibility of modifying the patch order to ease and speed up the review process. Review tools usually list the files concerned by the submitted code change in alphabetical order, even though previous work argues that this might be suboptimal, especially for long code submissions [43, 67]. Based on reviewer interviews, Baum et al. establish principles on code change ordering to help reviewers become more effective [31]. Among those principles, they recommend grouping related changes parts (i.e, files, chunks, lines) and providing information before needed (i.e., showing

a function's code before it is used). However, they did not explore or discuss patch quality prediction as a criteria for code change ordering.

By combining Baum's Principles with approaches that predict patch quality [13, 32] (i.e., predicting if a code change chunk will be commented on/needs revision), we could also prioritize code changes that need revisions in the ordering, so that code chunks that need the most attention are shown early on in the review process, increasing the chances of the reviewer to catch problematic code changes.

Review comment generation

The field of review comment recommendation studies the generation of comments for a submitted patch. Over the years, several such approaches have been proposed, which represent code changes using tokenization [37, 38], word- and character-level embeddings [37], or bag-of-words approaches [12, 32, 36, 40]. State-of-the-art (SOTA) approaches for review comment generation leverage LLMs such as T5 and CodeT5 [32, 36, 68]. Ben Sghaier et al. consider Cross-Task Knowledge Distillation, optimizing combined tasks, comment generation and code refinement [68]. Ben Sghaier et al. consider Cross-Task Knowledge Distillation, optimizing combined tasks, comment generation and code refinement [68]. Tufano et al. [39] investigate the comparison between SOTA LLM-based approaches for code review generation and ChatGPT with a zero-shot cognitive architecture. By manually analyzing generated reviews, they propose a taxonomy of comment types, highlighting the topics that SOTA struggles and succeeds with. The authors report that ChaptGPT as-is performs slightly better on coding tasks while struggling with different aspects when generating code reviews.

Hong et al. [40] propose *CommentFinder*, a tool that retrieves past code review

comments for ongoing review of similar patches and reports them as-is. They motivate their approach based on many other studies that found simpler approaches to be similar or even better than Deep Learning (DL) approaches in the context of software engineering tasks in terms of performance [69, 70] and speed [71]. We believe that generative approaches could take advantage of *CommentFinder*'s idea of selecting comments associated with the patch under review as few-shot examples in the prompt, which we considered in our study.

Prior work [12, 32, 40] uses traditional NLP metrics, such as BLEU score (i.e., a metric of text translation quality) and Exact Match (i.e., the prediction matching exactly the expected output). Tufano et al. report BLEU score around 5% and Exact Match of 3%, and they suggest that these metrics are not perfect, and they might under-evaluate the model's performance, thus the need for either investigating the creation of better metrics or conducting user studies and manual validations by experts [36].

2.3 Background on Machine Learning

2.3.1 Concept drift

The concept drift domain studies the evolution of a learning problem on a dataset that is non-stationary [24]. Concept drift in a dataset can be defined as the shift of a concept [72], where the term concept refers to assumptions made on the dataset distributions [73] in terms of probabilities on the dataset (X) and the class labels (Y). Usually, the assumptions will involve the prior class probability $P(Y)$, the class conditional probability $P(X|Y)$, or the posterior class probabilities $P(Y|X)$.

Drift characterization commonly distinguishes sudden, gradual, incremental and

recurrent drift [73]. In the context of software analytics tools, previous work also suggested approaches to analyze and catalog different types of data drift [74, 75]. MLOps engineers need to solve the problem of adapting deployed models to concept changes, while having to estimate the right time for such changes to happen. An active approach is recommended to balance the performance evolution of the model with the updating effort [76]. Tracking concept drift in software analytics models deployed in long-term production is important to maintain the model performance while the data evolves over time. Concept drift in the software concepts has been studied in defect prediction tools [77] and anomaly detection [78].

ML models typically are stationary ("offline"), i.e., once trained and evaluated on a test set (often only using cross-validation [79]), they are deployed in production as-is. However, the real world encountered in production is dynamic and continuously changing, and the occurrence of drift gradually decreases the performance of the deployed models. To avoid the costs and loss of model continuity of RFS update approaches, researchers have identified the need for developing dynamic/online models, as discussed next [80].

2.3.2 Online Learning for software analytics

Although RFS setups are the state-of-practice for maintaining ML models in evolving environments such as SE, such approaches can be computationally expensive due to training effort and can lead to discontinuity between models, as past models are discarded as a new model is trained.

In the context of software analytics, previous works have studied several aspects of online learning leveraging open-source datasets [22, 23, 26] and closed-source datasets

covering periods of less than a year [81] exploring the evolution of software development process features. Other work has also considered Lifelong Learning for tasks involving encoding and decoding of source code [82, 83].

Tan et al. [26] proposed an online learning approach that continuously updates the training dataset with new data points in the context of defect prediction, then uses resampling techniques and updatable learning algorithms based on recomputed probabilities (Naive Bayes) or clustering. Gao et al. explored online learning for pre-trained NLP models in the context of code summarization, vulnerability detection and code clone detection with a memory-based approach (or replay buffer) [23]. However, those approaches have not addressed the trade-off between performance optimization and maintenance effort, which we consider in Study 2.

Other works for online learning rely on bagging approaches, where bagged incremental models are retrained in two cumulative ways: (1) over time with every newly labeled data point and (2) with regular "replay" trainings involving past data points selected using resampling (to deal with imbalanced data) [22, 81, 84]. Although the results show that bagging tends to improve performance through regularization [85], bagging approaches multiply the computational effort by the number of NN models in the bag, thus increasing considerably the computational effort, especially for the replay trainings, which require more data. As in Study 2, our goal is to have a lightweight approach allowing frequent incremental training using limited resources, we instead consider a single model trained in an LL setup. Secondly, updating the model at each new data point is not practical as this requires substantially more effort for storing the model and for the MLOps engineers to validate the model.

With models continuously evolving on each new labeled data input, monitoring

of model updates becomes challenging and the validation of each new model update by MLOps engineers becomes unfeasible, making the adoption of software analytics tools risky for tasks related to critical parts of project development [86]. To be able to perform a roll-back when needed, each model update should back up a copy of the previous model version's weights, increasing considerably the storage complexity of that type of online setups. We thus need to find a trade-off between having regular updates and still allowing human monitoring of the updated models.

2.3.3 Lifelong Learning in the ML community

The ML research community has introduced Lifelong learning, also known as continual learning, a sub-branch of online learning that dynamically evolves an existing incremental model based on a stream of information (which are usually identified as "tasks"), similar to how humans and animals acquire knowledge, maintain it, and transfer it to new situations [27], in order to solve the problem of concept drift. As computational systems and agents need to be able to adapt their knowledge over long time periods and multiple tasks, especially when data distributions shift dynamically over time, a traditional LL setup could adapt the model over time as new data (e.g., new tasks) is made available. Over time, the data stream will introduce new tasks that the model needs to learn while keeping past acquired knowledge.

The main challenge with lifelong learning is so-called "catastrophic forgetting", which occurs when training a neural network with new data interferes with, or even completely overwrites, previously acquired knowledge [85]. This leads to a drop in model performance, especially for previously learned tasks for which the knowledge is not available anymore [87]. The learning system must thus find a balance between

retraining past knowledge and molding knowledge to new information in order to still be able to learn new tasks [88], when scaling to online learning.

The oldest method in use to reduce catastrophic forgetting is using a memory-based approach. A memory, also called replay buffer, is built over time from data samples and is regularly replayed during retraining, in addition to new samples being added, to remind the model of past observations [89]. Since keeping the entire historical data stream quickly becomes unmanageable in terms of both data storage and training time, and is also counterproductive due to outdated data, usually a simpler approach is to use a First-In-First-Out (FIFO) queue to limit the memory size [90]. Selecting the replay buffer samples to cover a larger diversity can significantly improve the performance of LL [91, 92], similar to approaches like dynamic NN size or regularization [93–95].

In the context of SE tasks, we assume that concept drift is present in the data, as the SE development environment is an evolving domain. To evaluate lifelong learning, the ML community has created artificial timelines of incremental tasks, to simulate the evolution of the learning environment. Examples of datasets used for lifelong learning evaluation are MNIST [90, 96] or CIFAR-10/-100 [97]: in those cases, the dataset is split in tasks, where a task is a label. The different tasks are fed incrementally to the model to learn. In our studies, we split the timeline according to time period, making it a more realistic dataset for the SE domain. At each time step, the model is updated using new data made available [98]. Previously learned tasks/time steps are usually not included in the training set, as the model is expected to maintain the previously acquired knowledge, but they are included in the validation set, to evaluate the capacity of the model to maintain acquired knowledge and the impact

on forgetting.

2.3.4 LLMs and Prompt Engineering

Amidst the many LLMs currently available, GPT, a family of LLMs developed by OpenAI, currently stands out as one of the most popular LLMs. GPT has been used for a variety of applications, among which SE use cases like code generation, summarization, testing, and documentation [99, 100].

To guide LLMs like GPT in performing specific tasks accurately, users make requests to LLMs through textual prompts. This process of defining, maintaining, and improving prompts is the subject of Prompt Engineering, a field that combines artificial intelligence and NLP tasks [101]. Various prompting patterns have been proposed to structure prompts more effectively. For instance, while *zero-shot* prompts rely exclusively on textual instructions given directly to an LLM [102], *few-shot* approaches provide more explicit context to the LLM by including concrete examples provided by the LLM user, to guide the LLM towards the expected behavior [103, 104]. *Chain-of-Thought* consists of providing a set of intermediate reasoning steps in the textual instructions, supporting the LLM to address tasks that require complex reasoning [105].

To improve the quality of a model's output, several techniques have been proposed. For instance, a user's prompt may specify constraints in terms of the style, structure, or type of answer expected from the model. Among those, the prompt can suggest intrinsic characteristics like human pre-defined behavior patterns or persona that the LLM should inpersonate [106]. Also, *Retrieval-Augmented Generation* can be used to enhance the prompt with extra information relevant to the prompt retrieved

from a corpus of user-provided documents, before the enriched prompt is sent to the LLM [107]. For instance, RAG has been used in SE for code generation [108] and code summarization [109]. Finally, *LLM-as-a-Judge* leverages the capacity of an LLM to auto-evaluate its generated content, which can be used to improve review comment generation through filtering or iterative generation [110]. As an example, LLM-as-a-Judge has been explored by the software community regarding code generation tasks, to follow given coding preferences [111].

Chapter 3

STUDY 1: Dynamic re-training scheduling strategies - Concept Drift detection for Brown Build Prediction

3.1 Introduction

The majority of software analytics research for CI automated gates trains their software analytics models for CI automated gates once, then serves the resulting model in a static manner for all future predictions. However, such static models are not sustainable as they cannot take into account the evolving nature of software engineering activities, and hence will experience concept drift and performance decay [24].

To avoid the impact of concept drift, the state-of-the-practice in MLOps is to use a Retraining-From-Scratch (RFS) setup [86]. In this case, a new model is trained periodically, to take into account the newest data points. The training periodicity is often based on a set schedule and the training set typically is composed of all available data points at the time of training, increasing the training data set over time.

Intuitively, one might think that the more training data we gather, the better

results we will get. However, training on a large amount of data can be time- and resource-consuming [112]. Furthermore, old data might get outdated, with given data patterns never reappearing in the newest software engineering activities. Within the context of detecting brown builds, logs produced by CI pipelines can become outdated, for instance when flaky tests [59] are removed by fixing the tests or by fixing the infrastructure on which the tests are executed, or when redundant tests are removed from a test suite.

Hence, one objective in this study is to identify the extent to which conserving outdated training data hinders the performance of the resulting models. Determining which subset of the data is relevant for training might improve model performance (as old data might be contradictory to recent data) and reduce training cost (as less data requires less training compute and storage).

Furthermore, we evaluate the question of when to retrain a model. On the one hand, retraining after each new labeled data point is computationally expensive, and the benefit of adding a single new data point to the training set might be relatively small. On the other hand, using the same model forever disregards any new data, reducing its ability to perform well on newer data points [38]. Analyzing such data drift allows us to take informed decisions on when models should be retrained. Dynamically determining the optimal training time might improve the trade-off between performance and training cost, as more or less training could be required, depending on the degree of concept drift at a given time.

In this study, we evaluate how well models can deal with and mitigate data concept drift, once the timeline of the data stream is taken into account, by measuring the Time-to-Live (TTL) of models and data. In particular, we aim (1) to evaluate the

minimum size of training dataset needed for a model to achieve optimal prediction results while limiting the computational effort of training and (2) to evaluate when to change models (and thus retrain from scratch). We will focus our empirical study on a brown build detection prototype that we developed during my Master, in collaboration with Ubisoft. Before my PhD started, the approach was only validated in a static cross-validation setup, not respecting the timeline of the data stream. Furthermore, during my Master, we had not evaluated how well the model would sustain over time, and when it should be retrained to maintain performance.

This chapter focuses on the following Research Questions:

RQ1 - What is the most relevant training set for Retraining-From-Scratch of brown build detection models at a given training time? We propose a methodology to determine the optimal training set for RFS. Through our empirical study focused on the brown build detection use case, we found that data older than 30 weeks does not significantly improve the F1-score performance (<0.01%), and even harms the model for one of our datasets.

RQ2 - How is the continuity between models impacted by concept drift? In order to measure continuity in the behavior of a model, we propose to measure the feature surviving ratio between an initial model version and model versions retrained at each follow-up timestamp. We observe that, after one week, the feature surviving ratio drops to 58-76%, then consistently decreases over time. This indicates that, although there is some continuity between retrained model version in terms of retained model features, new features are rapidly becoming important, hampering the continuity between model versions

RQ3 - What is the most effective retraining schedule for Retraining-From-Scratch of brown build detection models? We explored different heuristics to determine when to retrain the models from scratch. On the specific use case of brown build detection, the best heuristics were cumulative approaches (e.g., based on performance decay) for 2/4 projects and fix approaches (e.g., based on model aging) for the two other approaches. Those approaches were able to schedule the retraining of the model every 2-6 weeks while maintaining performance compared to weekly retraining.

3.2 Methodology

To determine the optimal training data set and RFS retraining schedule, our approach aims to determine the degree of concept drift on software analytics models, in terms of their performance over time. First, we propose a methodology to evaluate the needed training window size. Second, we evaluate the continuity of the retrained models by exploring the selected features' survival ratio. Finally, we propose a number of model switching heuristics, to decide when to retrain a model from scratch.

3.2.1 Recurrent retraining setup

To study the impact of concept drift, we need to simulate, for each available use case, retraining of the initial model over time. In this study, the model is trained on a RFS-based setup, with weekly retraining (sub-dataset being split into weeks from Sunday to Monday) of an entirely new model on past data, followed by prediction on data points posterior to the training data. The model is not changing once trained.

Furthermore, each weekly retrained model will be our baseline until the next

model (at the next time step) is trained. That baseline is an intuitive approach but might be computationally costly, as re-training each week might not be necessary in order to maintain performance. Based on these different baseline models and their predictions, we will perform concept drift analysis on the knowledge models acquired during training (i.e., their feature selection) and outputs, identifying how model performance evolves over time and how the models themselves change.

3.2.2 Training dataset reduction approach

As a first effort to reduce the dataset size for training, we propose a sliding window approach where a window of a given size is used as training data size for RFS, and is shifted across the whole data set to simulate a model being retrained each week: each retraining is referred to as a drifting step (see Figure 3.1). By varying the size of the training window, we can evaluate how far in the past data must be retrieved to achieve relevant prediction performance.

The three sets needed for our model computation and validation (training, validation and test sets) are computed respecting time isolation (**Isolation-Hypo**), by selecting consecutive ordered groups:

Isolation-Hypo time isolation: The data-set weeks are ordered such that if week i comes before week j, all the jobs in week i precede all the jobs in week j.

The sets are then defined for each drifting step i:

- Train set: $\{\text{build} \in \text{week } j \mid j \in [i, i + win_{train} - 1]\}$
- Valid set: $\{\text{build} \in \text{week } j \mid j = i + win_{train}\}$
- Test set: $\{\text{build} \in \text{week } j \mid j > i + win_{train}\}$

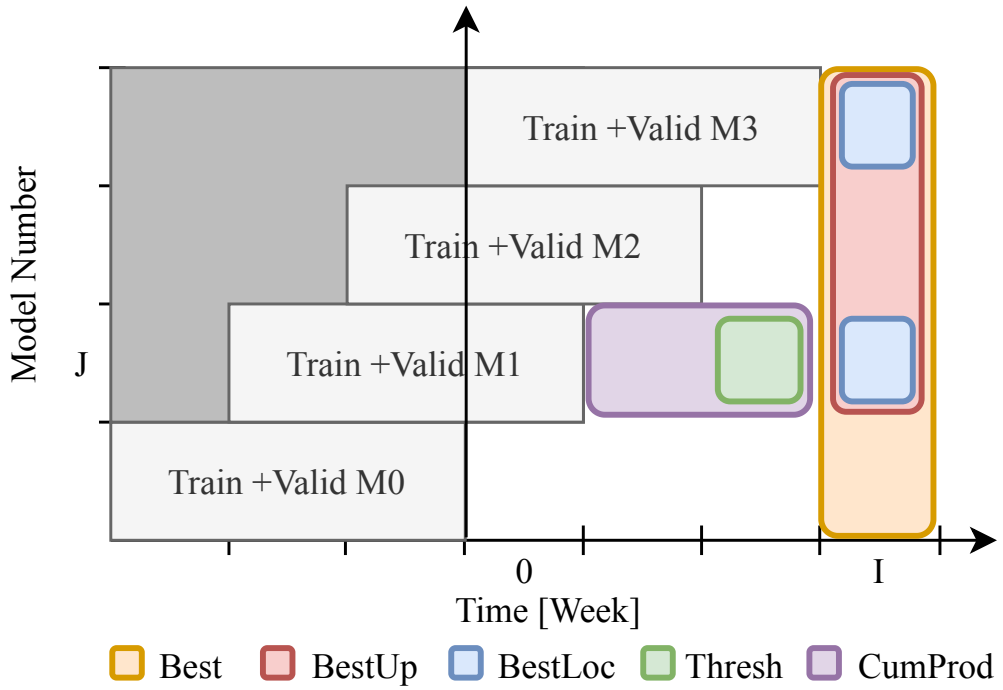


Figure 3.1: Switching heuristics. We suppose that the current Week is I and the model selected used at $I-1$ is m_j. Switch heuristics use the colored zones as criteria for model selection.

To evaluate how the model ages across time, we evaluate the performance on the test sets for each drifting step: we can then observe the concept drift impact on a given software analytics model by observing the performance of each drifting step by groups (weeks), and by the number of weeks since the model was trained.

3.2.3 Feature survival ratio analysis

To analyze whether the studied models actually experience concept drift other than in terms of performance, we also evaluate the continuity between successive model versions' most important features. Machine learning models typically rely mostly on a small set of features having the most predictive power, i.e., the highest

feature importance [113, 114]. By comparing this set between successive model versions, we can understand how quickly the retrained model versions are diverging from the initial model versions.

For this, we measure the weekly feature surviving ratio, where model i's feature surviving ratio at week j is the percentage of features selected at week i that still are selected in the model of week j. If the surviving ratio decreases consistently over time for all models, then the feature selection and the model are impacted by concept drift. The feature surviving ratio is computed as the ratio of features selected by the training that still appear in a new model, x weeks after the training of the reference model: for instance, if model_3 has a feature surviving ratio of 60% after week 4 of usage, that means that 60% of the features from model_3 still appear in model_7 (4 weeks after).

We will analyze the median feature surviving ratio of models after x weeks of existence. This helps us evaluate how the training evolves over time and how important retraining can be: if the ratio is too low, the features are not relevant anymore to the prediction since a new model would have chosen other features with higher importance.

3.2.4 Model switching heuristics

The evolution of the performance of models retrained across drifting steps can then be used to evaluate when to change from a model based on group (week) i's data to a model based on group j in order to obtain the best performance. For this, we define different switching heuristics that will be used to decide when to retrain and switch models. In those algorithms, either the switching algorithm will be *a*

priori or *a posteriori*. *A priori* algorithms will choose to switch to the model of week i based on information about all weeks j before i ($j < i$). Those algorithms are applicable in practice since all the information they require is available at the moment of switching. *A posteriori* algorithms will choose to switch to the model of week i based on information about all weeks j before i and week i included ($j \leq i$). Those are unrealistic models since they need information about the current week before it even happened. We still consider such algorithms as optimal baseline models the a priori algorithms can be compared to.

For the definition of the model switching heuristics, let us assume that the drift parameters are $win_{train} + win_{valid} = W$ (with $win_{valid} = 1$ the window size of the validation set) and that we have $W + N$ weeks of data, and that for each week w_i with $i \geq 0$ there is a model m_i created with the data from weeks $[w_{i-W-1}; w_{i-1}]$, with $i \in [0, N]$. We also define the model m_{s_i} as the model selected by a switching heuristic at w_i. The performance of model m_j on week w_i is $perf(i, j)$. Figure 3.1 shows an example with $W = 3$.

For the following switch heuristic descriptions, we suppose that we are currently starting week w_i and that the previous chosen model $m_{s_{i-1}}$ is m_j. Each of the following description indicates how the model for week w_i will be chosen. The "a posteriori" approaches are indicated with a start "*".

Best* : the model chosen is chosen among all available models at that time: at week w_i is the model m_k with $0 \leq k \leq i$ that has the highest performance $perf(i, k)$, as shown in orange on Figure 3.1.

BestUp* : the model is chosen among any model trained since the last chosen model (included): at week w_i is the model m_k with $s_{i-1} \leq k \leq i$ that has the highest

performance $perf(i, k)$, as shown in red on Figure 3.1.

BestLoc* : the model is chosen between the last chosen model and the newest model: at week w_i is the model m_k with $k = s_j$ (previous best chosen model) or $k = i$ such that the highest performance $perf(i, k)$ is obtained, as shown in blue on Figure 3.1.

Weekly : the model chosen is always the latest: at week w_i is m_i, without looking at other models' performance (switch every week to the newest model).

Fix : the model is switched every fixed number of weeks to the most recent model (i.e., switch every 4 weeks). If week w_i needs to switch (because the model has been used for the fixed number of weeks), the chosen model is m_i. Otherwise, keep m_{s_j}, without looking at the performances.

Thresh : the model at week w_i is m_i if the performance $perf(i-1, s_{i-1})$ during week w_{i-1} was lower than a threshold T, as shown in green on Figure 3.1. Otherwise, keep m_{s_j}.

CumulProd : the model chosen depends on the cumulative product of the past model performance: at week w_i is m_i if the cumulative product $\prod_{j=curr}^{i-1} perf(j, s_j)$ of the performance of the models of week w_j with $s_j \leq j < i - 1$ was lower than a threshold T. This is shown in purple on Figure 3.1. Otherwise, keep m_{s_j}. We use a cumulative product such that the model ages over time, since the multiplication will reduce the value over time.

Among these algorithms, algorithm **Best**, **BestUp** and **BestLoc** are weekly upper-bounds for our analysis, since they represent three ways to choose the best

models a posteriori. However, those three heuristics can be changed to be a priori if we look at models performance in the previous week. The issue with those heuristics is that a model still needs to be trained every week and kept. **Best** keeps all models forever, **BestUp** keeps all the models till the model switches to a new model, then it forgets all older models. **BestLoc** only keeps two models, the currently used model and the most recently trained.

Weekly and **Fix** are heuristics based only on the model's age. They will switch when the model has been up for a certain number of weeks.

Thresh and **CumulProd** are more advanced heuristics based on the evaluated performance and age of the model. To compare model switching heuristics, we take the performance over the whole period (weeks w_0 to w_N) with the selected models for each week.

3.2.5 Use case: Brown Build detection

In Continuous Integration (CI) systems, when a code change is submitted to the integration pipeline, builds are generated to compile and test the changes and validate code quality. If all builds succeed (turn "green"), the code change is validated and sent through the next step for integration (usually, manual validation). If the builds fail (turn "red"), the code change is rejected and the developer needs to correct the change before resubmitting to the integration pipeline. However, builds can sometimes fail, not due to the code change being faulty but due to the build system itself (i.e., missing dependencies, server failure) or the tests being launched (i.e., flaky tests). We refer to those unreliable builds as "brown" builds.

During my Master, we developed a language- and project-agnostic method for

brown build detection. The dataset is composed of the build jobs' log file and some meta information (i.e., job's start date, job's id, commit's id). Based on the job's log content, we used TF-IDF [115] to extract features. To limit the vocabulary size, we used KBest[1] for feature selection. Additional metrics are computed: for each job, we compute the number of prior reruns, prior success, and prior fails, as well as the number of jobs since the last brown build was observed.

The training method starts by the data going through a preprocessing step, where the build job content is vectorized. The content of the build jobs log is first reduced, by filtering the vocabulary (i.e., removing special symbols, identifying ids, path/files names, etc.). Then, vectorization is applied using a TF-IDF vectorization algorithm on the jobs' content. We also apply KBest feature selection method to reduce the vocabulary of the TF-IDF vectorization. Once the data vectorization is done, the reduced TF-IDF vectorization and the other metrics computed on the metadata go through an XGBoost classifier [48], which is a boosted random forest algorithm.

The setup was optimized and validated in my master's thesis based on a cross-validation setup, a cross-project validation and manual validation by experts. However, the basic model was never retrained, nor was a chronologically separated training vs. test data split used. Our empirical evaluation aims to fill that gap, by analyzing the TTL of models and data to optimize performance while reducing the computational cost of retraining a model every week.

[1]Sklearn selectkbest python package: https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.SelectKBest.html

Table 3.1: Information on the full history of the studied projects for brown build detection.

Project	**Full project history**			
	#Contributors	**#commit**	**Main Language(s)**	**Age** [y]
Brown_A	15	3k	Go, JS	3.5
Brown_B	29	2k	C#	4
Brown_C	64	7k	C++, C	3
Brown_OS	20	14k	C,C++	17

Table 3.2: Information on the scraped part of the studied projects for brown build detection.

Project	**Scraped data**				
	#months	**#jobs**	**#failing jobs**	**Brown failure ratio** (BFR)	**Mean job duration** [mm:ss]
Brown_A	18	23k	3008	30%	03:27±04
Brown_B	29	63k	8100	37%	03:14±14
Brown_C	17	22k	5986	5%	15:49±30
Brown_OS	43	47k	1237	13%	06:12±17

3.2.6 Design of Empirical Evaluation

The dataset considered for this study covers 4 projects, 3 closed-source dataset provided by Ubisoft and one open-source project, Graphviz[2]. The dataset for those projects was scraped from Gitlab, using its APIs[3]. Some information on the studied projects can be found in Table 3.1 (regarding the studied projects) and Table 3.2 (regarding the data scraped for said projects). The name of the closed-source projects are anonymized for privacy reasons (projects Brown_A/Brown_B/Brown_C). The open-source project Graphviz will be referred to as project Brown_OS.

The particularity of this project is thus that we need to deal with text input (log files) and select the right features to extract from it.

[2]Graphviz website: https://graphviz.org/
[3]GitLab API: https://docs.gitlab.com/ee/api/

Table 3.3: Intuitive baselines (with BFR the brown failure ratio by project).

Baseline name	**Chances of brown prediction**	**Precision**	**Recall**	**Specificity**	**F1-score**
Random50	50%	BFR	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{\frac{1}{BFR}+2} = \frac{BFR}{1+2BFR}$
RandomB	BFR%	BFR	BFR	$1 - BFR$	$\frac{1}{\frac{1}{BFR}+\frac{1}{BFR}} = \frac{BFR}{2}$
AlwaysBrown	100%	BFR	1	0	$\frac{1}{\frac{1}{BFR}+1} = \frac{BFR}{1+BFR}$

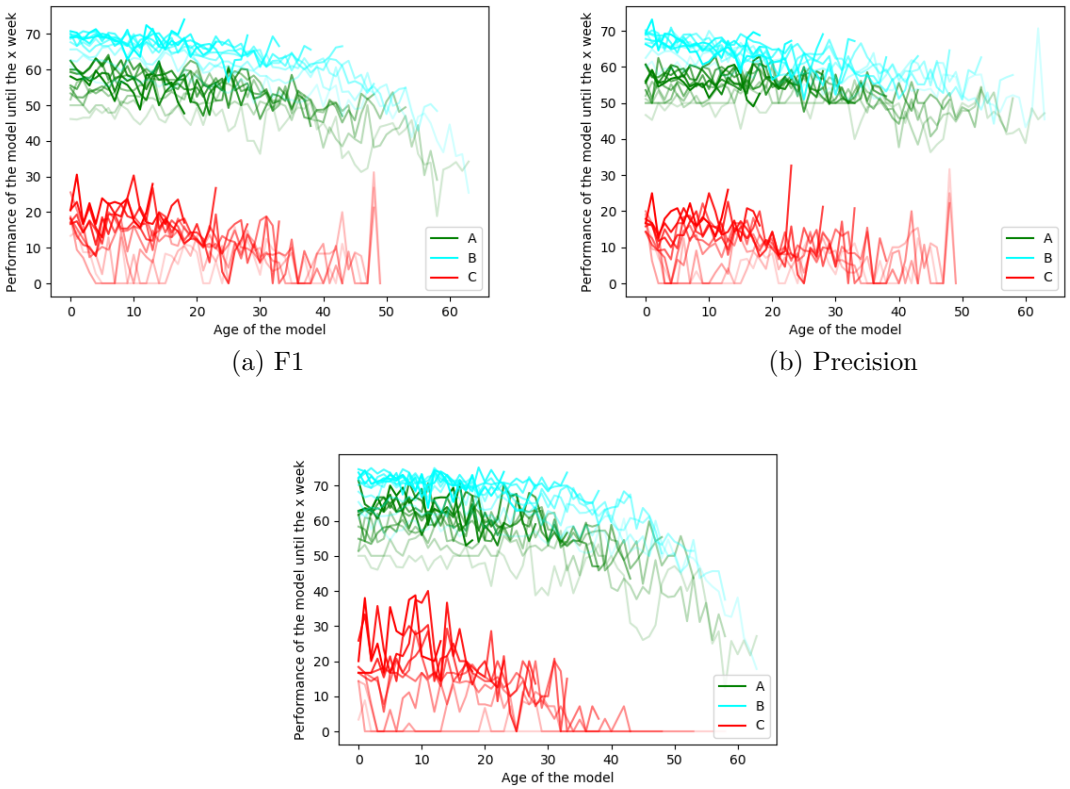
3.3 Baselines

Regarding the performance of the retraining setup, we want to compare the results to baselines. To our knowledge, no other tool for language- and project-agnostic brown build detection approaches have been developed, let alone in an online setup such as the weekly retraining we are doing. We will compare our results with some intuitive baselines using the brown failure ratio (BFR)[4] and with the results of the cross validation setup of the model found in my Master's thesis.

The intuitive baselines are based on random prediction of a build being brown. The probability of predicting brownness is either 50% (*Random50*), 100% (*Always-Brown* or based on the brownness of the dataset (*RandomB*). Those baselines are described in Table 3.3.

Regarding the previous results of the prototype, we used a 10-fold cross validation setup with random splits of the dataset (which does not respect the time isolation hypothesis **Isolation-Hypo**). We can compare the results of the weekly retraining setup with those results, while keeping in mind that the **Isolation-Hypo** is not respected and thus the results of the cross-validation setup will be higher than the

[4] BFR is the number of failed builds that are brown divided by the total number of failed builds.



(a) F1

(b) Precision

(c) Recall

Figure 3.2: Optimization of the drifting window size, windows varying from 5 to 50 (step of 5). The darker the line, the larger the window size is.

new setup.

3.4 RQ1: What is the most relevant training set for Retraining-From-Scratch of brown build detection models at a given training time?

A window size of 30 weeks of data is sufficient to have significant performance, since adding more weeks does not considerably increase the performance ($< 0.1\%$) and, in the case of project Brown_A, using more than 40 weeks even harms the performance. Figure 3.2 shows the median

performance (F1-score, precision, recall) of the models over time, depending on the project and win_{train} training window size. The (x, y) values on line l represent:

- x: the number of weeks since the models were trained;
- y: the median F1-score of all models over time from training (week 0) to week x;
- l: the colors indicate the project, and the more transparent the line, the smaller the size of the win_{train} was.

We can see here that all lines decrease between 0 and 10% per week. This decrease in performance over time indicates that retraining a new model at some point would be beneficial. We also see that the smaller the win_{train} size is, the lower the F1-score is. For each color the darker lines have larger y values. However, as we increase the win_{train} size, the median improvement on the F1-score becomes $< 0.1\%$.

For Project Brown_A, a closer analysis showed that win_{train} sizes 45 and 50 obtain an F1-score worse than 40. This shows that using too old data harms the model, since the data is not coherent with more recent data. Closer analysis showed that a win_{train} size between 30 and 40 gets the best results for all studied projects.

Project Brown_OS was ignored for this first experiment since the number of weeks was 48 and we needed up to 50 weeks of data in the training set.

RQ1: Our methodology for selecting the training window size shows how 30 weeks of data is sufficient to optimize performance, as adding more data did not improve performance significantly, and even harmed the results for one dataset.

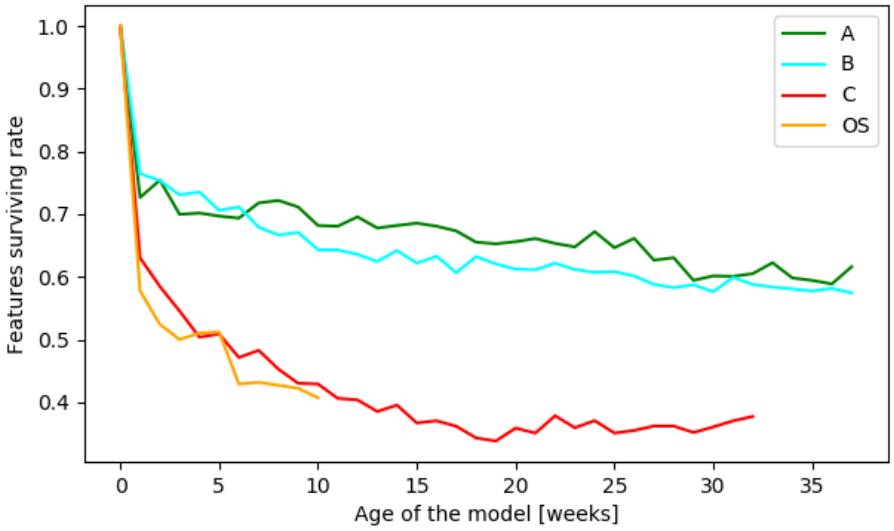


Figure 3.3: Features surviving ratio per project with a win_{train} size of 30.

3.5 RQ2: How is the continuity between models impacted by concept drift?

The training feature surviving ratio drops down to 58-76% after one week, then decreases consistently over time for all projects.

Figure 3.3 shows the training feature surviving ratio for a win_{train} size of 30 weeks (since we found that this was the optimal value). Projects Brown_A and Brown_B show similar behavior, dropping quickly to a median feature drifting ratio of 72% and 76% respectively, then reducing linearly to 60% at week 38. Project Brown_C and Brown_OS both have a median feature surviving ratio drops to 63% and 58%, then reduce to 35% at week 30 and 40% at week 10 respectively.

The first drop for the four projects seems to be due to features local to the week of the training, only relevant to the prediction of the first few (1 to 2) weeks after training. The consistent decrease after that shows that other features get rejected

over time, even if they stay relevant longer. The trained models evolve quickly over time, prioritizing different features as the data evolves, thus suggesting that newer models might be more relevant and yield better performance. This indicates that, although there is some continuity between models, as part of the selected features survive from one training to another, that percentage consistently decreases, suggesting that switching to a newer model would optimize the use of features. Moreover, this is a major issue of RFS in general, as this discontinuity somehow is similar in nature as the catastrophic forgetting issue of LL setups, since the model has no recall of features found important in prior training.

RQ2: There is only local continuity between re-trained models, as 58-76% of features survive after one week, yet the feature survival ratio decreases consistently over time, suggesting that the more recent models becoming more relevant differ substantially from the previous models.

3.6 RQ3: What is the most effective retraining schedule for Retraining-From-Scratch of brown build detection models?

A priori switching heuristics yield an F1-score about 7-10% lower than a posteriori heuristics, while keeping results 2-to-4 times higher than the baseline *AlwaysBrown*.

Table 3.4 shows all performance metrics that we used, as well as the life expectation *LifeExp* (in number of weeks) when using a particular switcher. We observe that the a priori switching heuristics have an F1-score 7-10% under the a posteriori heuristic *Best*, except for the project Brown_OS. In the case of project Brown_OS, the

Table 3.4: Comparison of heuristics with a win_{train} size of 30. The table is split vertically to fit two projects per row, then each section is split vertically again into a posteriori heuristics, default heuristics (first and weekly), a priori heuristics and baselines. Bold indicates the line with the best F1-score of a project per category.

Heuristics	**Project Brown_A**						**Project Brown_B**					
	Brown			Safe		Life	Brown			Safe		Life
	F1	Pre	Rec	Pre	Rec	Exp	F1	Pre	Rec	Pre	Rec	Exp
best (post)	**69**	**70**	**69**	**87**	**87**	**1**	**74**	**79**	**70**	**89**	**83**	**1**
bestup (post)	65	65	64	85	85	2	71	78	65	89	81	2
bestloc (post)	63	64	62	85	84	2	70	77	63	89	80	2
first	51	56	46	84	78	43	62	70	56	86	77	43
weekly	**56**	**59**	**54**	**84**	**81**	**1**	**67**	**75**	**61**	**88**	**79**	**1**
fix (2 weeks)	57	59	54	84	81	2	66	73	61	86	78	2
fix (5 weeks)	59	60	57	84	82	5	66	73	61	86	79	5
fix (10 weeks)	56	59	54	84	81	10	67	71	64	85	79	10
fix (15 weeks)	56	59	53	84	81	15	66	70	62	84	78	15
cum ($T = 0.01$)	**60**	**61**	**60**	**84**	**83**	**6**	**67**	**73**	**62**	**86**	**79**	**5**
cum ($T = 0.02$)	59	61	58	84	82	4	66	74	60	87	78	5
cum ($T = 0.05$)	60	60	59	83	83	4	66	74	60	87	78	4
thresh ($T = 0.7$)	59	60	58	84	82	1	67	74	60	87	79	1
thresh ($T = 0.8$)	59	60	57	83	82	1	67	74	60	87	79	1
AlwaysBrown	23	30	100	na	0		27	37	100	na	0	
Cross-valid	**62**	**55**	**72**	**75**	**86**		**67**	**57**	**81**	**64**	**85**	

Heuristics	**Project Brown_C**						**Project Brown_OSS**					
	Brown			Safe		Life	Brown			Safe		Life
	F1	Pre	Rec	Pre	Rec	Exp	F1	Pre	Rec	Pre	Rec	Exp
best (post)	**32**	**59**	**21**	**99**	**96**	**2**	**36**	**28**	**48**	**95**	**98**	**4**
bestup (post)	30	57	20	99	96	3	36	28	48	95	98	4
bestloc (post)	22	45	14	99	95	6	21	24	19	97	97	8
first	14	35	9	99	95	38	32	27	39	96	97	16
weekly	**22**	**45**	**15**	**99**	**95**	**1**	**44**	**44**	**45**	**98**	**98**	**1**
fix (2 weeks)	**27**	**50**	**19**	**99**	**96**	**2**	39	40	39	98	97	2
fix (5 weeks)	24	44	16	99	95	5	**42**	**39**	**45**	**97**	**98**	**5**
fix (10 weeks)	20	40	14	99	95	10	38	33	45	96	98	8
fix (15 weeks)	22	42	15	99	95	15	34	29	42	96	98	8
cum ($T = 0.01$)	22	39	15	99	95	5	23	24	23	97	97	8
cum ($T = 0.02$)	20	42	13	99	95	4	23	24	23	97	97	8
cum ($T = 0.05$)	20	38	14	99	95	3	29	29	29	97	97	5
thresh ($T = 0.7$)	26	52	18	99	96	2	35	35	35	97	97	2
thresh ($T = 0.8$)	26	52	18	99	96	2	35	35	35	97	97	2
AlwaysBrown	5	5	100	nan	0		12	13	100	na	0	
Cross-valid	**36**	**35**	**38**	**96**	**97**		**52**	**47**	**57**	**91**	**91**	

Best heuristics are lower than the weekly due to the fact that taking the best model at each week, in this case, lowers the overall precision and recall, and hence the F1-score as well (known as Simpson's paradox [116]). Compared to *weekly* (switching every week), *CumulProd* gets the same F1-score for project Brown_B and Brown_C and a higher score for project Brown_A, while needing less training computation (the median *LifeExp* for project A is 6 and for project Brown_B and Brown_C is 5). Project

Brown_OS's F1-score is 2% lower than the weekly with the *Fix (5 weeks)* heuristic.

Furthermore, when comparing our switch heuristics with the baseline *Always-Brown*, we observed that our F1-score is two to four times better. When compared with the cross-validation performance, the F1-score performance loses 0-10% depending on the project when including the time constraint and using a priori switch heuristics instead of a posteriori heuristics.

The brown build detection prototype, once extended with a switch heuristic, maintains performance significantly higher than the baseline. The model can adapt through time, obtaining the best performance possible by relying on switch metrics based on performance decay (i.e., threshold heuristics), model aging (i.e., fix heuristics) or on both (i.e., cumulative heuristics).

RQ3: With a priori heuristic, we manage perform 2-4X better than the baseline and similarly to the weekly training (*weekly*) while getting median life-expectation of the models to 2-6 weeks depending on the project.

3.7 Threats to Validity

External validity. Our approaches were evaluated on four large projects with various brown ratios and build activity. We observed that the recall and precision performance decrease as the brown ratio decreases. A project having a lower brown failure ratio than 1% might not justify the need for our algorithm since the performance might not be sufficient.

We considered only one open-source project, Graphviz, among the seven studied projects. Further analysis on other projects showing brownness would be relevant

to validate the generalization of our results. However, finding projects that have a relevant brown failure ratio can be difficult, hence us only working with one open-source project.

Construct validity. Our brown job identification ground truth might be mistaken since we base the a posteriori identification on the reruns of jobs on the same commit-ID. If a job is rerun at least once and changes results for the same commit-ID as the job is identified as brown. However, most of the build jobs are run only once for a commit-ID, since brown builds are rare, as shown by our studied projects: those are systematically identified as true failures or successes. We thus might be mislabeling some brown jobs as true failure, which could impact the models' prediction performance range, which could notably lead to keeping models up longer in the case of threshold-based scheduling heuristics.

3.8 Conclusion

In this chapter, we study the impact of concept drift on a typical software analytics model for automated CI quality gates, showing how models and data age over time and impact the performance and sustainability of ML approaches. We observe that a dataset of 30 weeks is enough to get good performance while increasing the size of the dataset further harmed performance in the case of project Brown_A. We find that there is only partial continuity between the trained models as feature survival ratios drop to 58-76% in the first week, then consistently decrease over time, suggesting that, although newer models could become more relevant by updating their selected features, in doing so only limited continuity is left between model trainings.

Furthermore, training less often requires less computation effort (retraining every

few weeks instead of every week): we found that we are able to determine when a model should be retrained using switching heuristics, while maintaining performance similar, if not better, than weekly retraining and keeping the model up longer, 2-6 weeks. We thus manage to predict more quickly the brownness of a build job while keeping the performance similar to explanatory models with optimized hyper-parameters.

This chapter is part of our publication at IEEE/ACM International Conference on Software Engineering (ICSE) 2022, "Towards Language-independent Brown Build Detection" [1], in collaboration with my co-authors, Mathieu Nayrolles (Ubisoft) and Bram Adams (Queen's University).

3.9 Future work

First, we applied the approach only on the brown build detection use case: exploring other use cases, which might present more or less concept drift than brown build detection would be interesting to consolidate our findings and identify the relevancy of the approach depending on the severity of the impact of concept drift on the task.

Finally, with RFS setup, we still need to retrain from scratch the model every 2-6 weeks for the studied projects, which can be computationally expensive. Furthermore, continuity between model trainings should be a concern, as we observe that the selected features change across trainings. To solve the lack of continuity, we would need to switch from RFS towards techniques like lifelong learning that incrementally evolve a given model when new data arrives. Such LL approaches could potentially result in less computational effort during model updates and in more continuity as the same model evolves over time instead of switching to new models altogether at

each time step. This is what led us to explore the topic of Chapter 4 (Study 2).

Chapter 4

STUDY 2: Lifelong learning setup for software analytics models

4.1 Introduction

While software analytics models are increasingly used in industrial settings, e.g., for build failure prediction in Continuous Integration (CI) [117], test case prioritization [118] or code recommendation [119, 120], such models tend to become outdated over time as they are being used in production [77]. MLOps engineers refer to this phenomenon of models and dataset aging as "concept drift" [121–123], which we studied in Study1.

As a consequence, in practice, drifting models are discarded and replaced by new models, trained from scratch on the existing datasets for the task, leading to model continuity issues. Such retraining-from-scratch (RFS) setups are also expensive to maintain, and the training itself becomes more and more computationally expensive as the dataset grows in size [124–126]. Due to this cost, retraining of models happens sparingly and is usually triggered when a clear performance decay is noticed. Also, as we saw in the previous chapter, RFS setups ignore prior models' knowledge, causing

discontinuity in their behavior, which goes against traditional software re-engineering practices [127].

To better deal with concept drift, the domain of machine learning has been exploring online learning approaches. These cover various paradigms trying to mimic the human ability to retrain and accumulate knowledge previously learned [25]. In the context of software analytics, prior work incrementally updates the training data with new incoming data, considering resampling approaches to deal with data imbalance [26, 75], bagging approaches [22, 81, 84] or approaches using memory-based updates [23]. Those works were evaluated on several types of software analytics models in the context of open-source development.

However, such online approaches come with many challenges in practice and have thus rarely been introduced in industrial settings [86], which are constrained by efficiency and limited resources for ML training. Industries also have expectations regarding ML tools, among which the ability to monitor and validate the model, to have better control over its behavior. As such, it is essential to understand both the benefits and costs of online learning approaches in software analytics models in practice, countering the drifting nature of software engineering data when training and deploying.

This chapter empirically compares the use of online learning approaches and RFS approaches in an industrial setting in terms of performance and computational effort. Inspired by prior work on online learning for generative software analytics tasks (e.g., code summarization) [23], Natural-Language Processing tasks (e.g., vulnerability detection, and code clone detection) [23, 26] and other classification tasks (e.g., defect prediction) [26, 84], we explore a paradigm of online learning called “Lifelong

learning" (LL) [27]. Our LL setup uses incremental learning models, i.e., Neural Networks (NN), similar to Cabral et al.'s approach [22], but with the addition of a memory-based mechanism inspired by Gao et al. [23].

Our empirical study focuses on two classification-based software analytics use cases of our industrial partner, Ubisoft, where signs of concept drift have been observed, i.e., brown build detection [1] and Just-In-Time (JIT) risk prediction [16]. Brown build detection is related to flaky test analysis, which is one of the emerging topics in software testing in recent years [1, 59, 61]. JIT risk prediction is a generalization of bug prediction models, a staple of software analytics for 20 years, that focuses on the overall risk (e.g., test failure, code complexity) of a change [128, 129]. Through empirical evaluation on a total of eight large projects, seven from Ubisoft and one open-source, we address the following research questions:

RQ1: How does an LL setup perform compared to an RFS setup for software classification tasks? Comparison with RFS setups for the two Ubisoft use cases shows that the LL setup significantly outperforms the currently used approaches (+9-31% F1 score) for 6 out of the 8 studied projects (large effect size), and behaves similarly in the other 2 projects. We also find that it performs similarly to an NN-based RFS setup in 4/8 cases, significantly better in 3/8 cases and significantly worse in 1/8 case.

RQ2: How essential is the use of a memory-based LL setup? A major issue encountered with LL setups is catastrophic forgetting, where, during updating, the model forgets previously acquired knowledge. To address this, we adopt a memory-based approach that trains the model with additional samples of past data points. We observe that this approach improves significantly the F1-score compared

Algorithm 1 Pseudo-code for training of an NN.

```
Require: trainset with dataset "X" and labels "y", NN, epoch
  for iter in epoch do
    Split trainset in minibatches
    for minibatch in trainset do
      pred = NN(minibatch[X]) //model's outputs on mini-batch
      loss = lossfun(minibatch[y], pred) //loss versus labeled data
      net.update(optimizer(NN, loss)) //update model weights
```

to a more naive LL setup (+4-26%) for 3/8 cases, while not showing significant differences in the other 5 cases (-1% to +6%).

RQ3: How time-efficient is an LL setup compared to an RFS setup? We compare the computational effort needed to perform LL updates with RFS. Even though the LL setup needs 1.1-8.75 times more updates, the LL updates are computationally less expensive as 3.3-13.7 times less data is necessary for retraining, on average. The LL setup reports thus 2-40 times less computation effort.

Finally, we discuss trade-offs in terms of the training setups' flexibility to adjust to drift versus continuity in their interpretability.

4.2 Lifelong Learning for Software analytics

4.2.1 Multi-layer perceptron neural networks

An LL setup requires an incremental learning algorithm, such as multi-layer perceptron (MLP) neural networks. A basic MLP has L layers of size N. Between each pair of neighboring hidden layers, an activation function (e.g., ReLU) is applied to the output of one layer before sending it to the next layer in order to have a non-linear transformation [130]. After the final layer, a sigmoid activation function is applied to

bound the final output values between 0 and 1 to obtain probabilities as output. Algorithm 1 shows the pseudo-code for a typical MLP training process [130]. In essence, in each iteration ("epoch") the training algorithm tries to minimize with an optimizer the "loss" between the current model's prediction on subsets ("mini-batch") of the training set and the expected output. The complexity of this algorithm is linearly dependent on the number of epochs and the size of the training set (divided by the size of a mini-batch).

Implementation-wise, the loss function we use is Binary Cross Entropy (BCE), while the optimizer we consider is Adam (with hyperparameter learning rate lr). Both were chosen after preliminary evaluation instead of, respectively, the loss functions Mean Squared Error (MSE) and Mean Absolute Error (MAE), and the optimizer Stochastic Gradient Descent (SGD). The MLP training is done using mini-batches [131] of size 20 and with 50 epochs at each training, since our case studies showed that this was enough to pass the under-training stage and to reach the over-training phase. Once all 50 epochs are run, the model stage at the best epoch (in terms of performance on the validation set) is kept, ready for production. The incremental nature of this algorithm refers to its capacity of updating the pretrained weights of an existing NN model with an updated training set.

4.2.2 Data splitting

In an actual production setup, the ML-models' datasets are split chronologically such that the training, validation, and test sets do not overlap. In the case of LL, these sets have to keep up with time, as the goal of LL is to update models based on recently encountered data. For this reason, in an LL setup, the dataset is split

into successive "time groups". Such groups are split according to the number of consecutive commits encountered, the group size (GS) being a hyperparameter of the setup. We opted to split based on number of commits instead of a fixed time period, since software engineering datasets have varying activity over time with some weeks having little to no data. Furthermore, in an LL setup, each model update needs to train on time groups with a decent amount of data in order not to overfit on new data and not to unlearn underrepresented classes' predictions because of sparse training sets.

4.2.3 Lifelong learning training process

Overall algorithm

The LL setup has two different types of time steps: one initialization step, followed by multiple update time steps. At each time step of the LL training process, the algorithm shifts up one time group of data (GS), then defines separate sub-datasets for the training, validation, and testing of the NN. Figure 4.1 shows an example of the first three time steps (named "Initialization", "Update1" and "Update2") of an LL setup with a sliding window validation set of size $VWin = 3$ (as an example).

A given time step's three sub-datasets never overlap amongst each other; we also avoid certain kinds of overlap across time steps. In particular, every data point that ever appeared in a validation set will never appear in a training set and vice-versa. This is achieved through an initial random sampling while respecting the class representations in each set. Note that the test set of a given time step by definition will overlap with the next step's training set. To illustrate, the data of Train1 will never overlap with Valid1 or Valid2, while the data of Valid1 and Valid2 can overlap.

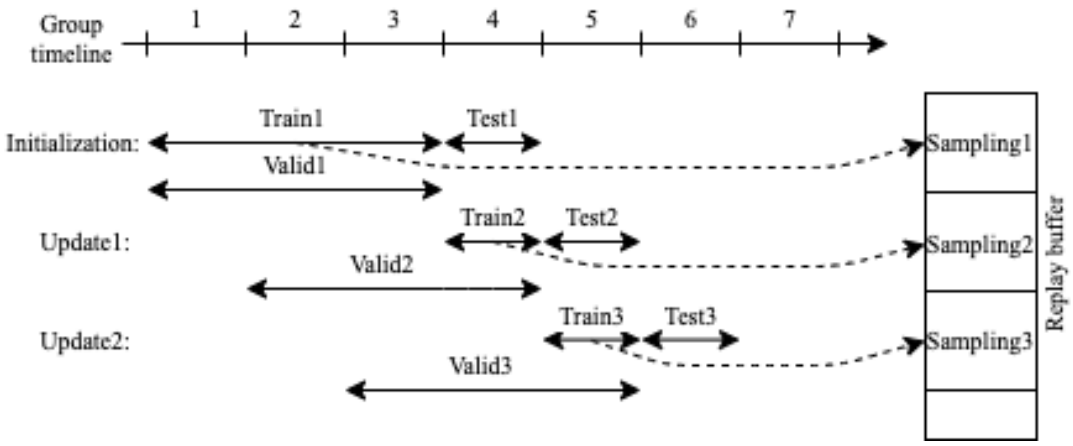


Figure 4.1: LL setup with validation set, initial training set and replay buffer of size $VWin = ITWin = RBWin = 3$, resp.

The validation sets are created with data points of a sliding window of size $VWin$, which is different from traditional LL setups where the whole timeline would be covered. This choice was made because we suspect projects to exhibit concept drift, hence progressively filtering out older time steps allows the model to eventually forget outdated knowledge.

However, in order to limit the catastrophic forgetting of the model, we define a replay buffer. This replay buffer is built as we go through time steps, by randomly selecting a portion of new training data points found since the last update, i.e., sampled from group t. That sample is balanced, i.e., the same number of data points is selected of each classification label, to have a significant representation of both labels at training time. This does not impact the evaluation as the validation and test set have the original data distribution. We need to keep the replay buffer's size limited in order to keep incremental retraining time manageable and to avoid old data from harming the model [1]. Hence, to ignore older data points, we use a First-In-First-Out (FIFO) approach with window size ($RBWin$) as hyperparameter.

Time step 1 (Initialization)

During the initialization step, the MLP is created and trained. We select an initial training window of size $ITWin$ groups, whose hyperparameters are tuned on a validation set of $VWin$ groups. Then, the trained model is deployed to predict each data point in the next group ($ITWin + 1$). During the initialization step, we can define the different sets as:

- Replay buffer: buffer = {}
- Training set: train = { points d sampled from group $i \mid i \in [1; ITWin]$ AND $d \notin valid$ } ;
- Validation set: valid = { points d sampled from group $i \mid i \in]ITWin - VWin; ITWin]$ AND $d \notin train$ };
- Test set: test = { points d in group $i \mid i = ITWin + 1$ }

After training, balanced random sampling is applied to the training set, and the sampled data is added to the replay buffer.

Subsequent time steps (Update)

Once the initial model has been deployed, the strength of LL starts to show up on subsequent time steps, when the MLP model will be updated incrementally. At a given time step t, the deployed model from $t-1$ is updated with the new data of time group t that has been labeled, as well as the replay buffer's current state, containing samples anterior to t. At the start of time step t, we can define the different sets as:

- New data set: new_data = { points d sampled in group $i \mid i = t$ AND $d \notin valid$ };

- Replay buffer: buffer = { points d sampled from group $i \mid i \in [t - RBWin; t[$ };
- Training set: train = New data set $\cup$ Replay buffer;
- Validation set: valid = { points d sampled from group $i \mid i \in]t - VWin; t]$ AND $d \notin train$} ;
- Test set: test = { points d in group $i \mid i = t + 1$}.

Again, the test set is not available at the time of retraining, but represents data encountered after deployment. Furthermore, the FIFO policy for the replay buffer is incorporated above by removing all data points from before group $t - RBWin$. At the end of the training phase, a balanced sampling is done on data group t and added to the replay buffer.

4.3 Empirical study methodology

4.3.1 Use cases

Our empirical study validates LL on two software analytics tools used in the CI pipelines of our industrial partner, Ubisoft, with historic data from 7 closed-source projects that each have been using one of the software analytics tools for 1.5 to 9 years. We also consider one open-source project that was used as a reference dataset when developing the brown build detection tool at Ubisoft, as the data distribution is similar to the closed-source ones. One tool performs brown build detection [1], while the other performs JIT risk prediction [16]. The features used in these tools are derived from build logs and version control metadata, respectively. Ubisoft has observed concept drift on those datasets through performance decay, as the studied projects, all related

to game development, follow development steps/gates spanning across the different development milestones over time. The changing nature of development activities along these gates motivates the need to update software analytics tools regularly.

Brown Build detection

Description. CI evaluates the quality and risk of new code changes submitted to the integration pipeline by executing builds and test jobs [44]. In theory, successful builds validate the changes, whereas failed builds indicate the need to rectify a code change before integrating it to the project. In practice, the difference between successful and failed builds is blurred by so-called "brown builds". A brown build (or flaky build) is defined as an intermittent build job failure that would change to success just by rerunning [54].

Previous approaches to predict whether a build is brown or safe are based on flaky test detection approaches [59,61,62] or based on build log vocabulary extraction [1,54]. Ubisoft developed a latter type of model. This approach takes as input build logs, processes them to extract the best (series of) words using KBest feature selection, then vectorizes each trace with TF-IDF. The vectors are then used as input for an XGBoost-based model.

Data extraction. Table 4.1 describes the projects for brown build detection used in this chapter. Our study focuses on three closed-source projects provided by our industrial partner and one open-source project (Brown_OSS). The Brown Failure Rate β refers to the percentage of brown job failures across all job failures, including reruns of builds. The studied projects were chosen to have at least 1 year of data with a consistent build job activity over time (at least 1k jobs per week). The projects

Table 4.1: Studied projects for Brown build and JIT risk prediction. The Brown failure rate represents the percentage of brown jobs among the failed build jobs and the risky rate the percentage of risky commits.

Project	**Period [month]**	**#jobs**	**#brown-jobs**	**Brown failure rate (β)**
Brown_1	18	23k	3k	30%
Brown_2	29	63k	8k	37%
Brown_3	17	22k	6k	5%
Brown_OSS	43	47k	1k	13%
Projects	**Period [years]**	**#commits**	**#risky- commits**	**Risky rate (RR)**
Risk_1	5	50k	12k	24%
Risk_2	4	49k	13k	27%
Risk_3	9	82k	20k	24%
Risk_4	5	57k	9k	15%

differ in languages, number of jobs, and in β. Project Brown_OSS corresponds to the Graphviz [132] project, a graph visualization project publicly available on GitLab. We include this project in our study since it has a non-trivial brown failure ratio of 13% that falls in between the other projects' brown failure rate.

For the four projects, the dataset extracted is composed of build logs and CI metadata such as the job's name, number, associated commit id, and timestamp information. This data is extracted from GitLab [133]. We also compute additional features about the number of past reruns of the job and the number of commits since the last brown job. These features are used to include information about the CI job's context and the moment it was run.

To reduce information contamination among build jobs from reruns, all the jobs allocated to a given commit and build job name appear in the same subset. This implies that despite the choice for a fixed number of *(commits,job name)* per group,

the size of the groups may vary depending on how many reruns were triggered.

Just-in-time risk prediction

Description. Just-In-Time (JIT) risk prediction is an approach where features related to a code change (such as the number of lines added/modified/removed, number of files touched, developer's experience, etc.) are used to detect code riskiness [16, 45]. Just-in-time refers to the fact that the detection is done at the time a code change is submitted for review, before the code is even built. Such a method has the advantage that the code change is still fresh in the developer's mind and current research subjects focus on making the prediction explainable, offering mitigation techniques to the developer to improve their code change [46].

In Ubisoft's implementation of this approach, features are fed to an XGBoost-based model [48]. The tool is currently used in almost all the projects of our industrial partner as a predictor for developers on the riskiness of their code changes or, in some cases, as an integration step.

Data extraction. We study 4 different closed-source projects of our industrial partner, see Table 4.1. A code change is labeled as risky if it is tagged as bug-introducing in a Jira ticket. In principle, Jira uses the SZZ approach [17] to derive bug introducing changes from bug fix data, but labels can also be manually overridden by developers, e.g., if they deem an SZZ blame to be incorrect.

The features used for the risk prediction, described in Table 4.2, involve metadata about code changes. The "approvers" are reviewers that can decide which code changes can be integrated into an upcoming release. All code changes are reviewed by at least one reviewer. When nearing a release date, the code changes are approved

Table 4.2: Features used for Just-In-Time risk prediction.

Name	Description
Subsystems	#Modified subsystems. A subsystem is a directory containing code files.
Files	#Modified, added or deleted files.
Lines added	#Added lines of code.
Lines deleted	#Deleted lines of code.
Lines total	#Modified lines of code (added and deleted).
Devs	#Developers who changed the modified files.
Age	Average age of the modified files.
Unique changes	#Unique changes.
Experience	Experience of the developer who submitted the code change. All types of experiences are measured in terms of contributions weighted over time $\frac{1}{(m-1)}$ where m is the #months since the contribution.
Reviewers	#Reviewers assigned to the code change.
Reviewer experience	#commits the reviewer has reviewed regarding the targeted files.
Reviewer dev. experience	#commits the reviewer has developed regarding the targeted files.
Reviewer subsystem exper.	#commits the reviewer has reviewed regarding the targeted files' subsystems.
Approvers	#Approvers assigned to the code change.
Approver experience	#commits the approver has reviewed regarding the targeted files.
Approver dev. experience	#commits the approver has developed regarding the targeted files.
Approver subsystem exper.	#commits the approver has reviewed regarding the targeted files' subsystems.
FilesRiskiness	Historical ratio of bug introducing changes in the modified files.
Subsystems-Riskiness	Historical ratio of bug introducing changes in the modified subsystems.
Days to release	#Days to the release date of next update of the software product.

by an approver to be integrated in the upcoming release (or pushed to the next release).

4.3.2 Baselines

The metric used to evaluate and optimize the performance of the approaches for both use cases is F1-score ($F1=\frac{2}{Pre^{-1}+Rec^{-1}}$). The F1-score is the harmonic mean of the precision ($Pre=\frac{TP}{TP+FP}$) and recall ($Rec=\frac{TP}{TP+FN}$). We also report the G-Mean ($GM=\sqrt{Rec \times Rec_{safe}}$), with $Rec_{safe}=\frac{TN}{TN+FP}$, also called specificity. We chose to use the latter two metrics because of the unbalanced nature of our datasets in terms of β and RR in Tables 4.1.

As baseline, we use the 8 setups of Table 4.3. For both use cases, there is a naive

Table 4.3: Training setup for LL and baselines.

	Training setup	Use case	Description
Baselines	Weekly XGB	Brown	RFS setup on XGBoost model with training window retrained every week [1] (training set ∼.5-1k jobs).
	Heuri XGB	Brown	RFS setup on XGBoost model with a training schedule based on performance decay [1] (training set ∼.5-1k jobs).
	Real Pred	Risk	Industrial collaborator-provided predictions of RFS setup on XGBoost model (retrained every 6-8 weeks, training set∼all data available).
	RFS MLP	Brown & Risk	RFS setup on MLP with all the available labeled data from the start of the dataset to the training point, retrained on a group-based schedule (every GS).
LL setups	LL noRB	Brown & Risk	LL setup on MLP, retrained on a group-based schedule (every GS) without replay buffer.
	LL	Brown & Risk	LL setup on MLP, retrained on a group-based schedule (every GS) with replay buffer.

AlwaysBrown/Risky baseline corresponding to always predicting builds/commits to be brown/risky. Those naive baselines are used as context regarding the expected performance in comparison to β and RR. For brown build analysis, we also include 2 RFS baselines proposed by Olewicki et al. [1], respectively RFS with weekly scheduling (Weekly XGB) or scheduling based on model performance decay (Heuri XGB). For JIT risk prediction, the ML architecture was not revealed to us, apart from learning that it is XGBoost-based and retrained from scratch every 6-8 weeks, using the full data history. Nevertheless, we did receive the historical prediction results of the actual model (RealPred), a highly relevant baseline consisting of the performance of the real (RFS) model.

Finally, for both use cases, we include a baseline retraining setup for MLP models using RFS (RFS MLP). This baseline is representative of state-of-the-practice RFS setups and setups relying on the evolutive sampling of the training set [25]. We schedule the retraining of the RFS MLP baseline at every group size, similar to how the LL retraining of our approach is scheduled.

4.4 RQ1: How does an LL setup perform compared to an RFS setup for software classification tasks?

4.4.1 Motivation

In this question, we empirically compare the performance of the LL setup with RFS approaches previously used by our industrial partner and with an NN-based RFS approach. We thus address the two following subquestions:

- RQ1.1: How beneficial is an NN-based LL compared to state-of-the-practice RFS approaches?
- RQ1.2: To what extent does an NN-based LL setup improve the performance of an NN-based RFS setup?

4.4.2 Approach

For each dataset, we used the MLP architecture and LL setup hyper-parameters with the best median F1-score performance out of 5 runs on the validation sets. For those best setups, Table 4.4, row "LL", shows the average prediction results out of 5 runs on the test sets, which we will compare with the RFS setup baselines. The tuned LL setup hyper-parameters are reported in Table 4.5. In RQ1.1, we use the Weekly XGB and Heuri XGB baselines for brown build prediction, and Real Pred for risk prediction. In RQ1.2, we use the RFS MLP baseline for both use cases. The latter baseline uses the same MLP model and scheduling approach as the LL setup, yet using different training and data selection approaches, as mentioned in Table 4.3 (RFS MLP). To assess the significance of the performance differences, we use, for each project, a Kruskal-Wallis statistical test ($\alpha = 0.05$) to compare the F1-score and

Table 4.4: Performance after hyper-parameter optimization for the LL setups and baselines. We report the average test results out of 5 runs. The **bold blue** values indicate the best F1-score (F1) and G-mean (GM) per project. For each project, Kruskal-Wallis tests ($\alpha = .05$) between the F1-score/G-mean of runs of the LL setup and of the other setups are performed. If significant, the Cliff's delta effect size is shown as "+" (large effect), otherwise "/" is shown.

Model	Project Brown_1					Project Brown_2				
	F1	GM	Brown Pre	Brown Rec	Safe Rec	F1	GM	Brown Pre	Brown Rec	Safe Rec
Weekly XGB	56 +	66 +	59	54	81	67 /	69 +	75	61	79
Heuri XGB	60 +	71 +	61	60	83	67 /	70 +	73	62	79
RFS MLP	64 +	69 /	53	80	59	70 /	70 /	68	72	69
LL noRB	66 /	70 /	54	83	58	69 /	67 /	64	74	62
LL	**72**	**77**	64	83	72	**71**	**71**	68	74	68

Model	Project Brown_3					Project Brown_OSS				
	F1	GM	Brown Pre	Brown Rec	Safe Rec	F1	GM	Brown Pre	Brown Rec	Safe Rec
Weekly XGB	22 +	37 +	45	15	95	44 +	66 +	44	45	98
Heuri XGB	27 +	43 +	50	19	96	44 +	66 +	44	45	98
RFS MLP	**36** +	**62** +	28	51	75	28 +	68 +	19	55	84
LL noRB	30 /	57 /	21	52	63	49 +	81 +	33	100	65
LL	31	57	23	47	69	**75**	**94**	60	100	89

Model	Project Risk_1					Project Risk_2				
	F1	GM	Risky Pre	Risky Rec	Safe Rec	F1	GM	Risky Pre	Risky Rec	Safe Rec
Real Pred	25 +	38 +	74	15	98	35 +	48 /	66	24	95
RFS MLP	**57** /	**70** /	51	65	76	48 /	54 /	36	74	40
LL noRB	52 +	66 +	44	64	69	50 /	**64** /	40	67	62
LL	56	69	49	64	75	**52**	59	39	77	46

Model	Project Risk_3					Project Risk_4				
	F1	GM	Risky Pre	Risky Rec	Safe Rec	F1	GM	Risky Pre	Risky Rec	Safe Rec
Real Pred	59 /	**76** +	45	88	66	42 +	**71** /	29	74	69
RFS MLP	59 /	67 /	47	78	58	47 +	66 +	38	61	72
LL noRB	**60** /	69 /	50	75	64	45 +	66 +	34	69	64
LL	59	66	45	85	52	**52**	**71**	42	67	74

G-mean of the different runs of the LL setup with the other setups.

4.4.3 Results

Results for RQ1.1: How beneficial is an NN-based LL compared to state-of-the-practice RFS approaches?

The LL setup has an F1-score significantly higher (with large effect)

Table 4.5: Optimal LL hyperparameter values used in our study.

Name	Project's tuned hyperparameters							
	Brown_				Risk_			
	1	2	3	oss	1	2	3	4
Group size GS	50	50	50	100	500	500	1000	500
Initial train window ITW	15	10	10	10	15	15	15	15
Validation window $VWin$	15	10	10	10	8	15	8	8
Replay Buffer sample size [%GS] $RBSize$	10	5	5	20	10	10	10	10
Replay buffer window $RBWin$	8	10	5	10	8	15	8	8

than an RFS setup in 6 projects (+9-31%), while performance is similar in the 2 other projects (+0-4%). For brown build prediction, when comparing the LL performance with the weekly retraining of an XGBoost model ("Weekly XGB" in Table 4.4), the F1-score of the LL setup is significantly higher (+9-31%) for all projects but Brown_2. Brown_2's F1-score of 71% still shows a small +4% increase. The comparison to heuristics-based retraining (row "Heuri XGB" line in Table 4.4) is very similar, again with statistical improvements for all projects but Brown_2.

For JIT risk prediction, we compare the LL setup with our industrial partner's historical model performance ("Real Pred" in Table 4.4). The F1-score of the LL setup is significantly higher with large effect size (+10-31%) for three projects and identical for Risk_3. While Risk_1 and 2 show a decrease in precision (-25% and -27%) compensated by an increase in recall (+49% and +43%), Risk_3 and 4 show the opposite, with respectively no improvement and improvement (+13%) in precision and a reduction in recall (-3% and -7%).

We can see that, overall, the observations with G-mean are the same as those of the F1-score for all the projects but two, with Risk_3's real prediction performing significantly better than the LL setup and Risk_4's real prediction behaving similarly. Note that we tuned the hyper-parameters based on the F1-score, which might explain

these observations.

RQ1.1: The LL setup performs at least similarly to the state-of-the-practice baselines for all datasets and significantly better (large effect) in 6/8 of the cases.

Results for RQ1.2: To what extent does an NN-based LL setup improve the performance of an NN-based RFS setup?

3/8 datasets show a significant improvement in F1-score (+5-8%) for the LL setup compared to an RFS MLP, 4/8 show no significant difference (-1% to +8%), and 1/8 shows a significant decrease (-5%). In Table 4.4, we observe that using an LL setup significantly improves the F1-score in comparison to RFS MLP for Brown_1 (+8%), Brown_OSS (+5%), and Risk_4 (+5%). In contrast, Brown_3 shows a significant decrease in F1-score (-5% significant). In the other cases, LL performed at least as well as RFS MLP. Regarding the G-mean, the same observations can be made apart from Brown_1 not having a significant improvement, still achieving a +8% increase.

In the case of Brown_3 (where LL performed significantly lower with large effect size), we noticed a decrease in precision (-5%) and recall (-4%) for the LL setup. This shows that in this case, seeing all of the past data actually improved the performance, which we suspect is due to the highly unbalanced nature of that dataset (β of 5%), causing LL to overtrain on the few brown jobs, while RFS has access to all historical jobs. On the other hand, Brown_OSS show huge improvement (+47%) in F1-score between the RFS and LL, which indicates that the model trained on mostly recent data points improved the performance and forgetting past data points improves the

model. Two of the four projects with equal performance have a heavily cyclic development process of 3-month sprints; future empirical analysis should investigate the potential link between these process characteristics and the model results.

Overall, RQ1(.2) shows that the LL setup is able to match and in most studied cases even outperform the performance of previously used approaches. Extending the LL setup with an online bagging approach [22, 84] would likely improve the performance even further [85], though also increasing the computational effort needed to train the multiple models in the bag.

RQ1.2: The LL setup performs at least similarly to NN-based RFS in 7/8 cases and is outperformed in 1 case. Hence, LL can bring value in terms of model performance, but is not a panacea.

4.5 RQ2: How essential is the use of a memory-based LL setup?

4.5.1 Motivation

LL setups are often at risk of showing symptoms of catastrophic forgetting, gradually decreasing the performance of LL retraining. To reduce this risk, we adopt a replay buffer (memory-based approach). In this question, we aim at evaluating how impactful this approach is on LL.

4.5.2 Approach

We compare the LL setup with(out) replay buffer, corresponding respectively to rows "LL"/"LL noRB" in Table 4.4.

4.5.3 Results

Using a replay buffer significantly improved the F1-score for 3 projects (+4% to +26%), without significant change in performance for the other projects (-1% to +6%). In Table 4.4, we observe that using a replay buffer significantly improves the F1-score (with large effect size) for Brown_OSS (+26%), Risk_1 (+4%), and Risk_4 (+7%). While Risk_3 sees an F1-score decrease by 1%, the difference in performance is not significant. The G-mean observations are similar to those of the F1-score.

We thus observe less improvement than Gao et al., whose replay approach improved results in all cases [23]. However, their NLP models take as input text, while our approaches respectively take pre-embedded input (brown builds) or feature-based inputs (risk prediction), both of which might capture less change over time than text does due to the abstraction of those inputs.

RQ2: The use of a memory-based approach (replay buffer) to reduce catastrophic forgetting significantly improves the performance for 3 out of 8 projects, while maintaining performance for the others.

4.6 RQ3: How time-efficient is an LL setup compared to an RFS setup?

4.6.1 Motivation

The computational effort of model updates in an LL setup is lower than RFS since less data is considered. However, depending on the retraining frequency in both setups, the combined computational effort across all model updates might outweigh the single model retraining computation of RFS. This might make LL less tempting

Table 4.6: Median life expectation of models in #days.

Life Exp [day]	Brown_1	Brown_2	Brown_3	Brown_OSS
Weekly XGB	7	7	7	7
Heuri XGB	42	35	14	7
LL (med.)	9	4	5	18
LL (avg.)	12±3	12±9	6±2	85±99
Life Exp [day]	Risk_1	Risk_2	Risk_3	Risk_4
Real Pred	49±7	49±7	49±7	49±7
LL (med.)	32	22	35	36
LL (avg.)	40±11	32±6	43±6	32±13

to adopt in the industry, especially in cases where LL did not outperform the RFS setups. Hence, in this RQ, we evaluate how often updates are necessary as well as approximate the actual computational effort of update/model training.

4.6.2 Approach

We model the overall computational effort for both retraining setups in terms of the life expectation of models and the complexity of the retraining algorithms [134]. First, the life expectation (LifeExp) of a model corresponds to the time between successive model updates based on the maximum timestamps of training and validation data points for each training. For each project and a given retraining setup, Table 4.6 shows the median life expectation across all retraining attempts. Secondly, as done by Thuijsman et al. [135], we calculate the algorithms' complexity in terms of the size of the training set $TrainS$. As shown in Algorithm 1, the update/training time of our MLP models is linearly dependent on the size of the training set, as is the complexity of the XGBoost training algorithm [125]. In other words, we can compare the computational effort of RFS and LL retraining setups by comparing the training data sizes.

To combine both measures into one measure of computational effort, we define the effort coefficient $coef = \frac{trainS}{lifeExp}$ and the relative coefficient $coef_{rel} = \frac{coef}{coef[LL]}$, relative to the coefficient of the LL setup. The higher the $coef_{rel}$ for a given training setup, the more effort it requires compared to LL. In particular, in RQ2, we compare the "RFS MLP" and "LL" setups, as they both train the same type of MLP and just differ in the update computational effort.

4.6.3 Results

The median life expectation of models with an LL setup is between 1.1-8.75 times shorter than the RFS MLP baseline for 7 of the studied projects and is 2.3 times longer for Brown_OSS. Table 4.6 shows how, in the case of brown build prediction, the median life expectation of LL models for Brown_1, 2 and 3 is 2.8-8.75 times shorter than for Heuri XGB, needing thus more updates for similar to significantly better model performance (cf. RQ1). On the other hand, for Brown_OSS, the median life expectation of models is 2.3 times longer than for both XGB-based baselines because that project has much slower brown build activity, if we refer to the total age of the project and the amount of brown jobs from Table 4.1. Since less frequent brown jobs are encountered, training more often (with smaller GS) would most likely overfit on the lower amount of brown jobs present in the training set.

For JIT risk prediction, the LL model needs updates every .6-1 month as a median, whereas the actual RFS is done every 1.5-2 months by our industrial partner (Real Pred), yielding thus a median life expectation 1.1-2.2 times shorter for the LL setup.

The LL updates use about 3.3-13.7 times less data during updates than

Table 4.7: Size of training sets for model training/updates in terms of GS for all datasets, and average model computation time across 5 runs for the Risk datasets (training with 50 epochs and 10 sample batches).

Dataset size	Brown_1	Brown_2	Brown_3	Brown_OSS
RFS XGB (med)	24.6GS	11.4GS	10.5GS	6.6GS
LL initializat.	15GS	10GS	10GS	10GS
LL updates	1.8GS	1.5GS	1.25GS	2GS
LL Speedup	8.3×	6.7×	8.×	5.×
Dataset size	Risk_1	Risk_2	Risk_3	Risk_OSS
LL initializat.	15GS	15GS	15GS	15GS
LL updates	1.8GS	2.5GS	1.8GS	1.8GS
LL Speedup	8.3×	6.×	8.3×	8.3×
Avg. time	Risk_1	Risk_2	Risk_3	Risk_OSS
LL initializat.	38.6 s	121.4 s	91.9 s	32.7 s
LL updates	10.4 s	12.7 s	11.4 s	9. s
LL Speedup	3.7×	9.6×	8.1×	3.6×

the RFS approach. Similar to Olewicki et al. [1], we found that the RFS approach for brown build detection requires an optimal training dataset size of 30 weeks for all studied projects. To put this in context, we express this in terms of the median GS (in terms of number of commits) of data groups used by LL in Table 4.7 (row "RFS XGB (med)"). The LL initialization step (which is equivalent to RFS) uses a training set of ITW time groups (see Table 4.5), with each group containing GS data points. We consider the resulting $\sim$ 10-15 GS data points as a lower bound for the dataset size in an RFS MLP setup. Later model updates use a training set composed of both new data points since the last update and a replay buffer, amounting to $GS + \frac{RBSSize}{100} \times RBWin \times GS$ data points.

When comparing in Table 4.7 the training set sizes of the LL updates to RFS XGB for brown build detection , the dataset of LL updates is 3.3-13.7x smaller (e.g., 24.6/1.8=13.7 for Brown_1). Then, when comparing LL updates with LL initialization

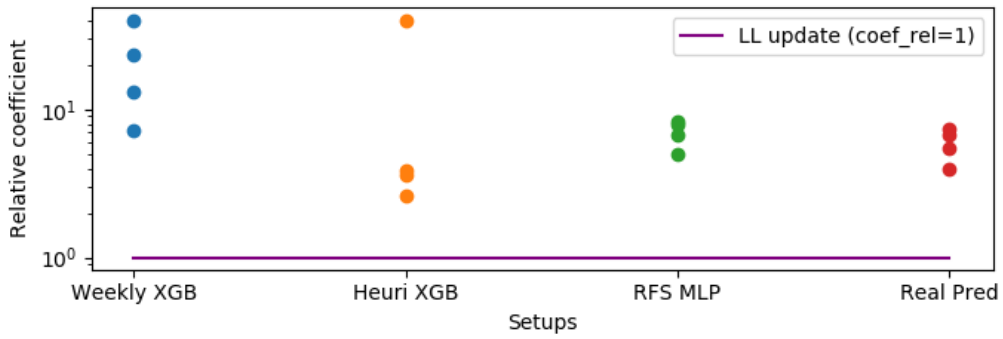


Figure 4.2: Relative coefficient $coef_{rel}$ of training setups compared to the LL setup updates.

(row "LL Speed up"), this time on all 8 projects, we find that the dataset of LL updates is 5-8.3x smaller than the LL initialization.

Figure 4.2 shows that the relative effort coefficient $coef_{rel}$ for all the setups is between 2-40, demonstrating the overall lower computational effort of the LL setup. With alternative state-of-the-art approaches, such as online bagging [22], the computation effort would be multiplied by the number of models in the bag. There is thus a trade-off to be made between improving the performance with bagging, while increasing the computational retraining effort.

The actual retraining time speedup matches the theoretical one for Risk_2 and 3 datasets, and is lower than expected for the other datasets, although still showing at least a 3.6-fold speedup compared to RFS updates. Table 4.7 shows the actual mean computation time for initialization and updates of the MLP model in an LL setup for the Risk datasets. The LL Speedup ratio ($\frac{initializat.}{updates}$) is given to compare LL to RFS computational effort, as each RFS retraining takes a similar time as the LL initialization time. In half of the projects,

the speedup in terms of time matches the theoretically expected speedup (linear dependency on data file size), while the LL time efficiency, though lower than expected, is still at least 3.6x as high for the others. The reason for the lower speedup is the non-optimized memory management in the Python libraries causing extra time consumption in the training steps, independent of the size of the data.

RQ3: The effort of maintaining an LL setup is 2-40x less computationally expensive than an RFS setup.

4.7 Discussion

While software analytics tools are being adopted in industry to ease the work for developers, the design and deployment of such tools do not take into account the evolving nature of software projects, instead relying on stationary ML models (and RFS). Through our RQs, we observe that the performance of an LL setup can match, if not improve, that of an RFS setup, while considerably reducing the computational effort of updating the model by a coefficient $coef_rel$ of 2-40. The speedup of individual model updates of the LL setup compared to RFS reduces energy consumption but also allows more frequent updates to keep the models up-to-date, addressing drift.

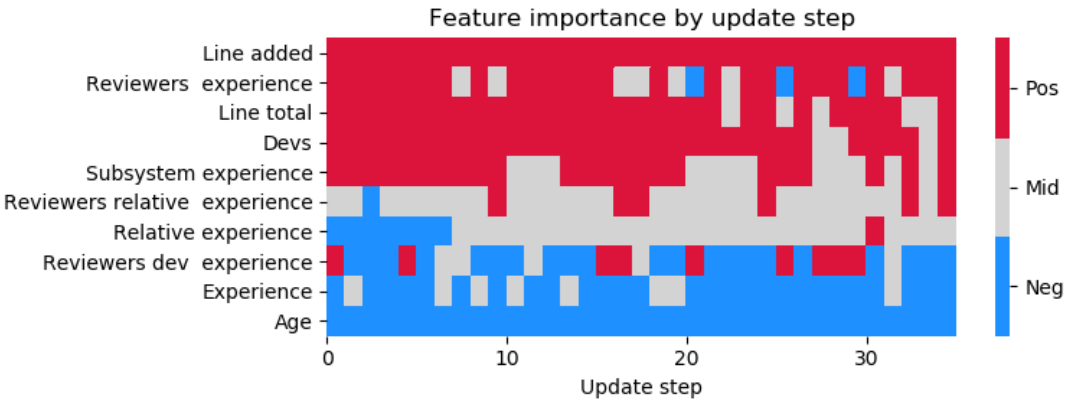
One aspect of software analytics models that was ignored until now is the human interpretation of said models, as an updated model might convey a contradictory message to developers compared to prior versions, for example when the age of a file at one time step would correlate with higher risk, yet at the next step would correlate with lower risk.

Hence, to analyze the relation of LL to the potential cognitive effort of changing

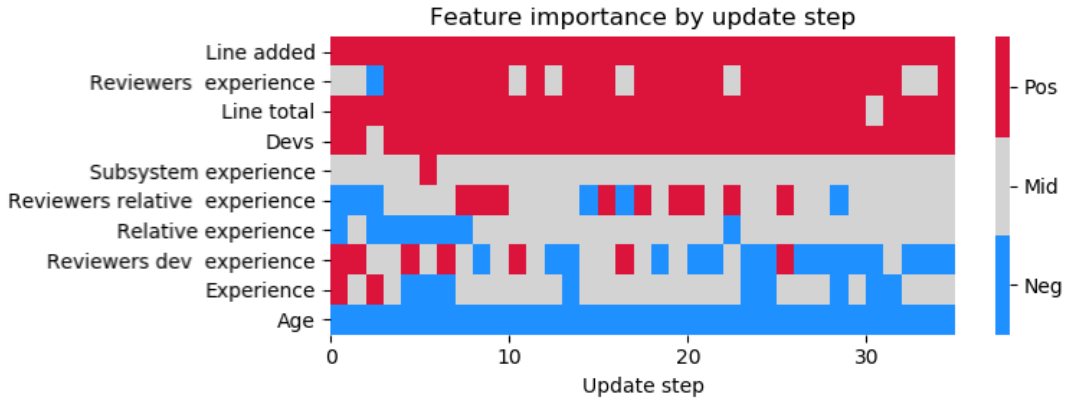
model interpretation, we evaluate how the feature importance of models in LL and RFS setups evolves over time using the `captum` [114] Python package. Feature importance is a metric that shows the strength of the learned correlation of a feature with the prediction [114]. Figure 4.3 shows how it evolves in the LL and RFS setups for the 10 features with the highest sum of absolute average and variance of feature importance. To better observe how feature importance can change from one extreme to another, we discretize the feature importance values. Positive importance values are colored in red ($> .1 \times Max$), negative importance in blue ($< -.1 \times Max$) and intermediate ("Mid") in gray, where Max is the maximum absolute value of feature importance at a given time step for a given project.

From Figure 4.3, we observe that sometimes the LL setup's feature importance varies less over time than the RFS setup's (e.g., Reviewers relative experience), while at other times we see the opposite (e.g., Reviewers experience). In order to quantify this observation, Table 4.8 provides the percentage of features that have a larger feature importance variance, calculated as $\frac{1}{|F|}\sum_{i \in F}(f_i - mean(F))^2$, with F the vector of feature importance across time steps. The LL setup has +6% to +58% more features with higher variation than RFS for Risk_1&2&3, while 40% less for Risk_4. However, when looking at the discretized feature importance values (using +.1, 0. and -.1 values f_i for Pos, Mid, Neg, respectively), only Risk_1 has a higher percentage of features varying more in the LL setup (+6%), while the other projects show the opposite (from +24% to +52% for RFS). This confirms our observations of LL vs. RFS in Figure 4.3.

In other words, either setup could actually lead to more stable or flexible models in terms of feature importance. The LL setup gains stability because all updates



(a) LL setup for Risk_1



(b) RFS setup for Risk_1

Figure 4.3: Discretized feature importance evolution for Risk_1.

incrementally derive from one initial model, while it gains flexibility from seeing a smaller amount of relatively recent data (new data and the replay buffer). On the other hand, the RFS setup gains flexibility from having a new model retrained from scratch at each step, but gains in stability from training on the complete dataset available at each training point.

The fact that the discretized analysis of feature importance variance shows more stability for LL relative to RFS than the non-discretized analysis indicates that many

Table 4.8: %Features with higher variance of importance for RFS or LL compared to the other for the Risk use case. The variance is computed on the feature importance value before discretization (first 3 rows) and after (last 3 rows).

[%]	Risk_1	Risk_2	Risk_3	Risk_4
for RFS (not discretized)	21	21	36	58
for LL (not discretized)	79	58	42	18
Same (not discretized)	0	21	21	24
for RFS (discretized)	30	48	58	36
for LL (discretized)	36	9	6	12
Same (discretized)	33	42	36	52

of the fluctuations in feature importance for LL occur within discretized categories (Pos, Neg or Mid), corresponding to minor changes in model interpretation, while major changes crossing borders between Pos/Mid/Neg are less common than for RFS. Hence, models in LL setups seem to have a relatively good continuity. Then, whether this continuity is "correct" or if the LL setup misses important feature importance changes, the fact that the LL setup often showed better F1-score and/or G-mean performance than the RFS setup hints at the former.

4.8 Threats to Validity

External validity. We evaluate LL setups in an industrial context regarding performance and required maintenance effort, for two use cases brown build detection and JIT risk prediction, as exemplars of software analytics classification models used in industry. Overall, the LL approach was applied to eight projects, among which seven from our industry partner, Ubisoft. More projects and analytics tools should be studied to validate the generalization of our results to the wider domain of software analytics.

Construct validity. With the current LL setup, the feature selection step of

the brown build detection tool is only done at the initialization step. This means that if new features become relevant over time, those will not be considered by the model. Future research could improve on this by using, for instance, flexibly-sized MLP models that would allow LL updates while adding new nodes when needed (and thus adapt to changing inputs) [95]. This is not an issue for JIT risk prediction, which has a fixed set of features.

We evaluated the computational effort of LL computed to RFS in RQ3, considering single MLP model architectures. It can be noted that in the case of using bagging [84], the computational effort would be multiplied by the size of the bag for both setups.

We used the time ordering of commits for data splitting, but our analyses did not take into account the delay between a prediction and the availability of its true JIT risk label. However, since we update the model every GS data points, many new data points would have had time to be correctly labeled in the meantime. Also, the JIT models yielded similar improvements compared to the baseline as our Brown build detection models, for which no delay in labeling occurs (since humans evaluated predicted risk cases immediately). This suggests that the impact of the delay issue might be limited.

Our approach was optimized regarding the F1-score metric, which considers precision and recall as equally important. Though the precision or recall could be improved by changing the threshold of decision, chosen to compute the F1-score. We used a default 50% threshold. By increasing the threshold, the precision would increase. As precision and recall are connected, this increase would also lead to a decrease in recall and vice-versa.

4.9 Conclusion

More and more ML-based software analytics tools are used in industry as a practical way of automating software engineering tasks. However, in practice, as a model trained at the beginning of the development process might not be relevant later on due to drift, these ML models are regularly being discarded in lieu of new models retrained from scratch. Ubisoft, our industrial collaborator, identified this issue for their software analytics tools. By using an LL setup, the same model could be updated incrementally based on the previously pre-trained model and only a limited historical dataset necessary to avoid catastrophic forgetting. By evaluating the LL setup on eight projects, seven industrial and one open-source, across two software analytics use cases, we observe that LL performs at least as well as RFS in most of the cases and often outperforms it in practice. Also, in contrast to recent work, we find that a replay buffer is not always necessary to avoid catastrophic forgetting in our use cases, though improving performance significantly in 3/8 cases.

More importantly, we observe an important gain with LL setups as they are computationally less expensive to maintain. We find that the LL setup leads to a considerable speed-up of the training process as updates use 3.3-13.7 times less data than RFS setups. This enables LL retraining to be performed more frequently without increasing the total computational effort of retraining, effectively making LL both feasible and valuable in practice.

Finally, we observe that LL setups have more stable predictions as the models are updated on top of past models, having a better continuity than RFS regarding feature importance. This reduces the cognitive effort of changing model interpretation for users.

This chapter was published as a conference paper titled "On the Costs and Benefits of Adopting Lifelong Learning for Software Analytics - Empirical Study on Brown Build and Risk Prediction" [2] at the IEEE/ACM International Conference on Software Engineering - Software Engineering in Practice (ICSE - SEIP) 2024, in collaboration with my co-authors, Sarra Habchi (Ubisoft), Mathieu Nayrolles (Ubisoft), Mojtaba Faramarzi (Université de Montréal), Sarath Chandar (Polytechnique Montréal), Brams Adams (Queen's University). The paper received a distinguished paper award.

4.10 Future work

Future work on this subject could apply active replay buffer selection to have a more representative set of past examples when doing updates. Furthermore, we considered traditional MLP models: it should be noted that the bigger the model, the more it would benefit from reduced computational effort. With the rise of Large Language Models in the last few years, such a lifelong learning setup should be considered for software tasks in evolving environments like CI, including generation tasks (e.g., code/fix generation, comment generation), This could extend our work from the automated gate to the manual gate use cases, and lead to evaluating Lifelong Learning approaches on more advanced machine learning architectures. Furthermore, future work should explore the comparison between the considered replay buffer with other catastrophic forgetting mitigation approaches such as regularization-based approaches (e.g., elastic weight consolidation [95], Riemannian Walk [136], and Synaptic Intelligence [137]).

However, another aspect of LLMs that we want to explore is their pre-trained characteristics. Pre-trained models can be used off-the-shelf or coupled with few-shot

techniques to address software use cases without needing a training phase [105]. LLMs can be used to address partial tasks, such as embedding for classification tasks [138], which we explore in Study 3, or entire tasks [139, 140], such as review comment generation which we explore in Study 4.

Chapter 5

STUDY 3: Hot-spot prediction-based file ordering for code review

5.1 Introduction

In software development, code reviewing is an essential manual quality assurance step during code integration [141, 142]. Developers submit their code for review as a patch. Then, reviewers either validate the code change, which is then forwarded to integration, or they provide feedback. The review process can go through several iterations, where the reviewer comments on the submission to have the developer revise it and submit a new version, referred to as a revision. Eventually, if the developer addresses all comments, the patch is accepted, otherwise it is rejected.

Given the essential role of code reviews in modern development [143], it is crucial to have a high code review velocity and effectiveness. Yet, Hong et al. report that developers waited a median of 15-64 hours for reviewers' feedback in three open source projects (i.e., Openstack Nova, Openstack Ironic and Qt Base) [13]. Those numbers were confirmed by our industrial partner, which experienced similar waiting times. Such long waiting times delay the integration of code and thus the projects'

development and release time. Furthermore, other factors can cause review processes to take a long time, including the reviewers involved (e.g., availability and experience), the submitted code (e.g., patch size and complexity, developers' experience) and the number of review iterations needed [8]. Reviewers are usually developers on the project, who have to split their work schedule between their own development tasks and reviews. Furthermore, they might have to review many patches themselves composed of many files and changes.

In order to improve review effectiveness, Hong et al. [13] automate the identification of the parts of patches that either need revisions or comments using a Bag-of-Words (BoW) embedding and two review process features (i.e., number of lines added and removed). As BoW embeddings do not represent the order of words such as Large Language Model (LLM) embeddings preferred in recent NLP-based tasks [144] do, we suspect that the former do not capture enough context. With approaches such as risk/bug prediction using software process features [16, 45, 66], it is also unknown how that type of embedding performs in the context of review activity prediction.

Furthermore, while all patch revisions are assumed to map to a corresponding review comment, revisions could also be triggered by a review comment on a different file or patch, or through communication external to the reviewing platform (e.g., in-person), yielding an incomplete ground truth. Hence, to be comprehensive, automation should not only target problematic parts of a patch requiring either a comment or a revision, but the union of these, which we refer to as **hot-spots**. Prediction of review activity needs has potential to assist reviewers through multiple use cases, such as adding warnings in the code for reviewers to focus on [42] or re-ordering files [145, 146].

In this chapter, we focused on the latter. In a wizard-of-oz user study with 29 professional developers, we find that, inspired by Fregnan et al.'s suggestion to "display first those files that are more critical" [147], hot-spot based file re-ordering correlates to improved review comment quality. In particular, more review comments were provided (+23%), and those comments were better targeted towards known hot-spots (+13% precision and +8% recall) compared to the alphanumeric ordering. Even though reviewers would still be required to look at the rest of a patch, automated identification of hot-spots would attract the reviewer's attention earlier to the most critical parts.

Hence, in order to improve the file-level prediction of three review tasks (i.e., need for comment, need for review, is an hot-spot) in terms of prediction performance and of the computation cost of embeddings and model training, this chapter studies the impact of text embeddings and of review process feature-based embeddings. We focus on file-level prediction due to its natural fit with the use case of prioritizing files in a submitted patch. Our empirical evaluation on three large open-source datasets and two closed-source datasets addresses the following research questions:

RQ1: Which text embeddings of code diffs perform best for review activity prediction? Although none of the specific text embedding stands out as the best, we found that applying any (or both) of our text embedding modifications, i.e., including removed lines as an additional context embedding or using a Large Language Model embedding instead of the Bag-of-Word approach, improved the performances in practice. In practice, our approaches report higher AUC in 13 out of 15 cases (median of 68-78%, improvement up to +5%) and higher F1-score in 10 out of 15 cases (median of 37-57%, improvement up to +4%) than the baseline.

RQ2: Which review process features perform best for review activity prediction? Closed-source datasets show no significant differences between the best text embedding and review process feature embedding (median F1-score 37-56%, with an improvement in practice of +2 to +8%), thus matching performance while reducing the embedding dimensionality. Moreover, for open-source datasets, feature-based embeddings perform similarly to better than the best text embedding in 5 out of 9 cases (median F1-score of 50-58%, improvement up to +4%) and lower in the other 4 cases (median F1-score 36-56%, lower by -2 to -15%). This shows that text embeddings have value as they capture relevant code/text information, not represented in the feature-based embeddings. We also find that the size-based features are the most important features for all datasets, as they bring information about the changed file (e.g., size, number of changes) but also context about the full patch size. The history-based features, though less important, also improve performance.

RQ3: How does the combination of review process features with text embeddings perform for review activity prediction? Compared to the state-of-the-art, our proposal of combining text embedding with review process features improved significantly the F1-score (median of 40-62%) for all datasets and all tasks, improving the results by up to +9%.

Finally, through an additional empirical analysis we confirm the user study's promise of hot-spot based file re-ordering, as our model predictions push +9-50% more hot-spots forward into the first half/quarter of file patches shown to reviewers compared to alphanumerical ordering.

This chapter's contributions are:

- A user study evaluating hot-spot-based file re-ordering;

- An empirical study of review activity prediction per-project and at file-level on 5 large datasets (including 2 closed-sources);
- 3 types of review activity labels: revised, commented, and hot-spot (either of the prior);
- A comparison of text embeddings from a Large Language Model and Bag-of-Words embedding, as well as the inclusion of only added lines or also including removed lines;
- Analysis of lightweight review process features;
- A replication package with the open-source dataset [148].

5.2 Motivational Use Case

By default, most review environments list the changed files of a patch in alphanumeric order, even though previous work argues that this might be suboptimal, especially for long code submissions [43,67]. Yet, it is common, especially in closed-source datasets, to have many file changes in a patch, making the review process more difficult and time consuming. Case in point, across the five datasets studied in section 5, we observed that 38-68% of patches are composed of more than 2 files, while in the two closed-source datasets, 25-30% of patches have more than 5 files.

Baum et al. established principles for code change ordering to help reviewers become more effective [31], while Li et al. [32] and Hong et al. [13] predict submission (patch) parts that need either review comments or a revision, yet neither of them explore or discuss the use of code diff quality as a way to prioritize the code change order. Fregnan et al.'s user study indicated that the earlier a file is shown in a patch,

the more likely it is to be commented on [147], hence suggesting to prioritize files that need review activity to the beginning of a patch.

In this motivation section, we empirically evaluate the impact of changed file reordering on the review process, through a user study that assumes the availability of models able to perfectly predict review hot-spots (i.e., any file requiring a comment or revision). If successful, this motivates the need for improving current review activity prediction models.

5.2.1 User study set-up

We carried out a wizard-of-oz user study [149] to compare hot-spot-based ordering of a patch's changed files to the typical alphanumeric ordering. The wizard-of-oz design means that we used knowledge about the actual hot-spots in the oracle (obtained similarly as in Section 5.3.1), which is useful to evaluate the promise of hot-spot prioritization for file reordering in optimal conditions compared to the alphanumeric ordering used by most review environments [147]. The user study was held on the premises of our industrial partner, and involved 29 developers with an average experience at the company of 6 years, knowledgeable both in developing and reviewing for the studied closed-source project, which is actively developed by more than 800 developers, with 3-4k patches per month.

We structure the study as an A/B study according to which each participant is asked to review two patches of the studied project, one patch ordered alphanumerically ("Alph"), and one patch ordered based on hot-spots ("Hot"). We paired up participants such that each pair had to review the same two patches (P1,P2), but

with different file ordering to control for learning bias. For example, the first participant of the pair would see P1-Alph followed by P2-Hot, while the second would see P1-Hot then P2-Alph. To increase the variety of patches, each pair of participants would work on different sets of patches.

In terms of participant recruitment in the project under study, we selected amongst the top 50% most active reviewers in the last 6 months who also have been relatively active in the last month (i.e., amongst top 40-50% reviewers). The latter choice aimed to find participants experienced in performing reviews, but not overloaded, in order to improve the chances of them participating. We then created pairs of participants that have worked on the same subsystems. For each pair, we also identified a fallback option, i.e., a third participant that could be invited if one of the original participants in the pair is unavailable or unresponsive. The sets were constructed in such a way that all three participants had worked on common subsystems of the studied project.

Each set was then associated with two patches (from the participants' common subsystems) that none of the participants had submitted or reviewed before. We selected only patches of exactly five files, similar to Fregnan et al. [147]. This choice is motivated by the need for patches large enough to observe the impact of ordering files while keeping the review effort manageable to attract more participants. The selected patches were not older than 2 months prior to the study, had received reviews, and needed at least one later revision or involved at least one thread of review comments.

The user study was performed via a mock online review platform. The main page explained to participants that they should not access any other platform to help them perform the review, and provided access to the 2 patches to review. The actual review pages had a user interface mimicking the review environment used at the company,

but customized to allow us to change the file ordering. We also incorporated back-end logic to monitor the start and end time of reviews and to record when each changed file was visible on the screen during the review.

The user study happened over the span of 15 work days, during which we contacted 16 pairs (i.e., 32 participants and patches) via our industrial partner's internal chat application. 11 spare participants from the invited pairs' corresponding set had to be contacted. Out of the 43 contacted participants, 29 participants performed the study, yielding 55 reviews in total (28 with alphanumeric ordering and 27 for hot-spot-based ordering, thus, not all participants did their 2 reviews). 20 patches were reviewed using both ordering setups. The study's recruitment message mentioned that the participation would take around 20 minutes (although most people took longer) and provided a link to our custom review platform. While the participants were aware of the study being about review processes, they were not explicitly told that we explored the impact of file re-ordering. Participants were not offered any compensation for participation, but they participated during their work hours, using their work computer.

5.2.2 Results

Table 5.1 shows that the median duration of individual reviews was 11.4 minutes for both orderings, without significant difference in distribution (Mann-Whitney test: p-val= .39). Analyzing the review duration came with some challenges, as variations in duration can come from many factors (e.g., participants' review habits, distraction by other tasks). 4 reviews took more than an hour, yet

Table 5.1: User study's patch run-through analysis: depending on the ordering approach, we report the number of patches reviewed, the median (Med.) and Average (Ave.) duration of the review, and the median and average ranking position of the files' initial opening during the reviews.

Ordering	**#Patches**		
	Total	**Duration**	
		Med.	**Ave.**
Hot-spot-based	27	11.4m	13.6m
Alphanumeric	28	11.4m	10.4m

Ordering	**Ranking position of files' first opening**									
	Median					**Average**				
	F1	F2	F3	F4	F5	F1	F2	F3	F4	F5
Hot-spot-based	1	2.5	3	4	3	2.26	2.96	3.14	3.26	3.30
Alphanumeric	1	2	3	3.5	4	2.25	2.92	2.93	3.21	3.42

Table 5.2: User study's review comment analysis: depending on the ordering approach, we report the total number of comments, the comments' position (files F1 to F5 or general Gen), the number of comments that found hot-spots, and the number of hot-spot files that were commented on.

Ordering	**#Comments**							
	Total	**Position**						**Found**
		F1	F2	F3	F4	F5	Gen	**Hot-spot**
Hot-spot-based	**77**	**36%** (28)	**32%** (25)	12% (10)	5% (4)	6% (5)	**5%** (5)	**51%** (40)
Alphanumeric	65	27% (18)	15% (10)	**16%** (11)	**23%** (15)	**15%** (10)	1% (1)	41% (27)

Ordering	**#Hot-spot files commented**		
	Total	**Precision**	**Recall**
Hot-spot-based	23	53% (out of 43)	28% (out of 80)
Alphanumeric	18	40% (out of 44)	20% (out of 88)

manual checking of the recorded review events (i.e., opening/closing of files and insertion of review comments) revealed gaps of more than 40min of inactivity for all 4 data points. We thus ignored those data points for the duration analysis. When aggregating the duration per patch, 8 patches out of 17 required a longer duration using hot-spot-based ordering compared to 9/17 patches taking longer for alphanumeric.

The order in which reviewers opened changed files tends to follow the provided order. Table 5.1 shows the median/average ranking position of the five (ordered) files of each patch in terms of when a file is opened for the first time. For instance, if in at least half of the patches for a particular ordering file F1 is opened first, its median ranking position would be 1. We can see that both the median and average ranking increase with the file number, except for file F5 (hot-spot based). The latter exception seems to be due to some reviewers starting to opening files from the end, and other reviewers opening a C++ file's corresponding header file before opening other files. The average ranking values show that the order can still vary a lot, depending on the reviewers' experience with the given files, the types of files and their review habits.

Across patches, we observe an average of 2.9 comments for hot-spot-based compared to 2.3 for alphanumeric (+23%) order. Table 5.2 shows that the total amount of submitted comments on the hot-spot-based ordering is 77 comments, 12 comments more than the alphanumeric ordering, even though one additional alphanumeric patch was reviewed. Among both sets of comments, 83% of the comments seem to be actionable, while the rest comprise comprehension questions.

Hot-spot-based ordering gathers 80% of review comments on the first 3 files, compared to only 58% for alphanumeric ordering. This is somehow

expected, as the hot-spots historically observed in our datasets by definition are a good indication of where comments are needed.

The participants' review comments on hot-spot ordered patches show an increase in precision (+13%) and recall (+8%) for finding hot-spots. By analyzing if the comments (1) are equivalent to ground truth comments or (2) pinpoint the location of a later revision, we found that 40 comments (51%) of the hot-spot-based ordering identified actual hot-spots compared to only 27 (41%) of the alphanumeric order. On average, 1.5 comments per patch found a hot-spot in the case of hot-spot-based ordering compared to 1 per patch for the baseline (+54%).

It should be noted that two participants indicated to us that they felt "something wrong" with the (hot-spot based) file ordering. In 3 out of the 27 reviews with hot-spot-based ordering that included C++ code, a participant initially missed the header file (extension ".h"), wrote a comment about it missing, then canceled the comment after seeing the missing file later in the ordering. This shows us that certain files forcibly should be shown together even when using hot-spot-based ordering.

File ordering has an impact on the review process. While there is no impact on review duration, the hot-spot-based ordering correlates with more comments (+23%) targeted at parts of the code known to need revising or commenting (+13% precision and +8% recall). This implies that it is important to improve prediction models of review hot-spots.

5.3 Methodology for Empirical Study of Hot-Spot Prediction

Based on the findings of our user study that file re-ordering during review activities is helpful, we want to explore how to improve the models for the three common

review activity prediction tasks (i.e., reviewed, commented, hot-spot). In particular, we empirically evaluate prediction models using different variations of two kinds of embeddings (i.e., text-based and feature-based) on 5 large datasets, amongst which 2 closed-source datasets.

5.3.1 Labeling

During review activities, reviewers comment on a patch, i.e., a code submission. Those patches involve one to several commits, where the first commit is the initial submission and the follow-up commits are revisions. Each commit contains changed files, which are composed of code line additions and/or deletions. Line modifications are lines that receive both a deletion and addition. Most diff tools will show the before and after versions of the file.

Our review activity prediction tasks extend existing code diff quality prediction works [13,32] in order to highlight changed files that most likely should be commented on or revised. Past approaches have used a binary labeling based on if the changed file was commented on or revised [13, 32]. Some works have also addressed the question of identifying useful comments, as they consider that commented changes that do not lead to revisions are not useful [33]. However, we consider such comments to still have value, as they can lead to future code submission or discussions.

Hence, in this chapter, we consider similar labels ("commented" and "revised") as prior works, but also define a "hot-spot" label, which indicates whether a code change has any review activity (i.e., comment, revisions, or both). This is because, while comments are meant to lead to revisions, and revisions to be caused by comments, review comments might indicate the need for revisions elsewhere in the patch or

Table 5.3: Label descriptions (file-level).

Label	Description
Commented	Changed file receiving a review comment on any of its lines.
Revised	Changed file for which at least one of its lines will be revised in a future patch.
Hot-spot	Changed file that is either commented, revised, or both.

changes in future independent patches. Furthermore, we observe that 10-17% of patches across our datasets contain changes that are revised, while never commented, typically because of a review comment made on another file or outside the reviewing platform. If considering only patches that were commented on, those cases would thus be missed, hence the importance of considering hot-spot prediction.The labels' definitions can be found in Table 5.3. This labeling is done at a file-level granularity, following Hong et al.'s approach [13].

5.3.2 Datasets

In this chapter, we use three open source datasets and two closed source datasets provided by our industrial partner (different from the dataset used in the user study). The open source datasets we consider are those of Hong et al. [13], i.e., Qt Base (Base), Openstack Ironic (Ironic), Openstack Nova (Nova), which were used for commented and revisioned prediction tasks [12, 13]. We then used Gerrit's API to gather extra information about the changed files, i.e., author id, reviewer ids, and full file diff (since their replication package only has the added lines), needed for our embeddings.

The closed-source datasets were provided by our industrial partner, and contain reviews of two projects developed over the course of 5-6 years that used a traditional review process. Though we are not able to share these datasets, we are interested

Table 5.4: The revision activity in the studied datasets.

Project	Period	#Patches				#Changed files			
		All	Commented	Revised	Hotspot	All	Commented	Revised	Hotspot
Base	2 years	5,390	32%	31%	42%	18,950	13%	18%	23%
Ironic	3 years	1,050	45%	38%	53%	4,193	16%	26%	31%
Nova	3 years	3,678	54%	42%	63%	11,997	26%	30%	43%
Closed1	5 years	34,511	43%	49%	63%	270,396	10%	25%	28%
Closed2	6 years	40,236	45%	49%	64%	390,075	9%	24%	27%

in showing those results as the datasets are much larger, go over longer periods and contain a denser review activity, as shown in Table 5.4.

More information on the datasets can be found in Table 5.4, which only considers the initial commit of each patch and its related changed files. In the context of review activity prediction, we make predictions on the initial commits only. Although the follow-up commits/revisions are necessary in our dataset for labeling, they are not part of the training or test datasets. The frequency of the three labels at the file and patch levels shows that open-source projects tend to have smaller patches in terms of changed files as they have on average 3-4 times more changed files than patches. Closed-source datasets, on the other hand, have 9-10 times more changed files than patches, thus larger patches. Also, as the patches are larger, closed-source datasets also report slightly lower review comment ratio (9-10% compared to 13-26% for open-source). For all datasets, we observe that the granularity of file changes compared to patches makes the labels more imbalanced.

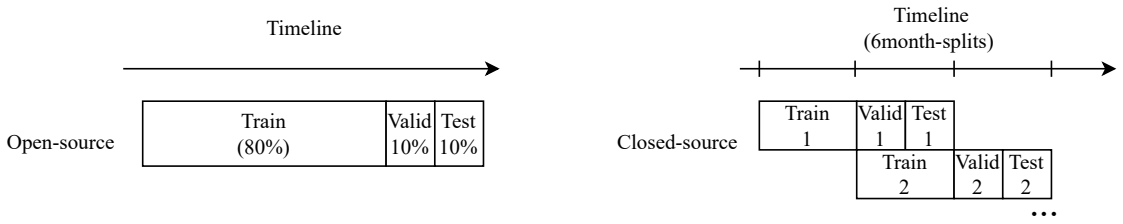


Figure 5.1: Data split for open- and closed-source datasets.

5.3.3 Data splitting

As shown in Table 5.4, the open- and closed-source datasets have different sizes, as the open-source datasets have 4k to 18k changed files, whereas the closed-source datasets have 270k to 390k. Since we are considering, among others, Bag-of-Words embedding approaches, training on the full closed-source datasets is not feasible, as the vocabulary of the corpora is too large to maintain. For this reason, we need different data-splitting approaches for both sources of data. However, both splitting approaches respect the data's timeline: changed files present in the test have to be posterior in time to all the changed files in the training set in order not to contaminate the training set, same as for the validation set. Both splitting approaches are shown in Figure 5.1.

In the case of the open-source datasets, we split each dataset along its timeline into a training set validation set and test set of size 80%-10%-10% respectively, similarly to Hong et al.'s approach [13].

In the case of the closed-source datasets, we split each dataset into 6-month periods. Closed1 has 10 such periods and Closed2 has 12. From preliminary analysis, we observed that a 6-month period size still allows us to apply the Bag-of-Words approach while keeping a reasonable vocabulary size. Then, we apply a sliding window where we train on a period and validating on the first half of the following period

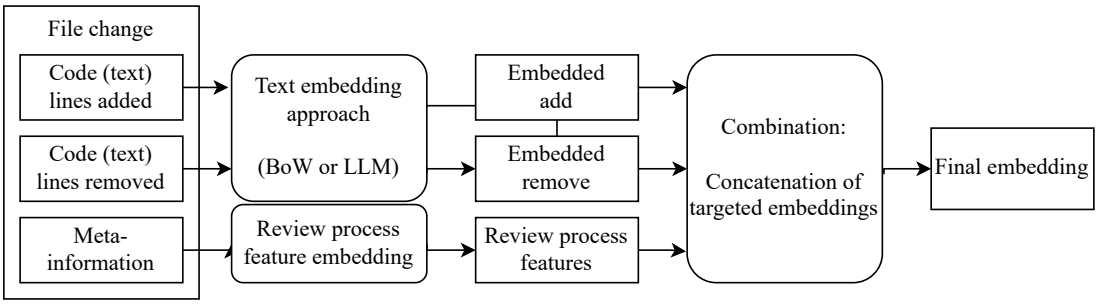


Figure 5.2: Embedding approach. The file change is composed of lines added, lines removed and meta-information about the submission (e.g., timestamp, author and reviewer ids, patch id). The combination will concatenate the targeted embeddings, i.e., the text embedding, features, or both. The final embedding for each file is 1D.

and test the second half. Each training yields an individual model, giving 9 trained models for the Closed1 dataset and 11 for Closed2.

5.3.4 Embeddings

In the context of this chapter, we consider two types of text embeddings, i.e., Bag-of-Words (BoW) and Large-Language Model (LLM) encoding, and several review process feature sets.

Figure 5.2 illustrates the embedding process. For each changed file, we generate all the embeddings, both text and feature-based. At the combination stage, we choose which embeddings to consider, text, feature-based or both. In the case of text embeddings, we have the choice between only considering added lines or also removed lines. In the latter case, added and removed lines are still embedded separately and concatenated at the combination stage. For RQ1, we only consider the text embeddings ("embedded add" and "embedded remove"), and we don't take into account any review process features. RQ2 considers only the review process features, but the specific features selected depend on the chosen feature setup. For RQ3, both

text embeddings and review process features are considered together and combined using a concatenation approach. In each case, the final embedding of a file is a one-dimensional vector.

Bag-of-Words

The BoW approach represents each input as a vector of word frequency where each column represents a word in the corpus vocabulary and the value in the vector at that column indicates the number of occurrences of the word in the input. Considering code changes as input, we have to clean the input in this context, since the source code of changes have many symbols that are not text. We follow the state-of-the-art's approach [13] for code pre-processing, which only considers alphanumeric terms. Numbers are also replaced by a token "`<NUMBER>`". Still following this approach, we do not perform any stemming, lowercase or lemmatization transformation as such approaches are valuable for natural languages used by humans but do not make sense in the case of code. Regarding the BoW computation itself, we use the `Countvectorize` function of the Scikit-Learn Python library [150]. The `Countvectorize` function is trained on the training set, then is applied to the test set for each data split. Words that appear in the test set but not in the training set will thus not be included in the bag of words. To reduce the vocabulary size, we remove words that appear only in one single changed files in the corpus as well as words that are too common (i.e., appear in more than 50% of the changed file corpus).

Large-Language Model encoding

To work with LLM models, the Hugging Face community provides many pre-trained models available to use as-is [151]. We selected Bloom from Big Science, an LLM trained on many programming languages (e.g., Java, C++, Python), who's direct use is text generation and exploring charateristics of language [152]. The advantage of Bloom is that it provides a condensed version of the model that still provides good results for NLP tasks. We do use `big science/bloom-560m` in the context of this chapter, which is about 4 GB compared to the normal version of Bloom, which is about 375 GB. Also, it counts more than 1.4M downloads in the month of August 2023, making it the most popular LLM pre-trained on programming languages at the time of our empirical study. To obtain the encoding, we used `SentenceTransformer`, a Hugging Face Python library [153]. For the encoding step, we do not clean the code, as Bloom is trained on raw code.

Review process features

software tasks such as risk prediction [16, 46] and bug prediction [45, 66] have been shown to use software process features for their training and getting respectable performance. Based on their example and inspired by their features, we defined review process features following two main categories: size-related features and historical features.

The count features refer to metrics considering the size of the changed file targeted by the prediction, as well as contextual features regarding the size of the changed files' patch. Historical features refer to the past activities that happened surrounding the targeted file, but also the experience of the code's author and of the patch's reviewers

known at the time of the prediction. The definitions of our features can be found in Table 5.5. Those features are generated on each individual dataset, based on the timestamps of the patch submission.

Table 5.5: Description of review process features generated at the file-level granularity. <LAB> refers to the different labels we consider.

		Feature name	**Description**
Count	1)	c_add	# lines added in the file.
	2)	c_rem	# lines removed in the file.
	3)	c_unchanged	# lines unchanged shown in the file. For Gerrit, at most 10 lines of context are shown above and under changed lines as context. In the tool used by our industrial collaborators, the context is at most 5 lines.
	4)	c_add_rev	# lines added in the patch of the file.
	5)	c_rem_rev	# lines removed in the patch of the file.
	6)	c_unchanged_rev	# lines unchanged in the patch of the file.
	7)	r_add_rev	$\frac{c_add}{c_add_rev}$
	8)	r_rem_rev	$\frac{c_rem}{c_rem_rev}$
	9)	r_unchanged_rev	$\frac{c_unchanged}{c_unchanged_rev}$
File's history	10)	file_exp	# past and current patches that included changes to this file (min = 1).
	11)	dir_exp	# past and current patches that included changes to this file's directory (min = 1).
	12- 14)	r_<LAB>_file	File's average <LAB> ratio before the current patch (default: 0, value between 0 and 1).
	15- 17)	r_<LAB>_dir	The directory of the file's average <LAB> ratio before the current patch (default: 0, value between 0 and 1).
Author's history	18)	auth_exp	# past and current patches submitted by the author (min = 1).
	19)	authfile_exp	# past and current patches in which the current file was submitted by the author(min = 1).
	20)	authdir_exp	# past and current patches in which the current file's directory was submitted by the author (min = 1).
	21- 23)	r_<LAB>_author	Author's average <LAB> ratio before the current patch (default: 0, value between 0 and 1).

	24- 26)	r_<LAB>_authfile	Author's average <LAB> ratio on the current file before the current patch (default: 0, value between 0 and 1).
	27- 29)	r_<LAB>_authdir	Author's average <LAB> ratio on the current file's directory before the current patch (default: 0, value between 0 and 1).
Reviewers' history	30)	max_reviewers_exp	Maximum # past and current patches reviewed by one of the reviewers out of all this patch's reviewers (min = 1).
	31)	mean_reviewers_exp	Average # past and current patches reviewed by one of the reviewers out of all this patch's reviewers (min = 1).
	32- 34)	r_<LAB>_max_ reviewers	Maximum Reviewers's <LAB> rate before the current patch (default: 0, value between 0 and 1).
	35- 37)	r_<LAB>_mean_ reviewers	Average Reviewers's <LAB> rate before the current patch (default: 0, value between 0 and 1).

5.3.5 Classification

Similar to Hong et al. [13], we considered Random Forest (RF), Naive Bayes (NB), Nearest Neighbors, Logistic Regression, Decision Trees, and off-the-shelf LLMs (`bloom560m`) to classify the changed files as "revised", "commented" and/or "hot-spots" as three separated tasks. From preliminary analysis on those classifiers, we narrowed down the list to Random Forest and Naive Bayes, as they performed better in terms of AUC and F1-score for our task, regardless of the used embeddings. This confirmed Hong et al.'s observation [13] that Random Forest gives the best AUC for their task, followed in most cases by Naive Bayes. During preliminary analysis, we used the off-the-shelf Bloom LLM classifier, yet this did not perform as well as the considered ML models. As all the datasets are imbalanced considering the changed file granularity, we consider the SMOTE data sampling approach during

training [154, 155]. Models are trained on a per-project basis.

In the end, we have thus two different models (RF and NB) with and without the SMOTE sampling approach (models with SMOTE will be referred to as RFs and NBs), making 4 options for model selection. Each model is trained on a specific label (i.e., commented, revised or hot-spot), thus needing 12 trained models in total for each dataset-split and each setup of embeddings.

5.3.6 Evaluation

We empirically evaluate the models for the three tasks using three open-source and two closed-source datasets. To do the evaluation, we report AUC, as Hong et al. [13]. We also report metrics used for imbalanced labeled datasets: F1-score (F1 = $\frac{2}{pre^{-1}+rec^{-1}}$) and Geometric Mean (GM = $\sqrt{pre \cdot rec}$), which are computed based on the precision (pre = $\frac{TP}{TP+FP}$) and recall (rec = $\frac{TP}{TP+FN}$). Those metrics need a threshold to define when the prediction is considered as true or false. We choose a threshold that maximizes the F1-score on the validation sets, as it captures best imbalance of the datasets [156].

In Machine Learning based approaches, it is best to train and test models several times to verify the robustness to randomness in the approaches or data splits. For the closed-source datasets, we mentioned in Section 5.3.3 how the dataset is split into several periods and thus runs. For the open-source datasets, as they are smaller than the closed-source datasets, only one data split is done, following the process explained in Section 5.3.3. However, each embedding setup and model is run 5 times, with a different seed in each iteration. That seed is used in the model (e.g., random forest takes a random seed) and in the SMOTE sampling when used.

For all the datasets, out of the different models (i.e., RF, RFs, NB, and NBs), we report the model with the highest median F1-score on the validation set, as that is the metric we want to optimize. The results shown in the tables are the metrics for the run with median F1-score for a given embedding setup and its best model.

5.3.7 Statistical tests

As mentioned in the previous section, we run each setup at least 5 times per dataset for each label and the median results are shown in the different tables. Hence, we need to perform statistical tests to validate whether or not the differences observed are significantly different.

Since we have a limited number of runs, we decided to group the runs into sub-categories of datasets: open-source datasets (i.e., Base, Ironic and Nova), closed-source datasets (i.e., Closed 1 and 2) and all datasets. When comparing two setups for a given label, we take all the runs of that setup for all the datasets in the subcategory. We apply a Wilcoxon paired test [157] with α-value = .01. We calculate the Cliff's Delta effect to evaluate the impact of the significant differences [158].

As an example, let us consider the comparison of two setups for the one label and the sub-category open-source datasets. Each dataset in that case has 5 runs of the best model, i.e., runs B1-5 for Base, I1-5 for Ironic and N1-5 for Nova. For each setup, we align the runs into a vector yielding one vector by setup of the form $[B1-B5\ I1-I5\ N1-N5]$, in this case of length 15. Then, we perform the pairwise test between both vectors where a p-value inferior to .01 is reported as significantly different, in which case we also compute the effect size on the same vectors.

5.4 RQ1: Which text embeddings of code diffs perform best for review activity prediction?

5.4.1 Motivation

In their paper, Hong et al. explore the use of the Bag-of-Words (BoW) to embed code changes [13]. Contrary to previous work using full-code changes to perform prediction [159], they only consider added lines of a code change. This choice has the advantage of not having removed or unchanged lines contaminating the text embedding, as those lines could appear as noise in the training. However, the prediction will thus have less context since only the added lines are considered.

Furthermore, the BoW approach, though easy to implement and easily explainable, presents several disadvantages. First, this approach does not scale with large datasets as the size of the vocabulary increases exponentially with the corpus size. Secondly, BoW does not capture the essence of source code, as the order of words does not impact the embedding. It captures the number of occurrences of words and thus the size of the code as well.

On the other hand, LLM pre-trained encoders are made to represent input data in a representative way, as to capture the underlying structure and semantics of the code. Also, the input encoding is a fixed-size embedding. However, the difficulty with these approaches is that models are more difficult to explain based on their feature importance, as the abstraction of the original input into the encoded output is not understandable to humans.

5.4.2 Approach

In RQ1, we explore how text embeddings impact the review activity predictions at file-level. Following the findings of Hong et al. [13], we will explore both (1) keeping only the added lines of a code change and (2) consider embedding removed lines in addition to the add lines. Among the different variations of each text embedding, we will identify the best embedding for a given dataset and label as "Best text" embedding.

Table 5.6 shows the results of both considered embeddings, BoW and LLM encoding, with the variations considering only added lines (+) or both added lines and removed lines (+/-). We performed a statistical pairwise test using Wilcoxon Test, comparing "BoW +" embedding to each text embedding for each sub-category of datasets (i.e., open-source, closed-source, or both), as mentioned in Section 5.3.7.

5.4.3 Results

Compared to "BoW +", no specific text embeddings report significant differences in F1-score across all datasets, for any labels (p-value = .012 to .441). The only case that returned significant improvement with small effect size is "LLM +/-" compared to "BoW +" for the label "hot-spot" (p-value = .001), with an improvement of up to +4%. Also, we notice a significant improvement with small effect regarding AUC with embedding "BoW +/-" for all datasets on labels "commented" and "revised" (p-value = .0).

However, when comparing the "Best text" embedding with the "BoW +", the F1-score is found to be significantly better with small effect size for labels "revised" (p-value = .007) and "hot-spot" (p-value = .007). Out of

Table 5.6: Results for RQ1: text embeddings for the code review activity prediction task at file-level. The results shown are returned for the best model ("mod") across each variation of text embedding. Blue cells indicate the maximum values for each metric per dataset and per label.

	Text Embed	Commented						Revised						Hot-spot					
		auc	f1	gm	pre	rec	mod	auc	f1	gm	pre	rec	mod	auc	f1	gm	pre	rec	mod
Base	BoW +	67	38	63	27	63	RF	67	42	59	40	44	RFs	66	49	62	42	59	RFs
	BoW +/-	68	40	64	30	58	RFs	67	44	62	37	55	RFs	67	50	62	41	63	RFs
	LLM +	69	39	65	28	69	RFs	68	46	62	42	51	RFs	68	51	64	44	60	RFs
	LLM +/-	69	40	65	28	69	RFs	68	46	63	39	52	RFs	69	52	63	44	61	RFs
Ironic	BoW +	77	46	67	39	53	RFs	70	53	66	41	81	RFs	71	57	63	45	82	RF
	BoW +/-	78	45	69	36	62	RFs	69	50	64	42	68	RFs	71	57	64	47	76	RF
	LLM +	78	45	66	39	52	RFs	67	52	63	38	75	RFs	70	58	65	47	76	RFs
	LLM +/-	78	40	61	40	45	RFs	67	51	62	37	77	RFs	70	58	66	48	76	RFs
Nova	BoW +	75	54	67	56	52	RFs	72	48	66	37	69	RFs	72	57	46	42	91	RFs
	BoW +/-	75	55	68	58	53	RFs	73	50	67	41	67	RFs	72	57	48	42	90	RFs
	LLM +	78	55	67	57	51	RFs	75	50	68	40	69	RFs	76	57	35	40	98	RF
	LLM +/-	78	53	66	59	48	RFs	75	51	67	41	73	RFs	76	57	37	41	97	RFs
Closed1	BoW +	74	38	62	31	48	RFs	70	46	65	37	60	RFs	67	51	63	41	67	RFs
	BoW +/-	74	36	65	28	50	RFs	71	44	64	37	57	RFs	68	51	61	40	73	RFs
	LLM +	75	37	62	29	53	RFs	72	47	66	36	70	RFs	65	55	55	42	79	RF
	LLM +/-	74	37	63	27	54	RFs	72	45	64	38	63	RFs	69	53	63	41	68	RFs
Closed2	BoW +	72	32	60	26	43	RFs	70	39	63	34	49	RFs	67	48	61	36	77	RFs
	BoW +/-	75	32	64	23	50	RFs	71	40	65	32	60	RFs	67	47	63	36	72	RFs
	LLM +	72	32	62	23	49	RFs	70	41	61	30	61	RFs	66	49	58	40	69	RFs
	LLM +/-	77	32	58	24	42	RFs	73	40	65	28	66	RFs	72	51	65	40	72	RFs

those 10 cases (5 datasets × 2 labels), 8/10 cases show higher F1-score (from +2 to +4%), 1/10 case show no effect and one got lower results (-1%). The label "commented" does not perform significantly better (p-value = .021). It at least matches in 3/5 datasets (up to +2%) and is lower in two cases (-1%). Also, out of the 15 cases, 8/15 cases got higher F1-score results than the baseline for "LLM +/-", 10/15 cases for "LLM +" and 6/15 for "BoW +/-".

We thus conclude that in general, none of the text embedding stands out in all cases, although in practice, choosing the right variations improves F1-score in 10/15 cases. The scalability of BoW still stands, as the vocabulary quickly explodes as the corpus increases in size.

For BoW, we found that 95-99% of the feature importance is focused on 1-5% of top features, compared to 50% of feature importance focused

on 20-40% of top features for the LLM embedding. We apply a feature importance analysis using Python's `feature_importances_`, which measures the percentage of feature importance of each individual feature. Comparison of the "BoW +" and "LLM +" embeddings shows that due to the very large size of the embedding outputs of BoW, the classifier will only focus on a low number of features, with the majority of stored features not playing an important role in the prediction. More details on feature importance can be found in the replication package [148].

RQ1: Compared to Bag-of-Words applied only on added lines, adding the removed lines and/or using the Large Language Model embedding significantly improved the F1-score for labels "revised" (up to +3%) and "hot-spot" (up to +5%) at file-level. The label "commented" has no significant improvement using the new text embeddings.

5.5 RQ2: Which review process features perform best for review activity prediction?

5.5.1 Motivation

The state-of-the-art approach only considers two basic software process features, i.e., number of lines add and removed, which were able to improve the performance of commented and revised prediction tasks [13]. We thus want to evaluate how review process features in general, such as size-related features, but also features based on the history of the file, the author and the reviewers, perform compared to text embeddings, inspired by other software processes that have used similar features in the past [45, 46].

5.5.2 Approach

In Table 5.7 we report the results of different combinations of features:

- "Add/Rem": this set of features refers to the features used by Hong et al. in their approach [13], i.e., "c_add" and "c_rem" in Table 5.5.
- "Count": described in the "Count" section of Table 5.5.
- "Hist": described in the File's , Author's and Reviewers' history sections of Table 5.5.
- "All feat": all the aforementioned review process features, thus all features in Table 5.5.

In RQ2, we only consider review process features, excluding any text embedding, as this is the topic of RQ3. However, we will also report in the table the best results of text embedding from RQ1 as "Best text", per dataset and predicted label in order to see how feature-based performs compared to text embeddings, which (theoretically) have more specific information about the file's content. Again we will define a best setup "Best feat" as being the local best feature-based embedding for each dataset and label. We also applied statistical tests to compare the "Best text" embedding with the feature-based embedding.

5.5.3 Results

For the open-source datasets, the "Best feat" embedding reports no significant difference regarding the F1-score for labels "revised" and "hot-spots" (p-value = .804 and .389), thus managing to perform as well as the

Table 5.7: Results for RQ2: software review process features embeddings on the code review activity prediction tasks at file-level. The results are returned for the best model ("mod") for each setting of text embedding. Blue cells indicate the maximum values for each metric per dataset and per label.

	Text Embed	Commented						Revised						Hot-spot					
		auc	f1	gm	pre	rec	mod	auc	f1	gm	pre	rec	mod	auc	f1	gm	pre	rec	mod
Base	Best text	68	40	64	30	58	RFs	68	46	63	39	52	RFs	69	52	63	44	61	RFs
	Add/Rem	59	33	58	25	50	RF	65	44	61	34	66	RF	64	50	55	36	81	RF
	Count	65	36	59	24	73	RF	65	42	60	38	49	RF	66	50	58	37	78	RFs
	Hist	62	33	39	20	93	RFs	63	43	59	32	70	RF	60	47	25	31	97	RF
	All	73	40	65	28	74	RFs	74	50	65	49	51	RFs	73	54	66	45	70	RFs
Ironic	Best text	77	46	67	39	53	RFs	70	53	66	41	81	RFs	70	58	65	47	76	RFs
	Add/Rem	53	30	53	20	68	RF	61	45	43	31	90	RFs	62	51	37	36	92	RFs
	Count	65	36	62	27	55	RF	61	44	58	38	49	RFs	63	52	56	40	72	RFs
	Hist	57	30	49	20	71	RFs	70	53	65	40	84	RFs	67	55	61	44	78	RFs
	All	67	36	62	25	67	RF	68	51	65	43	64	RFs	69	56	62	45	80	RFs
Nova	Best text	75	55	68	58	53	RFs	75	51	67	41	73	RFs	76	57	37	41	97	RFs
	Add/Rem	63	40	39	26	88	RF	65	45	63	36	64	RFs	67	56	34	40	95	RFs
	Count	68	47	62	34	78	RFs	72	48	65	35	73	RF	68	58	54	44	86	RF
	Hist	60	30	47	32	28	RFs	56	38	56	29	57	RF	58	41	53	49	36	RFs
	All	70	40	17	25	100	RFs	69	47	62	45	45	RFs	67	55	10	38	100	RF
Closed1	Best text	74	38	62	31	48	RFs	72	47	66	36	70	RFs	65	55	55	42	79	RF
	Add/Rem	63	24	59	19	56	RF	67	39	60	29	64	RF	60	48	49	36	73	RF
	Count	74	36	64	28	50	RFs	75	47	68	35	64	RF	68	55	60	40	82	RF
	Hist	59	25	53	16	83	RFs	58	38	47	24	83	RF	59	47	42	32	83	RF
	All	77	37	66	32	52	RF	75	48	69	35	67	RFs	69	56	64	43	78	RFs
Closed2	Best text	77	32	58	24	42	RFs	70	41	61	30	61	RFs	72	51	65	40	72	RFs
	#Add/Rem	67	24	60	17	46	RF	65	35	61	23	64	RF	59	45	45	29	84	RFs
	Count	79	34	67	26	52	RF	72	42	66	32	62	RFs	68	49	63	36	77	RFs
	Hist	59	18	52	11	60	RFs	63	34	55	22	79	RF	61	45	45	32	87	RF
	All	81	37	69	29	56	RF	76	45	68	35	64	RFs	74	53	67	41	76	RFs

(supposedly richer) text embeddings. We can note that for 3/6 cases, at least one of the feature-based approaches improves F1-score compared to "Best text" (up to +4% F1-score). Out of the 6 cases, "All feat" reports better F1-score than the "Best text" in 2/6 cases and "Count" reports a better F1-score in 1/6 cases. The F1-scores for the label "commented" are significantly lower with medium effect (p-value = .007). In 4 out of 9 open-source cases, the "Best text" embedding performs better than the feature-based embeddings (+2 to +12% F1-score). This is expected, as the text embedding has more specific information than the high-level information of the feature-based embeddings.

For the closed-source datasets, the "Count" embedding is not significantly different from the "best text" embedding for any label with respect to F1 score (p-value = .355 to .966). Statistically, the text embedding and the "Count" perform similarly though having access to different types of information, the "Count" approach being less computationally expensive as the embedding size is much smaller. In practice, we observe that in those 6 cases (2 datasets × 3 labels), "Count" reports at least as well in F1-score to "best text" in 4/6 cases (up to +2%) and lower in the last 2 cases (up to -3%).

Similarly, for the closed-source datasets, "All feat" does not report significantly different F1-score for any label compared to "best text" (p-value = .034 to .393). It reports significantly different AUC for labels "revised" (p-value = .0) and "hot-spot" (p-value = .003) with large effect in both cases. We observe that "All feat" gets a better F1-score in 5/6 cases (+1 to +5%), -1% in the last case, and it reports better AUC in all 6 cases (+2 to +6%). Furthermore, "All feat" compared to "Count" reports significantly better F1-score with small effect regarding the label "commented" (p-value = .0) and not significantly different for the labels "revised" (p-value = .014) and "hot-spot" (p-value = .032). However, "All feat" improves F1-score compared to "Count" in all 6 cases, with +1 to +4%. In practice, adding the history-based features does improve F1-score and performs better than "Best text".

Through these results, we observe that in the case of open-source datasets, the review process feature-based approach does not consistently perform as well as the "Best text" embedding, though the F1-score reported by the best feature embeddings per dataset and label improved performance in 4/9 cases (up to +12%). Regarding

closed-source datasets, the "All feat" embedding performs as well as the "Best text" embedding, while reducing the computational effort of model training thanks to the reduced dimensionality. Looking at feature importance per feature category of Table 5.5, we found that all datasets benefit most from the "Add/Rem" and "Count" categories, whose features each represent 2 to 13% of the overall feature importance. This indicates that those features regarding the size of changed files have the most value, including also the contextual features regarding the full patch size. More details on feature importance can be found in the replication package [148].

RQ2: In the closed-source projects, considering all review process features performs as good as the best text embeddings at file-level in terms of F1-score, while reducing the computational effort as the embedding is of lower dimensionality. For open-source datasets, the best feature embedding per dataset and label matches the F1-score of text embeddings, although no specific feature-based embedding is consistently as good. All datasets benefit the most from size-based features.

5.6 RQ3: How does the combination of review process features with text embeddings perform for review activity prediction?

5.6.1 Motivation

Hong et al. propose in their paper a review activity prediction approach using a BoW embedding in addition to features measuring the number of lines added and removed to a file [40]. In RQ1 and RQ2, we explored how text embeddings and review process feature embeddings perform separately on the three tasks and what they capture about the code submission. In this question, we combine both types of

embeddings to find the best embedding, compared to the state-of-the-art.

5.6.2 Approach

We combine each text embedding with each review process feature embedding presented in the previous RQs. The results are shown in Table 5.8. Each project is presented with three comparisons: the baseline (state-of-the-art [13], i.e., BoW embedding with "Add/Rem" features), "Best text" (from RQ1), and "Best feat" (from RQ2). We report two main combinations: first, "Heavy combo" which refers to our most "heaviest" embedding compared to the baseline, i.e., the one considering the "LLM +/-" text embedding and "All feat" feature-based embedding. Then, we also report "Best combo", which reports the combination with the best F1-score per dataset and label. The lines directly following "Best combo" report the specific combination chosen (i.e., the name of the text and review process feature embeddings used).

5.6.3 Results

For closed-source datasets, "Heavy combo" improves significantly the F1-score compared to the baseline regarding labels "revised" (p-value = .002 with medium effect) and "hot-spots" (p-value = .004 with large effect). Out of those 4 cases (2 datasets × 2 labels), the improvement on F1-score is +3 to +6%. In the case of the label "commented", "Heavy combo" performs better for Closed2 (+4%) and lower for Closed1 (-3%). In all 6 cases, the AUC is improved (+3 to +8%). Furthermore, the "Heavy combo" is also the "Best combo" in the case of the label "revised".

For open-source datasets, "Heavy combo" is not significantly different than the baseline regarding F1-score for any label (p-value = .048 to .421). In practice, it improves F1-score in 5/9 cases (+1 to +6 %) and AUC in 7/9 cases (+1 to +5%), but not significantly.

For all datasets, "Best combo" performs +1 to +9% better regarding F1-score than the baseline, with medium effect for labels "commented" and "revised" (p-value = .0) and large effect for large "hot-spot" (p-value = .0). "Best feat" significantly improves the F1-score for labels "revised" (p-value = .0 medium effect) and "hot-spot" (p-value = .004 small effect), with improvement up to +8% for "revised" and +6% for "hot-spot" compared to the baseline.

Table 5.8 shows the different embedding choices for the best combinations of text-embedding and feature embedding regarding F1-score. We observe that out of the 15 cases, the best setup chosen involved BoW in 6/15 cases and LLM in the other 9. The text embedding including both added and removed lines was chosen in 9/15 cases and only the added lines in 6/15 cases. Overall, the classic text embedding "BoW +" was chosen only twice. It should also be noted that in 3/15 cases, the "Best feat" embedding got the best results, which indicates that in those cases, combining features with the text embedding did not even improve performance, making the lower-dimensionality review process feature-only approach the most performant.

Regarding the feature embedding choice for "Best Combo", "Add/Rem", the simpler embedding, was never chosen, "Count" was chosen 3/15 times, "Hist" was chosen 4/15 times, and "All feat" was chosen 8/15, making the latter the most popular feature-embedding.

Though the "Heavy combo" embedding is not unanimous, as it improved F1-score

for 10/15 cases, the new review process features and text embeddings variations that we consider are worth exploring as their best combination ("Best combo") improves F1-score in all cases.

RQ3: The Best combination of embeddings improves significantly the F1-score (+1 to +9%) for all datasets and labels at file-level. Though combining LLM embeddings and all process features is the most common best combination, there is no consensus on the best combination. The heavy combination reports significantly higher F1-score for closed-source datasets regarding the labels "revised" and "hot-spots", and in practice improves the F1-score in 10 out of 15 cases across all datasets.

5.7 Discussion

5.7.1 Hot-spot detection for file re-ordering

To evaluate the impact of hot-spot prediction-based file ordering (instead of a wizard-of-oz setting, using the ground truth), we took patches with at least 2 changed files and at least one of the files labeled as hot-spot, then made model predictions using RQ3's heavy combination. We ordered each patch's changed files based on their predicted probability of hot-spotness. We compute the average Recall@k ($= \frac{TP}{TP+FN}$) across patches, where TP are the true positive cases in the top k, and $(TP + FN)$ is the total number of hot-spots (maxed at k). k is chosen depending on the size of the patch ($size$), so Recall@50% means k = $\frac{size}{2}$ and Recall@25% means k = $\frac{size}{4}$. Table 5.9 compares the recall of our ordering with the alphanumeric order, i.e., the default ordering used by review environments, on patches of different sizes. **Our prediction-based empirical analysis shows that hot-spot-based file ordering**

Table 5.8: Results for RQ3: combining the text embeddings with software meta-features embeddings for the code review activity prediction task at file-level. The results are returned for the best model ("mod") for each setting of text embedding. Blue cells indicate the maximum values for each metric per dataset and per label.

	Text Embed	Commented						Revised						Hot-spot					
		auc	f1	gm	pre	rec	mod	auc	f1	gm	pre	rec	mod	auc	f1	gm	pre	rec	mod
Base	Baseline	68	39	62	29	52	RFs	67	42	59	40	42	RFs	66	50	62	41	65	RFs
	Best text	68	40	64	30	58	RFs	68	46	63	39	52	RFs	69	52	63	44	61	RFs
	Best feat	73	40	65	28	74	RFs	74	50	65	49	51	RFs	73	54	66	45	70	RFs
	Heavy combo	71	40	64	28	66	RFs	70	47	64	40	59	RFs	70	52	61	39	78	RFs
	Best combo	73	41	65	29	71	RFs	70	47	64	40	59	RFs	71	53	62	40	76	RFs
		BoW +/- & All						LLM +/- & All						LLM + & All					
Ironic	Baseline	79	46	69	37	58	RFs	71	51	65	41	69	RFs	72	57	65	47	74	RF
	Best text	77	46	67	39	53	RFs	70	53	66	41	81	RFs	70	58	65	47	76	RFs
	Best feat	65	36	62	27	55	RF	70	53	65	40	84	RFs	69	56	62	45	80	RFs
	Heavy combo	79	44	63	41	47	RFs	67	48	60	36	74	RF	73	58	67	53	67	RFs
	Best combo	78	47	70	38	64	RFs	75	54	67	51	54	RFs	73	60	69	52	70	RFs
		LLM + & Count						BoW +/- & Hist						LLM +/- & Hist					
Nova	Baseline	77	56	68	60	52	RFs	71	47	65	38	62	RFs	73	57	49	42	88	RFs
	Best text	75	55	68	58	53	RFs	75	51	67	41	73	RFs	76	57	37	41	97	RFs
	Best feat	68	47	62	34	78	RFs	72	48	65	35	73	RF	68	58	54	44	86	RF
	Heavy combo	78	46	56	31	93	RF	76	53	69	46	64	RFs	75	56	28	39	99	RF
	Best combo	82	61	73	59	63	RFs	76	53	69	46	64	RFs	75	62	69	63	63	RFs
		BoW + & Hist						LLM +/- & All						BoW +/- & Hist					
Closed1	Baseline	73	38	63	32	49	RFs	71	45	63	37	66	RFs	67	50	61	39	74	RFs
	Best text	74	38	62	31	48	RFs	72	47	66	36	70	RFs	65	55	55	42	79	RF
	Best feat	77	37	66	32	52	RF	75	48	69	35	67	RFs	69	56	64	43	78	RFs
	Heavy combo	78	35	66	26	56	RFs	75	50	68	40	74	RFs	70	53	64	43	70	RFs
	Best combo	79	41	67	32	52	RFs	75	50	68	40	74	RFs	70	59	64	45	80	RFs
		BoW +/- & Count						LLM +/- & All						LLM + & All					
Closed2	Baseline	73	33	59	27	38	RFs	70	41	63	37	56	RFs	66	48	61	36	74	RF
	Best text	77	32	58	24	42	RFs	70	41	61	30	61	RFs	72	51	65	40	72	RFs
	Best feat	81	37	69	29	56	RF	76	45	68	35	64	RFs	74	53	67	41	76	RFs
	Heavy combo	81	37	67	29	55	RFs	76	45	68	35	64	RFs	73	54	65	41	78	RFs
	Best combo	81	40	68	33	54	RFs	76	45	68	35	64	RFs	75	54	67	42	79	RFs
		BoW + & Count						LLM +/- & All						LLM + & All					

performs consistently better than alphanumeric ordering in all datasets and for different patch sizes. Specifically, Table 5.9 reports a Recall@50% of 71-90% for patches of size 5 to 9, and 69-78% for patches of size 10 and more. Compared to the alphanumeric order, this is an improvement of respectively +9 to +29% and +9 to +22%. When looking at only the top Recall@25%, our ordering still performs better than the alphanumeric ordering in all cases, improving the results from +13 to +50% for size 5 to 9, and +4 to +31% for patch sizes over 10.

In future work, we consider evaluating the use of this hot-spot prediction for file

Table 5.9: Ordering based on hot-spot prediction. We report the "Alph" (alphabetic) ordering, and "Ours" our ordering, based on hot-spot prediction. "%patch" reports the percentage of patches with the range of file changes. The blue cells indicate the best results by dataset and by metric.

Dataset	Size 2 to 4			Size 5 to 9					Size 10 and more				
	% patch	Rec@50% Alph	Ours	% patch	Rec@50% Alph	Ours	Rec@25% Alph	Ours	% patch	Rec@50% Alph	Ours	Rec@25% Alph	Ours
Base	30	72	81	7	72	81	55	70	3	67	78	60	64
Ironic	50	82	82	14	61	90	44	94	5	68	77	63	74
Nova	48	69	86	11	71	89	53	69	7	51	73	31	62
Closed1	38	61	78	18	55	71	45	64	12	52	69	38	59
Closed2	35	67	79	14	57	73	48	69	11	54	71	42	62

ordering in a set-up similar to the user study of Section 5.2, but with a larger scale and over a longer period, as well as gathering overall feedback of users on their impression of the new ordering.

5.7.2 Computation time

One of our motivations for evaluating embeddings is that past work reports that simpler approaches sometimes work better in the case of software engineering prediction tasks [40]. The use of feature-based approaches, and thus low dimensionality representations, has the advantage of low storage complexity and lower computational effort for training. Text embedding, on the other hand, depending on the approach, can be quite expensive to compute and store, and also increases the training effort. For the example of BoW, the size of the vocabulary explodes with the size of the corpus increasing. LLM encoding approaches usually embed the text into lower dimensions compared to BoW (e.g., around 2k for Bloom).

While our RQ1 results observed that no specific text-embedding stands out in terms of prediction performance, LLM embeddings report much longer embedding

times. In particular, LLM embeddings take .3 seconds per embedding, leading to 1-2 hours of embedding for open-source and 35-48 hours for closed-source datasets. In contrast, BoW embedding the full datasets could be done in 4-12 seconds for open-source and around 5 minutes for the closed-source projects. Even if two days of training might not be an issue if a model is kept up for 6 months, we did not perform any data drift analysis to find the optimal retraining frequency, and hence the potential impact of longer embedding time. Model training on the other hand takes longer with BoW, where the longest training reported for a period was 64 minutes for the BoW embedding and 12 minutes for the LLM embedding, thus 5 times shorter. We recommend being mindful of the effort needed for each type of encoding, since LLM embeddings require consequently more time overall.

5.8 Threats to Validity

Internal validity. Review environments can only capture so much, as offline review activities can happen as well. In the case of our industrial partner, developers have reported chatting via other tools or doing life reviews in person, which is not captured by their review environment. In this study, we thus only focus on what has been recorded in the review environments and aim at predicting those review activities.

Construct validity. We optimize the results around F1-score, as F1-score is a metric that captures both the recall and precision of the model, which represents best the model's performance with unbalanced datasets. Results could vary slightly if optimized around another metric.

We chose to evaluate the review activity task at file-level, as it is the granularity

that is necessary for the use case we considered, file-ordering. Future work could consider the impact of the granularity on the prediction, which is outside the scope of this chapter.

We used a zero-shot LLM-encoder as to reduce the computational effort. Fine-tuning the LLMs, though being computationally expensive, could improve the embedding and thus the results. However, past work has studied LLM fine-tuning for "commented" prediction [32] and reported similar results to other Bag-of-Words embeddings [13]. Larger LLMs might lead to better embeddings, even though the smaller `bloom-560m` has been reported as more performant than bigger counterparts in some other use-cases [160], hence it is not uncommon in research to leverage smaller models. As a compromise, we considered `bloom-560m` as a lower bound of the LLM embedding approach.

We worked on a per-project basis for embedding, training and evaluation, which is the default approach in this domain [32, 36, 40]. An early bootstrap option, when data is not yet available, might be to initially train on similar datasets (e.g., same coding languages).

External validity. The LLM field proposes newly trained/updated LLMs regularly. We recommend considering recent LLMs for text embeddings to stay on top of the newest findings in case of replication, which could yield better results. The version of Bloom we used is from May 2023.

Regarding our user study with 29 professionals, the participants only had to review two patches. A large-scale user study over a longer period would be necessary to evaluate the extent to which the hot-spot-based ordering approach can impact the reviews, especially when reviewers start to become aware of the underlying ranking

heuristics.

5.9 Conclusion

In this chapter, we explore different approaches for embedding code changes to predict review activities. This was motivated by the results of a user study involving 29 professional developers showing how ordering files in patches by prioritizing files that need review activity improves the review process. Indeed, we observe that such ordering approach lead to more review comments (+23%), that targeted parts needing revisions or comments (+13% precision and +8% recall), hence the need for an efficient review automation prediction approach. While previous work used a Bag-of-Word approach on the added lines and additional features (i.e., #lines added and #lines removed) [13], we propose variations to that embedding.

First, we improve the textual embedding part by considering embedding the removed lines as a contextual embedding and also considering LLM embeddings. Though no specific text embedding stands out in our empirical study on 3 open-source and 2 closed-source projects, adding any of our variations improved results compared to the BoW on the added lines in terms of F1-score in 13/15 cases (up to +5%). Then, we propose review process feature embeddings, using size- and history-based features. In practice, this performed at least as good as the textual embeddings in 10/15 cases regarding F1-score, while being less computationally expensive. Especially closed-source datasets does not show significant differences between text embeddings and the feature-based approaches. Finally, when combining the text and feature-based embeddings, our approach reports a median F1-score of 40-62%, improving significantly the performance with medium to large effect compared to the baseline in all

cases (up to +9% F1-score) with the best combination approach for all tasks.

This chapter was published as "An Empirical Study on Code Review Activity Prediction and Its Impact in Practice" at the ACM International Conference on the Foundations of Software Engineering (FSE) 2024, in collaboration with my co-authors, Sarra Habchi (Ubisoft) and Bram Adams (Queen's University).

5.10 Future work

We recommend future work on these types of tasks to consider our text embeddings and review process feature embeddings approaches, as they correlate to significant performance improvements. We observed that hot-spot prediction as file-ordering approach reports up to +29% recall in the first half of patches compared to the default alphanumeric ordering.

Furthermore, now that we can introduce file order based on hot-spot-prediction in review processes, reviewers still have the task of understanding the code and communicating properly the potential issues through review comments [6]. To assist them in this task as well, Chapter 6 explores LLM-based code review comment generation and its potential impact on reviewers via a large-scale user study.

Chapter 6

STUDY 4: Impact of code review comment generation

6.1 Introduction

Software code review is a core practice for modern software quality assurance [6, 161], widely adopted in industrial and open-source projects [143,162]. While, initially, code review mostly was a synonym for code inspections on a submitted patch (structured as a set of chunks, i.e., successive lines of modified code) [163, 164], the field gradually adopted a more dynamic approach to performing reviews commonly known as *modern code reviews* [165], embracing social dimensions, like facilitating knowledge transfer between developers and strengthening synergy within teams [166].

Despite such benefits, code reviews can also bring additional costs, due to the delay between a patch submission and its final approval for integration by the reviewers caused by back-and-forth between its author and reviewers [167]. Additionally, providing valid and effective reviews requires non-trivial efforts from reviewers in terms of technical, social, and personal aspects [168]. Reviewers need to understand and

rationalize the overall impact of the changes under analysis while using effective communication [169–171]. Reviewers, even when qualified and focused on a patch, might miss issues due to fatigue [172], distraction, or pressure from other deadlines [44].

To overcome these limitations, the research community has explored different approaches to generate review comments with LLM-based approaches such as T5, CodeT5 and ChatGPT [32, 36, 39]. However, thus far these have only been evaluated using textual metrics such as BLEU score and Exact Match, which are imperfect as they do not capture the actual impact and synergy the approach can have on reviewers [36]. Furthermore, it is unclear how reviewers receive LLM-generated comments in the long-term, i.e., apart from their relevance, to what extent do generated comments solicit follow-up comments by the patch author or lead to follow-up patch revisions?

To fill in this gap, we conducted the first large-scale user study involving two organizations, Mozilla and Ubisoft. Our choice for both an open-source (Mozilla) and closed-source (Ubisoft) organization is deliberate, since open-source codebases risk being part of the pretraining data of LLMs like GPT4o, yielding an unfair advantage in empirical studies. Hence, this study also aims at investigating whether open-source organizations actually benefit from this potential advantage compared to closed-source organizations.

Spanning over 6 weeks, our user study involved 59 reviewers, covered 587 patch reviews, and led to the evaluation of 1.6k generated comments. During this study, we monitored the reviewers' interactions with the generated comments, assessed the impact of these comments on the review flow, and finally conducted a survey of the participating reviewers to gain insight into their perspectives, filled by 37/59 participants.

For this evaluation, we built *RevMate*, an LLM-based review assistant that generates review comments and that is easy to integrate into modern review environments. RevMate builds on GPT4o and uses both (i) Retrieval Augmented Generation (RAG) [107] to enclose relevant information and ground the model on the project under analysis, and (ii) LLM-as-a-Judge [110] to leverage LLMs' capacity to evaluate generated content and discard irrelevant review comments. In particular, we exploit two different approaches for RevMate:

RevMate with extra code context (Code): inspired by Zhou et al.'s approach for code generation [108], the model can request a code retrieval tool to provide function definitions and additional code lines from the codebase under analysis.

RevMate with related comment examples (Example): like Parvez et al.'s approach for code summarization [109], for each patch, the model dynamically selects few-shot examples of review comments similar to its chunks, then provides them to the LLM.

This chapter addresses the following research questions:

RQ1: How frequently do reviewers accept comments generated by RevMate?

Based on the participants' feedback in both organizations, we found that 8.1% and 7.2% of comments are accepted by Mozilla and Ubisoft reviewers, respectively, with 23% and 28.3% other comments still appreciated as being helpful. Despite the organizations differences in data availability, both the acceptance and appreciation ratios were consistent among them, indicating that the results may generalize across closed- and open-source development processes. Across both organizations, 23/37 participants reported they would continue using RevMate at least sometimes outside of the study context.

RQ2: Do acceptance ratios depend on the type of generated review comment?

We found that, for Mozilla and Ubisoft respectively, 79.2% and 84.9% of generated comments are classified as functional issues, and 15.8% and 14.5% as refactoring-related comments. Furthermore, we found that refactoring-related comments have higher acceptance ratios than functional comments (18.2% and 18.6% compared to 4.8% and 5.2%).

RQ3: How does the adoption of LLM-generated comments impact the code review workflow?

We find that, although reviews take longer due to the extra time needed to evaluate the generated comments, the time spent per generated comment is reasonable (17s as a median). This time is mostly spent on investigating the validity of the comment and on editing the comment when applicable, as 37/119 accepted comments were edited, with the most popular type of edit being shortening the comment (25/37 cases).

RQ4: How does the adoption of LLM-generated comments impact the review outcome?

This RQ assesses the extent to which accepted generated comments lead to changes in the codebase. We found that accepted review comments lead to as many patch revisions (i.e., changes in a future version of the patch) as human comments (74% vs 73% at chunk level) in Ubisoft, indicating that the generated comments have the same impact as human comments. Also, generated comments lead to fewer follow-up comments among developers (23% compared to 34%).

The main contributions of this study are:

- A large-scale mixed closed-/open-source user study, with 57 expert reviewers,

assessing the usefulness of LLM-generated review comments and their impact on the review flow and outcome;

- A diverse set of 3.4k generated code review comments by GPT4o for the Mozilla organization, among which 426 were evaluated by reviewers [173];
- An open source project implementing the comment review generation approach [174].

6.2 User Study Design

Our goal is to evaluate the impact of integrating review comment generation into reviewers' workflow using a large-scale, mixed open-/closed-source user study. To do so, we developed a review assistant (*RevMate*) that integrates easily into common review environments. This section discusses the approaches used by RevMate and how the feedback of reviewers is gathered from the tool in the context of a user study.

6.2.1 RevMate's LLM-based approach

RevMate relies on the following common LLM-based mechanisms: RAG (tool-LLM collaboration for code retrieval) and LLM-as-a-Judge through LLM-based evaluation. The approach leverage the Chain-of-Thought architecture, by splitting the overall review comment generation process into multiple prompts, some of which (i.e., prompts (5) and (6) in Table 6.1) also use Chain-of-Thought internally. Furthermore, a memory conversation is used to tie the prompts together ((4) in Table 6.1). In the following, we will describe the resulting workflow, summarized in Algorithm 2.

Algorithm 2 RevMate Workflow. The (x) notation refers to the corresponding prompt in Table 6.1.

Input: patch, approach
Output: code_review

```
1: if patchNeedsReview(patch.status) then
2:    fpatch ← format(patch.sourcecode)
3:    sum ← askLLMforSummary(fpatch) (1)
4:    mem ← createMemory([sum]) (4)
5:    if approach = Code then
6:       funcs ← askLLMNeedsFuncs(fpatch) (2)
7:       lines ← askLLMNeedsLines(fpatch) (3)
8:       if funcs.size > 0 then
9:          context_funcs ← getContext(funcs)
10:         mem.addToMemory(context_funcs) (4)
11:      if lines.size > 0 then
12:         context_lines ← getContext(lines)
13:         mem.addToMemory(context_lines) (4)
14:      ex ← default_set_of_few_shot_examples
15:   if approach = Example then
16:      ex ← getSimilarExamples(fpatch)
17: first_review ← askLLMReview(fpatch, mem, ex) (5)
18: code_review ← askLLMFilterGeneration(fpatch, first_review) (6)
19: return code_review
```

Table 6.1: Prompts

(1) Summary prompt (*askLLMforSummary*):
You are an expert reviewer for source code, with experience in source code reviews. Please, analyze the code provided and report a summarization about the new changes; for that, focus on the coded added represented by lines that start with "+". {*patch*}
(2) Function request prompt (*askLLMRequestFunctions*):
Based on the patch provided below and its related summarization, identify the functions you need to examine for reviewing the patch. List the names of these functions, providing only the function names, with each name on a separate line. Avoid using list indicators such as hyphens or numbers. If no function declaration is required, just return "". {*patch*} {*summarization*}
(3) Line context request prompt (*askLLMRequestLines*):

Based on the patch provided below and its related summarization, report the code lines for which more context is required. For that, list the lines with the their associated line numbers, grouping each one on a separate line. Avoid using list indicators such as hyphens or numbers. If no code line is required, just return "". Examples of valid code lines: '152 const selector = notification.getDescription();' and '56 file.getElement(this.targetElement());' {*patch*} {*summarization*}
(4) Memory conversation content (*memory*):
in: You are an expert reviewer for source code, with experience on source code reviews. **out:** Sure, I'm aware of source code practices in the development community. **in:** Please, analyze the code provided and report a summarization about the new changes; for that, focus on the code added represented by lines that start with "+". {*patch*} **out:** {*summarization*} if requested context: **in:** Attached, you can find some function definitions that are used in the current patch and might be useful to you, by giving more context about the code under analysis. {*function context*} **out:** Okay, I will consider the provided function definitions as additional context to the given patch. **in:** Attached, you can also have more context of the target code under analysis. {*line context*} **out:** Okay, I will also consider the code as additional context to the given patch.
(5) Review generation prompt (*askLLMReview*):
You will be given a task to generate a code review for the patch below. Use the following steps to solve it: 1. Understand the changes done in the patch by reasoning about the summarization as previously reported. 2. Identify possible code snippets that might result in possible bugs, major readability regressions, and similar concerns. 3. Reason about each identified problem to make sure they are valid. Have in mind, your review must be consistent with the source code. 4. Filter out comments that focuses on documentation, comments, error handling, tests, and confirmation whether objects, methods and files exist or not. 5. Filter out comments that are descriptive and filter out comments that are praising (example: "This is a good addition to the code."). 6. Filter out comments that are not about added lines (have '+' symbol at the start of the line). 7. Final answer: Write down the comments and report them using the JSON format previously adopted for the valid comment examples. As valid comments, consider the examples below: {*ex*} Here is the patch that we need you to review: {*patch*}
(6) Review generation filtering prompt (*askLLMFilter*):

Please, double check the code review provided for the patch below. Just report the comments that are: - applicable for the patch;
- consistent with the source code;
- focusing on reporting possible bugs, major readability regressions, or similar concerns;
- filter out any descriptive comments;
- filter out any praising comments;
- filter out any comments that do not suggest code changes.
Do not change the contents of the comments and the report format. Adopt the template below as the report format:
[{ "file": "com/br/main/Pressure.java", "code_line": 458, "comment" : "a generated comment" }]
Do not report any explaination about your choice.
Review: {*review*}
Patch: {*patch*}
As examples of not expected comments, not related to the current patch, please, check some below:
{*undesired_comments*}

Starting Point

RevMate is automatically called when a reviewer opens a patch in their review environment. Only patches labeled as *Needs Review* are analyzed (line 1, Algorithm 2). Review comments are generated once per patch and kept in cache to maintain consistency if reviewers reload the page. Moreover, a reviewer will only be able to see any previously generated comments that have not yet been evaluated by another reviewer.

RevMate re-formats the patch to help the LLM understand the patch's content (line 2, Algorithm 2). Knowing that a patch consists of one or more chunks (i.e., a sequence of changed lines), RevMate groups a patch's chunks per file while also adding the corresponding file line number to the beginning of each code line (instead of Git's default line number-range per chunk). This allows the LLM to better associate the generated review comments with the target lines.

Next, RevMate requests the LLM to summarize the formatted patch (line 3, Algorithm 2; prompt (1) in Table 6.1), using the LLM's ability to self-explain code [105, 175, 176].

Extra context

After generating the summary, RevMate seeks extra context to ground the model on the codebase under analysis and improve the quality of the generated comments. Building on the RAG mechanism, we consider two ways of providing extra context for comment generation, i.e., ***Code*** and ***Example***. Although both approaches are complementary, in our user study on the impact of generated comments on the review process, we only considered one approach at a time.

Code context (Code) Being aware that the patch under analysis might not contain sufficient details to perform a full analysis, we ask the LLM whether there are function calls referred by the patch that require the full function definition to understand its use (line 6, Algorithm 2). For that, we feed the patch and its associated summary to the model, and request a list of the needed functions' names (prompt (2), Table 6.1). Then, for each function name, a code retrieval tool accesses the respective project repository to retrieve the function's definition [177] (line 9, Algorithm 2) to provide to the model via the memory (line 10, Algorithm 2; conversation (4), Table 6.1).

Besides functions' definitions, other code context can also be needed for the model's review generation. In particular, the extracted code chunks only show up to 10 lines of context (i.e., unmodified lines above or below modified lines). This too might lead to missing context and hinder patch understanding. Thus, we also ask the

LLM whether it requires additional context for specific patch lines (line 7, Algorithm 2; prompt (3), Table 6.1), this time getting extended context around the requested lines by retrieving the functions that include said lines (line 12, Algorithm 2), then providing it to the memory (line 13, Algorithm 2; conversation (4), Table 6.1).

Finally, when using the *Code* approach, the comment generation prompt (prompt (5), Table 6.1) will be provided with a fixed list of few-shot examples ex randomly selected from past examples of chunks with their associated review comments (line 14, Algorithm 2). This default set is available in the replication package [174].

Related comment examples (Example) The *Example* approach improves on the default set of examples provided by the *Code* approach, by explicitly selecting relevant examples chosen based on their similarity to the current patch (line 16, Algorithm 2). To do so, we first constructed a dataset for each studied project comprising past chunks and their associated comments (tuples of $(chunk, com)$), excluding tuples where the comment had more than 500 characters (to encourage concise comments only), or comments that included urls (which the model does not have access to). The examples are loaded in a vector database [178] with GPT embeddings. Examples are retrieved from this database using the cosine distance between the embeddings of the current patch's chunks and the historical ones stored in the vector database. We retrieve 10 examples for each chunk in the current patch, then report the 10 examples with highest similarity out of all retrieved examples for the patch.

Generating code reviews

The information retrieved (i.e., summary and extra context) is sequentially structured into a memory conversation buffer ((4) in Table 6.1), which is responsible for

providing the task context, including code context for the *Code* approach and relevant examples for the *Example* approach. First, we set up the LLM to assume the persona of an expert reviewer. Second, we feed the patch summary, providing context for the changes under analysis (line 4, Algorithm 2). If functions and context lines are provided by the code retrieval tool based on previous steps, they are also added to the buffer (lines 10&13, Algorithm 2).

Finally, RevMate asks the model to perform a code review (line 17, Algorithm 2; prompt (5) in Table 6.1): we instruct it to follow a *few-shot* approach by providing examples of valid comments, either the default set or the retrieved set, aiming to guide the LLM to provide related and relevant content [103, 104]. This way, we impose a standard pattern for the comments, consisting of the comment, associated line number, and file ($\{com; line; file\}$).

Filtering generated comments

Once the LLM has generated the comments, RevMate makes one final LLM-request in another conversation (i.e., without memory) to filter out comments that do not address valid problems. Leveraging the concept of LLM-as-a-Judge [110], RevMate prompts a filtering model providing the list of generated comments and the associated patch as input (line 18, Algorithm 2; prompt (6) shown in Table 6.1). The model is also provided with a list of undesired comment examples (*undesired_comments*, which we constructed from preliminary feedback from a early users at Mozilla and iterations with the prompt) to guide the filtering process, which are available in our replication package [174]. Last, RevMate guides the LLM to report its results in JSON format (i.e., list of $\{com; line; file\}$).

With the final list of comments generated (line 19, Algorithm 2), RevMate stores them in a database server. Based on the reported line numbers and file names associated with each comment, RevMate shows the generated comments directly on the patch under analysis in the review environment.

Technical Details

After initial experiments, we decided on using GPT4o with a temperature of .2 to provide more accurate and deterministic results compared to higher temperatures, following Lin et al. [179]. For the embeddings used for the *Example* approach, we used GPT text-embedding-3-large also with a .2 temperature. However, RevMate has been built for ease of extensibility and could be easily adapted to work with different LLMs and under different configurations.

6.2.2 UI for RevMate's generated comments

To evaluate the benefits and trade-offs associated with adopting RevMate, we integrated the tool into participants' review environment, making it available to them when performing their day-to-day code review tasks. Through the user study, reviewers can evaluate each generated comment, which is presented as a suggestion through an evaluation box shown in Figure 6.1: "add to comment" to *accept* it or "ignore" to *reject* it. In case of rejection, we additionally asked reviewers to provide a reason for ignoring comments, giving the options shown in Figure 6.2. Before evaluating the comment, reviewers can also edit its content. Once evaluated, the evaluation box disappears, leaving no trace in the case of rejection and leaving a publicly published comment in the case of acceptance. Each generated comment can only be evaluated

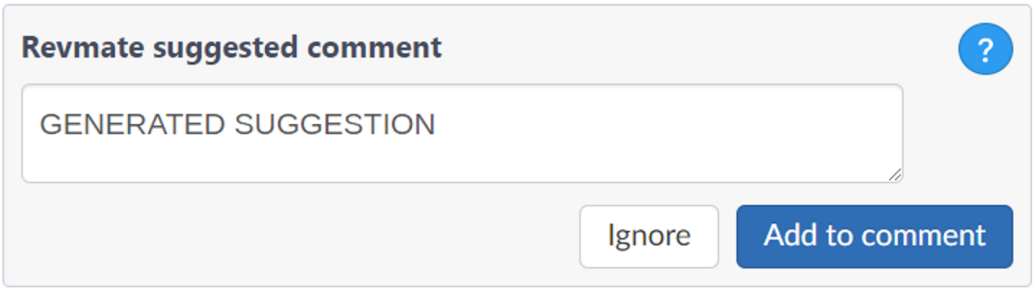


Figure 6.1: UI for generated comments' evaluation boxes

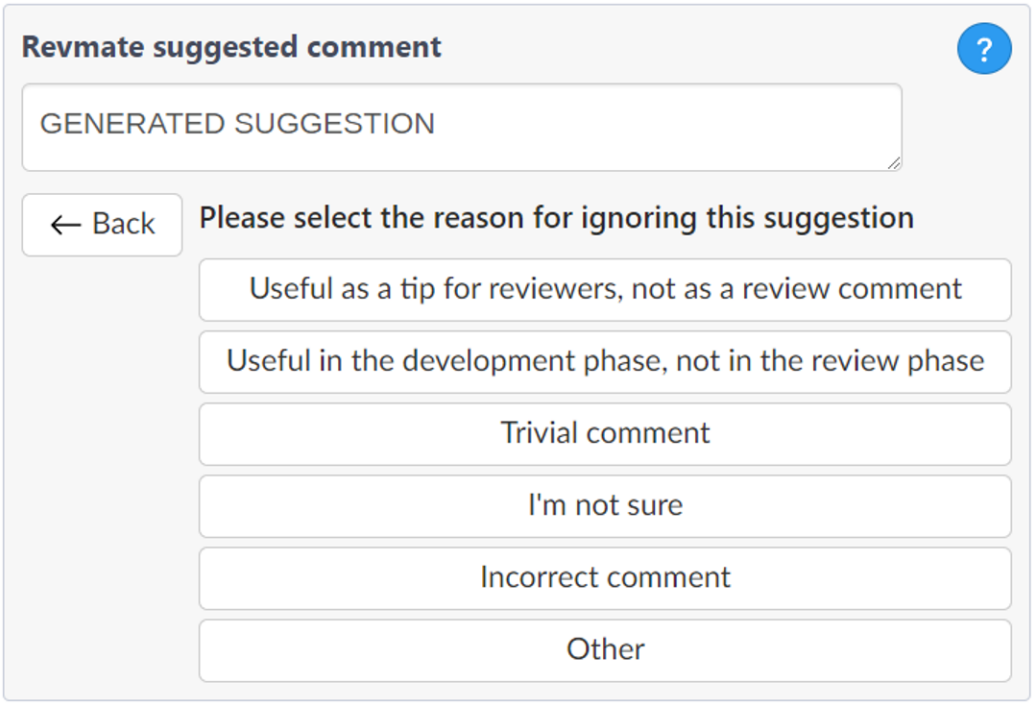


Figure 6.2: UI for choice of reasons for ignoring generated comments

once.

As the participating organizations use two different review environments (Mozilla uses Phabricator[1], while Ubisoft uses Swarm[2]), we also had to adapt RevMate to follow each organization's specific policies regarding reviews and generative-AI. This

[1]Phabricator, accessed: 2024-10-15. secure.phabricator.com/
[2]Swarm, accessed: 2024-10-15. www.perforce.com/products/helix-swarm

led to the following differences in design choices:

Mozilla: Accepted generated review comments had to be published as a separate reviewer named "*RevMate*". The tool shows the generated comments' evaluation boxes directly when opening a review. In order to nudge the user study participants towards evaluating all review comments, the UI delays the publication of accepted generated comments to the patch author until all generated comments have been evaluated.

Ubisoft: The reviewers had to publish the generated review comments that they accepted in their own name. Furthermore, due to the review environment design and API, the generated comments could not be displayed immediately when opening the page. To mark where generated comments were, we added a star symbol next to the lines on which reviewers needed to click for it to appear. To further help reviewers with finding the generated comments, for each file, we indicated next to the filename the comments' line numbers.

For both organizations, a summary at the beginning of the patch's page tells users how many generated comments were generated and need to be evaluated.

6.2.3 User study setup

We now discuss the setup of the user study that we designed to address the research questions listed in the introduction.

Participants and Target Projects

Since our main goal is to evaluate the real-world potential of adopting LLMs as

review assistant, we randomly contacted 50+ participants actively performing code reviews in ongoing projects in each organization. The targeted projects all are mature projects that are continuously updated and written in different programming languages (e.g., C++, C#).

As the contacted users joined on a voluntary base, we did not have control on the demographics. However, we can report that Mozilla had 28 participants with an average experience of 8.5 years inside the organization (from 1-13 years) and 13.4 years in their career (from 5-25 years), while Ubisoft had 31 participants, with average experience of 7.6 years inside the organization (from 1-25 years) and 10.9 years in their career (from 4-25 years).

Before starting the study, a manual with instructions to install and run the tool in their specific review environment was sent to them.

Running the Study

Participants of the study within each organization are randomly split into two groups: one group gets the *Code* approach and the other gets the *Example* approach (see Section 6.1). The differences between the two approaches are studied in RQ1.

For practical reasons, the study was conducted during 5 weeks at Mozilla and 6 at Ubisoft, covering respectively 165 and 422 code reviews assisted by RevMate. The set of all generated reviews from Mozilla can be found in our online Appendix [173], whereas Ubisoft's data has to remain closed-source.

Table 6.2: End of study survey questions

#	**Question**
1	What is your experience as a developer in the company? (open)
2	What is your experience as a developer in your carreer? (open)
3	When using the tool, how was the duration of your reviews affected? Slower - A bit slower - Same time - A bit quicker - Quicker
4	What would you change about the suggestions' content, if anything? (open)
5	Would you use a review tool like this tool during your review processes outside of the study? Never - Rarely - Sometimes - Often - Always

6.2.4 Data Collection

For each generated review comment, we gathered the following information: comment identifier, line number, filename, patch identifier, reviewer identifier, timestamp of evaluation, content of the generated comment, edited content (if applicable), evaluation (i.e., accept or ignore), reason (choice from list in Figure 6.2). In the case of Ubisoft, we also have the timestamp on which the generated comment is opened, as users had to open them to read then evaluate them, which we used for RQ3.

After having access to the tool for a few weeks, we sent a survey to users to get their general feedback on the generative tool. The questions can be found in Table 6.2. Mozilla received responses from 15/28 participants, while Ubisoft received responses from 22/31 participants, for a total of 37 responses.

6.2.5 Metrics and statistical tests

The accepted comments obviously represent high-quality generated comments, as they were adopted by reviewers. Hence, we analyze the acceptance ratio, $\frac{\#Accepted}{\#Evaluated}$, for both organizations. However, among comments that were not accepted, some rejection motivations still indicate a positive impact for comments found to be useful

as a tip for reviewers or during the development phase, which we will collectively refer to as "Valuable Tips" ($\#ValuableTip$). We thus also report a second ratio, i.e., the appreciation ratio, $\frac{\#Accepted+\#ValuableTip}{\#Evaluated}$.

To compare proportions in our analysis, we use Fisher's statistical test, with $\alpha = .05$. When the difference is statistically significant, we then use Cohen's d score (CD) to identify the effect [180], with effect size small (>.2), medium (>.5), or large (>.8).

6.3 RQ1: How frequently do reviewers accept comments generated by RevMate?

6.3.1 Approach

In Table 6.3, we report numbers about the participants' evaluation of generated comments and, when applicable, their reasons for ignoring LLM-generated comments. Even though participants were asked to evaluate all comments, we gave them the option to indicate cases where they felt unable to perform an evaluation (option "Ignored - I'm not sure"). When considering all evaluated comments ($\#Evaluated$ in the ratios), we exclude those comments.

It should also be noted that due to differences in the review environment UI in the two organizations (see Section 6.2.2), reviewers from Ubisoft were not forced to evaluate the full set of generated comments per patch, but were encouraged to do so, contrary to Mozilla that enforced users to do full evaluations. This leads to comments that were seen but not evaluated in the case of Ubisoft, which we consider as ignored with no selected reason. We marked those as "Ignore - Seen". Acceptance and appreciation ratios are reported in Table 6.4.

Table 6.3: RQ1 - Evaluation counts. "Total" refers to the union of all the generated comments, disregarding the approach used.

Macro-Evaluation	**Evaluation**	**Mozilla**		
		Approach		**Total**
		Code	Example	
Accepted	Accept	9 (6.8%)	25 (8.5%)	34 (8.0%)
Valuable Tip	Ignore - Useful as a tip for reviewers	19 (14.4%)	20 (6.8%)	39 (9.2%)
	Ignore - Useful for developer	10 (7.6%)	13 (4.4%)	23 (5.4%)
Rejected	Ignore - Trivial comment	37 (28.0%)	114 (38.8%)	151 (35.4%)
	Ignore - Incorrect comment	42 (31.8%)	100 (34.0%)	142 (33.3%)
	Ignore - Other	10 (7.6%)	19 (6.5%)	29 (6.8%)
	Ignore - Seen	/	/	/
Filter out	Ignore - I'm not sure	5 (3.8%)	3 (1.0%)	8 (1.9%)
	Total #comments	132	294	426
	Total #evaluated	127	291	418
	Total #reviewers	19	22	28
	Total #reviews	59	106	165

Macro-Evaluation	**Evaluation**	**Ubisoft**		
		Approach		**Total**
		Code	Example	
Accepted	Accept	31 (5.3%)	54 (8.5%)	85 (7.0%)
Valuable Tip	Ignore - Useful as a tip for reviewers	51 (8.7%)	43 (6.8%)	94 (7.7%)
	Ignore - Useful for developer	105 (17.9%)	51 (8.1%)	156 (12.8%)
Rejected	Ignore - Trivial comment	112 (19.1%)	66 (10.4%)	178 (14.6%)
	Ignore - Incorrect comment	47 (8.0%)	137 (21.7%)	184 (15.1%)
	Ignore - Other	28 (4.8%)	50 (7.9%)	78 (6.4%)
	Ignore - Seen	206 (35.2%)	204 (32.3%)	410 (33.7%)
Filter out	Ignore - I'm not sure	6 (1.0%)	27 (4.3%)	33 (2.7%)
	Total #comments	586	632	1218
	Total #evaluated	580	605	1185
	Total #reviewers	17	14	31
	Total #reviews	197	227	422

Table 6.4: RQ1- Evaluation ratios. "Total" refers to the union of all the generated comments, disregarding the approach used.

Ratio	**Mozilla**			**Ubisoft**		
	Approach		**Total**	**Approach**		**Total**
	Code	Ex		Code	Ex	
Acceptance [%]	7.1	8.6	8.1	5.3	8.9	7.2
Appreciation [%]	29.9	19.9	23.0	32.2	24.5	28.3

6.3.2 Results

8.1% and 7.2% of generated comments were accepted, for Mozilla and Ubisoft respectively, as can be seen in Table 6.4. For Ubisoft, the *Example* approach reports a significantly better acceptance ratio (8.9% vs 5.3%) with a small

effect size compared to the *Code* approach (p-val=.011). For Mozilla, the *Example* approach still reports a higher acceptance ratio (8.6% vs 7.1%), although the difference is not significant (p-val=.381). We suspect that the *Example* approach has the advantage of showing comments more similar to what users are used to seeing and publishing themselves, as the few-shot examples shown by the *Example* approach are taken from past similar published cases.

23% and 28.3% generated comments, comprising both accepted comments and valuable tips, are appreciated at Mozilla and Ubisoft respectively. Table 6.4 shows how, this time, the *Code* approach reports a significantly higher appreciation (medium effect size) than the *Example* approach, for both organizations (respectively, 29.9% vs 19.9% with p-val=.022 and 32.2% vs 24.5% with p-val=.002). This difference could indicate that the context related to the change provided by the *Code* approach helps the model generate more informed and accurate comments. Yet, as the few-shot examples provided by this approach are quite limited compared to the *Example* approach, and do not adapt to the precise style of comments required, the resulting comments do not seem to make it as publishable (accepted) review comments.

Overall, both organizations report similar results, despite their differences in codebase availability, development and review processes. As indicated in Section 6.2.3, Mozilla has the advantage that its codebase is open-source, with modern LLMs potentially having acquired knowledge about Mozilla's codebase through pre-training. Yet, we do not observe any significant differences in results when comparing against each other both organizations, for any combination of ratios (i.e., acceptance and appreciation) and sets (i.e., Code, Example or Total) (p-value=.279

to .986).

23/37 reviewers across both organizations reported that they would use a review assistive tool like RevMate at least sometimes outside of the study (question 5, Table 6.2). To analyze this closer, two authors aggregated the participants' mentioned concerns from the open answers to question 4 of the survey, which asked users what they would change about the comments' content. Overall, 20/37 participants explicitly mentioned that the relevancy of comments should be improved, as most of the suggested comments are "*accurate*", but "*superficial*" , "*obvious*", "*generic*", "*vague*", or sometimes even "*wrong*", as also confirmed by the numbers for "Ignore - Trivial comment" and "Ignore - Incorrect comment" in Table 6.3. Yet, one participant mentioned that "*the tool can be super useful for less experienced reviewers and a good way for them to learn*", confirmed by 2 other users. Also, 6/37 mentioned being optimistic on the future of LLM-based approaches for review assistance tasks, and 11/37 mentioned that, at least, the tool was able to guide their review (i.e., providing a tip for reviewers).

RQ1: Generated review comments achieve promising acceptance (8.1% and 7.2%) and appreciation ratios (23% and 28.3%). 23/37 reviewers that answered our survey reported that they would continue using RevMate at least sometimes.

6.4 RQ2: Do acceptance ratios depend on the type of generated review comment?

6.4.1 Approach

To automatically categorize the generated comments, we clustered them according to their embeddings, then used an LLM to synthesize a category label for each cluster based on the comments that it contains [181]. The categories we considered are the general categories presented by Turzo et al. [4]: functional, refactoring, documentation, and discussion.

First, comments are gathered into clusters using Kmeans [182] on the embedded comment (using GPT's text-embedding-3-large). Then, we ask the LLM to label each cluster of comments with a descriptive label, by providing it with a list of label examples of the form "{*general_category*} - {*specific_category*}*:* {*description*}", corresponding to the table of categories presented by Turzo et al. [4]. Given those examples, we observed that often, multiple clusters were labeled with the same label, which led us to eventually merge the 363/400 concerned clusters into the final 37 clusters shown in Table 6.5. Also, not all generated labels existed in the original list of examples we provided, which we indicate by a star "*" in Table 6.5. We ran the LLM-base labeling 5 times and report for each comment the label that got the majority vote.

In order to determine the optimal number of clusters, we compared, for the number of clusters ranging from 2 to 400, the *general_category* of each cluster with that of a manually labeled ground truth, which we discuss below. We found that the percentage of matches with the ground truth converged between 300-400 clusters, leading to

76.3% of matches (171/244) for 400 clusters, with only 37 distinct labels being generated even with that amount of clusters. To evaluate the agreement between the model and the ground truth (human labeling), we used Cohen's kappa score, a metric between -1 and +1 where +1 indicates total agreement and -1 total disagreement [183]. We obtained a kappa score of .45, which is considered as moderate but acceptable agreement.

Table 6.5: RQ2 - For each automatically generated comment category, we list the number of comments from each Organization (Moz and Ubi) that were labeled as such. "*" indicates that the label was an example in the few-shot prompt [4].

#	Label	**Moz.**	**Ubi.**
1	Discussion - Design discussion *	3	3
2	Discussion - Question *	7	1
3	Documentation *	11	4
4	Functional - Conditional Compilation	2	11
5	Functional - Consistency and Thread Safety	1	0
6	Functional - Error Handling	4	2
7	Functional - Exception Handling	0	2
8	Functional - Initialization	1	8
9	Functional - Interface *	81	91
10	Functional - Lambda Usage	2	3
11	Functional - Logical *	74	272
12	Functional - Null Handling	0	3
13	Functional - Performance	3	18
14	Functional - Performance Optimization	0	1
15	Functional - Performance and Safety	0	1
16	Functional - Resource *	58	135
17	Functional - Security	1	0
18	Functional - Serialization	0	4
19	Functional - Support *	3	2
20	Functional - Syntax	3	2

21	Functional - Timing *	3	56
22	Functional - Type Safety	0	2
23	Functional - Validation *	95	392
24	Refactoring - Alternate Output *	11	6
25	Refactoring - Code Duplication	0	7
26	Refactoring - Code Simplification	0	2
27	Refactoring - Consistency	0	1
28	Refactoring - Magic Numbers	0	1
29	Refactoring - Naming Convention *	15	24
30	Refactoring - Organization of the code *	10	79
31	Refactoring - Performance Optimization	0	6
32	Refactoring - Readability	3	11
33	Refactoring - Simplification	0	3
34	Refactoring - Solution approach *	26	27
35	Refactoring - Unused Variables	0	1
36	Refactoring - Variable Declarations	0	1
37	Refactoring - Visual Representation	1	3

To obtain the manually labeled ground truth, used to determine the optimal number of clusters, we sampled from each of the macro-evaluation categories of RQ1 (i.e., accepted, valuable tip, and rejected) 83 generated comments (70% of #accepted, the ones available at the time of the analysis), in order to have a balanced set of each type, for a total of 249 comments. Then, two authors manually labeled each sample with a comment category using Turzo et al.'s main categories [4]. We did not consider their "false positive" category, as we did not evaluate the validity of the comment, only its category. In the end, in 93% of the cases (230/249) the authors agreed on the selection of one out of the four possible categories. The kappa score between both authors' labeling of .83 is considered as almost perfect agreement. For any conflicts, the two authors performing the labeling discussed and eventually agreed on

the category using the *negotiating agreement* technique [184], but 5 comments were left unlabeled because of ambiguity in the comments themselves. The latter were discarded, leaving us with 244 labeled comments.

For the selected number of clusters (400) and resulting cluster labels, we now split comments into groups according to the generated labels' *general_category*, shown in Table 6.6, as well as their acceptance and appreciation ratios.

Table 6.6: RQ2 - Categorization of generated comment and associated appreciation and acceptance.

Organization	**Mozilla**				**Ubisoft**			
Category / **Metric**	***Functional***	***Refactoring***	***Documentation***	***Discussion***	***Functional***	***Refactoring***	***Documentation***	***Discussion***
Count [#]	331	66	11	10	1005	172	4	4
%Total [%]	79.2	15.8	2.6	2.4	84.8	14.5	.34	.34
Acceptance [%]	4.8	18.2	-	-	5.2	18.6	-	-
Appreciation [%]	19.0	36.4	-	-	29.0	25.0	-	-

6.4.2 Results

Table 6.6 shows that Functional and Refactoring comments are the most commonly generated comments, with respectively 79.2% and 15.8% for Mozilla and 84.8% and 14.5% for Ubisoft. Notably, Table 6.5 shows that the most popular sub-categories of Functional comments are "Validation", "Logical", "Resource" and "Interface", and "Organization of the code" and "Solution approach" in the case of Refactoring comments. Those subcategories were proposed through the few-shot examples provided to the LLM, from Turzo et al. [4]. The Documentation

and Discussion labels are the rarest, with a total of 21/418 comments for Mozilla and 8/1,185 for Ubisoft. This is expected, as the prompt "*AskLLMReview*" (see prompt (5) in Table 6.1) suggests that such comments are not desirable, which is also why we did not report their acceptance and appreciation ratios in Table 6.6.

Refactoring comments are accepted more than functional comments (18.2% vs 4.8% for Mozilla and 18.6% vs 5.2% for Ubisoft). This difference is statistically significant with large effect size, with respectively p-values .0 and .002.

In Mozilla, appreciation for refactoring comments is significantly higher than for functional comments (36.4% vs 19%, p-val=.002, large effect size). In the case of Ubisoft, the difference is not significant (p-val=.877), although in practice Functional comments are more appreciated (29% vs 25%).

RQ2: Refactoring comments report a significantly higher acceptance ratio than Functional comments in most cases (18.2 vs 4.8% for Mozilla and 18.6 vs 5.2% for Ubisoft). Discussion and Documentation comments are rare (29/1,603 comments), which is expected based on our prompt design.

6.5 RQ3: How does the adoption of LLM-generated comments impact the code review workflow?

6.5.1 Approach

In question 3 of the survey (Table 6.2), we asked participants whether interacting with the generated comments impacted the duration of their reviews. In the case of Ubisoft, we also captured the time it took to evaluate the generated comments in order to empirically estimate that time. This was possible due to the difference in design

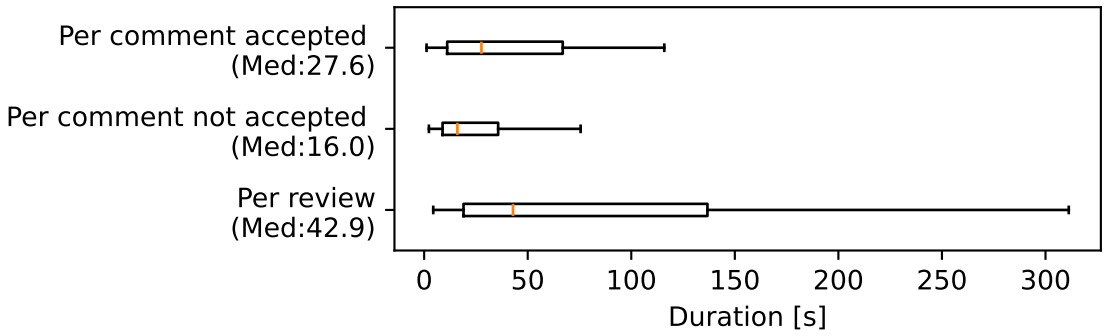


Figure 6.3: RQ3 - Evaluation duration. Outliers are hidden for visibility.

(see Section 6.2.2): for each generated comment, we marked the timestamp of when a given comment was first opened (not applicable to Mozilla) and of the timestamp when the reviewer submitted their evaluation of this comment. The difference between those timestamps provides an upper bound on the actual time it took a reviewer to evaluate a generated comment, as this period is longer than the actual time taken for evaluation due to switching to other comments or distractions. In Figure 6.3's boxplots, we report the distribution of the duration of evaluation per comment and per patch review (i.e., total duration for all its generated comments' evaluations).

We also evaluate how reviewers edit the accepted review comments. In Table 6.7, we report four different types of edits on accepted comments:

- As-is: with no modification (i.e., exact match between generated and published comment);
- Shorten: the published comment is shorter than the generated one;
- Extended: the generated comment is included as-is in the published one, but more text was added around it;

Table 6.7: RQ3 - Editing of accepted comments

Evaluation	Mozilla	Ubisoft
Suggestion as-is	27	55
Shortened content	6	19
Additional content	1	4
Other modification	0	7

- Other modification: any modification not included above.

6.5.2 Results

The median time spent at Ubisoft on the evaluation of individual comments is 27.6s for accepted comments and 16s for others, as shown in Figure 6.3. By 24/37 users in the survey, the additional effort of evaluating comments was reported as, to some extent, slowing down the review process. Our quantitative analysis of the comment timestamps showed that this slowdown is relatively limited, falling within a 95% confidence interval of [1s,116s] (median of 27.6s) for accepted comments, and [1s,76s] (median of 16s) for others. Although time spent on ignored comments might be considered "wasted time" by reviewers, in some cases the comments might still inspire the reviewer to comment on the code about problems they would have missed otherwise, or at least force an extra verification of the code. We did not find any significant difference in the median time spent evaluating appreciated comments vs fully rejected comments, hence did not include the former in Fig. 6.3.

At the review-level, the median total time spent on generated comments is 42.9sec (95% confidence interval [5s,5min12s]). This effort is considered by many users as acceptable. For instance, one of the reviewers commented: *"Often looking at the suggestion helps me think more about the code and reflect a bit more on*

what is done and why it is done. So overall, I think it is a bit slower to do reviews, but my reviews are actually better now that I use the tool". Similar comments were made by 11/37 users. The extra time spent on the generated comments also comes from either verifying the validity of the comment or, specifically for the accepted ones, modifying the comments.

27/34 accepted comments were published as-is for Mozilla, 55/85 for Ubisoft. The most common type of edit to generated comments is to shorten the comment (6/34 for Mozilla, 19/85 for Ubisoft), which suggests that generated comments can often be too wordy, which was confirmed in the survey by three participants. In future work, we want to iterate on the prompt to make generated comments more succinct.

RQ3: 24/37 surveyed reviewers report that review duration is increased by our tool. We quantitatively measured that the median time dedicated to the inspection of generated comments at the review level is 43s in Ubisoft. In both organizations, most comments, when accepted, are mostly published as-is (82/119) or shortened (25/119).

6.6 RQ4: How does the adoption of LLM-generated comments impact the review outcome?

6.6.1 Approach

To evaluate the impact of accepted comments on the further processing of the patch, we captured, for each line commented on via an accepted generated comment, whether that comment led to threads of review comments and/or follow-up revisions.

A potential impact of the comment would be any follow-up review comments, i.e.,

threads of comments, made by the author of the patch or other reviewers in response to the posted comment to (1) better understand the needed change, (2) argue about the need for adjusted revisions suggested or (3) acknowledge the actions taken for the requested change. It should be noted that the follow-up revisions might not be directly related to the accepted comments, but at least this characteristic provides a reasonable approximation of the potential impact the comment may have had on the evaluation of the commented patch, in both generated comments and human comments from patch reviews outside of the user study.

For the analysis of the impact of generated comments on later patch revisions and/or review discussions, we selected comments based on the following constraints. First, we only consider accepted comments. Then, we restricted the selection to patches with "finished" review status, since some reviews were still ongoing at the time of the study, thus the comment might not yet have been dealt with. Finally, we considered only results from Ubisoft, which represent 69 comments satisfying the selection criteria, as Mozilla had only 25 cases after filtering, which would not have led to significant comparison.

As a baseline for our analysis on generated comments, we also collected a dataset of "Human" comments, by other reviewers not part of the user study, sampled randomly from patches under review in Ubisoft but that were not included in the actual user study, yielding a total of 1,211 comments.

We report in Table 6.8 three different types of impact that accepted comments can exhibit:

- Revised line: The comment's line was modified in a follow-up patch revision.
- Revised chunk: The comment's line was modified or was part of the context in

Table 6.8: RQ4 - Impact of generated comments on finished patch reviews. "Generated" refers to accepted generated comment. "Human" refers to manually written comments on patches without generated comments.

Evaluation	**Ubisoft**			
	Generated		**Human**	
Revised line	43	62.3%	779	64.3%
Revised hunk	51	73.9%	887	73.2%
Thread	16	23.2%	411	33.9%
Total	69		1211	

a follow-up patch revision.

- Thread: There was a comment in response to it.

6.6.2 Results

In Ubisoft, generated comments lead to as many revisions on the line (62.3% and 64.3%) and chunk level (73.9% and 73.2%) as human comments do. No significant difference is found in revision ratios on either levels (p-val=.413 and p-val=.515, respectively).

For Ubisoft, significantly less comment threads were started on generated comments (23.2%) compared to human comments (33.9%). The difference is statistically significant (p-val=.041) with medium effect size. This is expected, as the generated comments are not meant to start discussions, whereas human comments can often, for instance, be phrased as genuine questions requesting a response. This expands on our findings from RQ2 that most comments are not of the Discussions category. However, from manual inspection of the concerned comments, in practice the threads tend to discuss the generated comment and/or to acknowledge the change (i.e., writing "Done") or reject it.

RQ4: Generated comments experienced as many revisions as human comments (73.9 vs 73.2% at chunk-level), but less follow-up comments (23.2 vs 33.9%).

6.7 Discussion

Evaluating review comment generation is a challenging task, especially as it is essential to capture the impact of assisting reviewers in their work. Where prior work has addressed the question of review comment generation by textually comparing the generated outputs with a ground truth of past chunks commented on by humans, such metrics do not evaluate the impact the generation has on the reviewers' workflow. In this chapter, we instead carried out a user study to gather feedback from reviewers in two software development organizations.

The first metric we report is acceptance, i.e., the ratio at which reviewers actually accepted comments for publication. We observed that reviewers, even if they ignore the comments, often still report them as having value (14.6% and 20.5%).

First, reviewers reported that 5.4% and 12.8% comments, respectively, could actually be more valuable for developers before they submit their patch. Those generated comments are thus shown too late in the development process. One participant reported in the survey "*It detects perfectly possible flaws in the code, but most of the suggestions are more in the development realm than the review*", with 4 participants making similar comments. That reason for ignoring LLM-generated comments was used by 28/57 reviewers.

Second, although some comments are not actionable as-is or not mentioning specific problems, they often (9.2% and 7.7%) raise interesting questions "*that force to*

look around, and it is what is found by looking around that is the most useful", as mentioned by one participant. Another even suggests, "*there's more value in highlighting potential issues with a more concise explanation and let the reviewer write the comment*", and similar comments were provided by 10 other participants. The "review tip" reason was used by 25/57 participants.

The high ratio of appreciation (23% and 28.3%) compared to comment acceptance highlights that the comment generation approach actually addresses multiple tasks: suggesting actionable comments, developer tips and review tips. Future work should explore the possibility to split the overall review comment generation task by predicting the type of generated comments or specializing towards the generation of specific comment types. Developer tips should be shown ahead of the review process (perhaps giving developers access to tools like RevMate before patch submission), and review tips should be shown to reviewers differently than actionable comments, as they act as a warning for reviewers to focus on some specific code snippet. The same observation can be done regarding categories of comments: future work could explore the categories of review comments, considering generation of each category as a separate task, and focus the generation on specific categories, depending on reviewers' preferences.

6.8 Threats to Validity

Construct to Validity:

RevMate currently is limited to two RAG approaches, one retrieving extra code context, the other relevant examples of past chunks with their review comment. Although other approaches could be explored, we chose those two, as well as other

architecture choices (e.g., LLM-as-a-Judge, chain-of-thought), to cover a wider range of modern LLM techniques currently applied in the SE domain [108, 109].

Internal Validity:

Reviewers might accept or reject comments based on their personal judgment and/or style of reviews. When evaluating productivity, different factors, like deadlines, policies, preferences, and tool changes, might affect our results, which we cannot control during our investigation.

When classifying the categories of generated comments in RQ2, we perform an automated labeling using LLM. To mitigate the potential hallucination of the model, we compared the automated labeling with a ground truth build through manual labeling. The LLM was provided with a few-shot list of categories [4]. When compared with the ground truth, the automated labeling reached a moderate agreement with the ground truth, i.e., we observed no substantial hallucination.

External Validity:

Exploring review comment generation for other organizations might impact results based on their review policies and practices. However, our investigation covers two organizations, one being open-source, and one closed-source. Despite their differences, we found that the two organizations achieved similar results in the user study.

In the current version, RevMate generates comments based on OpenAI's GPT4o. LLMs are known to be computationally expensive, regarding both training and generations. The approach we consider uses an off-the-shelf model with few-shot and chain-of-thoughts prompt architecture, thus avoiding the additional costs of fine-tuning the model and maintaining it. Future work should consider exploring fine-tuning the model, regarding the trade-off between the improvement on comment generation and

the related extra costs, either monetary when using an online solution, or computational and storage costs when hosting the model, which was outside of the scope of this study.

6.9 Conclusion

Code review is a major manual quality gate for CI, representing a way to improve the quality of evolving systems while improving developers' social and technical skills [6]. Recent studies have introduced review comment generation tasks into the review process, enhanced by LLM-based approaches [32,36]. However, they have not evaluated the impact of such approaches on the review process.

Through a large-scale mixed open-/closed-source user study on generated review comments, we report acceptance ratios of 8.1% for Mozilla and 7.2% for Ubisoft, and we further observe that appreciation is higher (23% and 28.3%, respectively). Refactoring comments, despite being the second most popular type of generated comments (15.8% and 14.5%) after functional comments (79.2% and 84.8%), have significantly better acceptance levels (18.2% vs 4.8% and 18.6% vs 5.2%). The impact of the LLM-based approach on the review workflow is reasonable as reviewers spend a median time of 43s per review on investigating generated comments and, when applicable, editing accepted ones (37/119) to, in most cases, shorten them (25/37). Regarding accepted comments, we find that their impact on patches' review processes is similar to human-written comments, as 74% vs 73% of comments have at least one follow-up revision at chunk-level, which is promising for future adoption of LLM-generated review comments.

This chapter was submitted as a journal paper titled "Impact of LLM-based Review Comment Generation in Practice: A Mixed Open-/Closed-source User Study" and is under review at IEEE Transactions on Software Engineering (TSE). This paper was written in collaboration with my co-authors: Leuson Da Silva (Polytechnique Montréal), Suhaib Mujahid (Mozilla), Arezou Amini (Ubisoft), Benjamin Mah (Mozilla), Marco Castelluccio (Mozilla), Sarra Habchi (Ubisoft), Foutse Khomh (Polytechnique Montréal) and Bram Adams (Queen's University).

6.10 Future work

We hypothesize that the gap between acceptance and appreciation of generated review comments stems from the task not being properly defined for the LLM. Indeed, the LLM involved addresses multiple tasks, i.e., generating publishable comments, tips for developers and tips for reviewers, all of those being valuable to the review process. We conclude that the current way of generating and presenting the results of review comment generation to reviewers and patch authors in the review environment should be reconsidered. We also argue that, similarly, different categories of comments could be considered as a separate task, and should be split as well or at least evaluated separately. Future work should thus explore how to properly deliver the predictions to reviewers, as suggested by Chaaben et al. [185].

Furthermore, customizing the tool to reviewers' needs, experience and preferences would improve the users' experience with review comment generation tools. One participant even suggested that the customization could depend on the patch author's experience, mentioning "*I would not add a trivial comment when reviewing the code of a tech lead, whereas I would add the trivial comment when reviewing the code of a*

junior developer".

Chapter 7

Conclusion

Using Continuous Integration (CI) leads to many challenges, as it requires to maintain both **an automated gate**, which deals with running compilation and tests continuously with new submitted patches [6], and **a manual gate**, which requires human intervention to review the patches [186]. To assist the CI quality gates, the Software Engineering community has introduced Software Analytics models that apply Machine Learning-based prediction approaches for SE tasks [21].

In the case of the automated gate, Software Analytics models in the form of classifications tasks with traditional ML models (e.g., Random Forests, Naive Bayes) address a large range of tasks, including build outcome prediction [1, 10] and defect prediction [15, 16]. Although such models are highly performant right after training, their performance decays due to concept drift [24]. More than just evaluating model performance at training time, we thus need to evaluate the maintenance effort required to keep performance relevant and the models sustainable.

The manual gate, which consist of the peer review process, has also been targeted by Software Analytics models in the past 10 years, for instance, to provide reviewers

recommendation [28–30] and patch quality prediction [13, 32]. With the recent emergence of LLM-based approaches, pre-trained models can be used for (1) embedding tasks, where the embedding can be used as input for a classifier to potentially improve on prior simpler feature embeddings used in SE, and for (2) generative tasks, including review comment generation [12, 32, 36–38] and code refinement [12, 32, 36].

Although those advanced models allow us to explore new tasks, their evaluations become more challenging: as we enter the realms of assisting or mimicking humans, ML models need to integrated in human-workflows and develop synergy with its users to be sustainable. The use of NLP metrics such as BLEU score and Exact Match, although informative on the capacity of the model to return similar results to a ground truth, does not reflect the actual impact such approach can have on review workflows [36]. While such textual metrics capture the ability of the model to generate outputs similar to a given ground truth, they do not value rephrasing or alternative outputs, nor do they capture if users actually adopt the tool.

In this thesis, we claim that **sustainable usage of ML techniques for continuous integration quality gates requires (1) intelligent decisions with respect to model retraining and (2) synergy with human reviewers' processes.**

7.1 Sustainable training of software analytics models for CI

To validate the first hypothesis, we explored two use-cases integrated in the automated CI gates at our industrial partner, Ubisoft, i.e., Brown Build detection and Just-In-Time risk prediction.

7.1.1 Main findings

In the first study, we carried out a large-scale empirical evaluation of a Brown build detection software analytics model on four projects (three closed-source and one open-source covering 17-43 months of data to evaluate the Time-to-Live (TTL) of models and datasets. We propose heuristics to find the optimal TTL of models based on evaluating the performance decay and model aging and to switch to the newest trained model using a retraining-from-scratch (RFS) setup. We find that the brown build detection model was able to maintain performance while retraining 2-6x less often compared to a weekly retraining. Moreover, the evaluation of our proposed methodology to determine the optimal training set for RFS on the brown build use case, showed that training with 30 weeks of data was sufficient to optimize the performance, as adding more data did not improve results and even harmed them for one of the projects.

Then, our second study compared this improved RFS setup with a Lifelong Learning (LL) setup on the same Brown Build detection use case, as well as on a second use case, i.e., Just-In-Time risk prediction, for which we studied four closed-source projects covering 4-9 years of data. With LL, the model does not need to be discarded, avoiding discontinuity between models, and it can be updated with the newest data points, leading to short training sessions. To avoid catastrophic forgetting, which happens when prior knowledge is lost because of model updates, we consider a replay buffer which, in addition to the newest data points, shows old samples to remind the model of past knowledge. Compared to the RFS setup used in the first study, we found that the LL setup leads to a considerable speed-up of the training process, as the updates use 3.3-13.7x less data, while maintaining performance, and even

improving them significantly in 3/8 of the datasets studied.

7.1.2 Future work

As more and more ML-based approaches are integrated into CI, the trade-off between computational effort and performance can be limiting to the sustainability of software analytics models. By considering such scheduling heuristics (Study 1) or incremental learning setup (Study 2), MLOps Engineers can optimize that trade-off. Until this point, I mentioned use cases from the automated gate, but the application of LL setups on other models in the context of Software Analytics could lead to similar findings.

Moreover, based on our findings, future work could explore other data selection techniques [187]. Indeed, in Study 1 we used sliding windows to choose the training dataset, which is also reflected in Study 2 through the use of a FIFO queue for the replay buffer. Such training windows are already capturing the data getting outdated, though we considered a fixed window over time. However, other data selection approaches could improve performance and computational effort, by selecting data points that are representative of a variety of data and of valuable knowledge at training time for future predictions, regarding the data distribution at the training time [188].

7.2 Sustainable synergy with human processes

The second part of this thesis statement concerns the sustainable integration and adoption of ML-approaches in human processes, hence focusing on the manual gates.

7.2.1 Main findings

Our third study explores the use of hot-spot prediction-based file ordering to assist reviewers in their task. Through a large-scale user study, we found that file ordering impacts review processes, as more problematic parts were commented on with the hot-spot-based ordering compared to the alpha-numeric ordering (+13% precision and +8% recall), hence the need for an efficient prediction approach. Regarding the prediction task training, we studied different types of embeddings, including LLM-based embeddings, and we found that for 5/9 cases, simpler review process feature-based embeddings perform similarly to advanced LLM-based embeddings while reducing the dimensionality, thus leading to reduce computational effort while maintaining performance. Similar findings came from prior work, stating that, in the case of SE classification tasks, simpler approaches are sometimes better than deep learning approaches in terms of performance [69, 70] and speed [71]. Hence, we recommend MLOps Engineers to compare simpler approaches to LLM-based approaches for such tasks before deploying new advanced ML-based solutions in CI.

The forth and final study of this thesis evaluates the impact of LLM-based review comment generation on review processes. The 59 participants of our mixed open-/closed-source user study, all expert reviewers from Ubisoft and Mozilla, were invited to evaluate generated comment suggestions that were provided to them directly in their review environment, during their day-to-day tasks. We found that acceptance of the approach was 8.1% for Mozilla and 7.2% for Ubisoft, while a large set of other comments are still marked as valuable (i.e. as tip for reviewer or for developer), leading to an overall appreciation of 23.6 and 29% respectively.

7.2.2 Future work

The two user studies revealed that LLM-based approaches have the potential to integrate sustainably in the review process since participants were open to using such tools in the future, and their interactions with the approaches led to a positive impact on the review process.

In Study 3, we suggest a file-ordering approach that can, at low-computational cost, improve the order in which files are displayed, as we observe that software related features as embedding performed similar, if not better in some cases, than more advanced approaches. With the rise of LLMs, it is thus still important to consider simpler approaches, as they can still lead to impactful results and successful adoption by users, while limiting the computation effort required to maintain the approach.

Finally, in Study 4, we observed that the comment generation task actually comprises multiple tasks, i.e., the generation of publishable comments, developer tips and review tips, but also the different categories of generation (e.g., refactoring, functional). Future work should consider splitting the tasks and exploring how to introduce the individual tasks in the review environment UI, as they might not be presented the same way to users (i.e., developers and reviewers).

Furthermore, future work should explore improving the quality of review comment generation, as many generated comments were evaluated as too long, trivial or even wrong. This improvement could be done by finetuning the model, which would give the model more knowledge about the codebase and the review habits of the specific targeted project. Such tasks could also benefit from Lifelong Learning-finetuning, extending on the first part of this thesis.

7.3 Final thoughts

This thesis explored the sustainable use of ML-based approaches in the context of software development. We found that the traditional RFS setup for software analytics models can be improved by considering better update scheduling approaches, as well as Lifelong Learning or incremental training approaches. Furthermore, we found not only that the ML models used in the context of software analytics need to be trained in a smarter way, their synergy with human-users also needs to be evaluated in terms of the necessity, adoption and usefulness. User studies are essential steps in such evaluations, as they allow researchers to measure the real-life impact of LLM-based approaches and gather valuable feedback for sustainable usage of such approaches in CI.

Bibliography

[1] D. Olewicki, M. Nayrolles, and B. Adams, "Towards language-independent brown build detection," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2177–2188.

[2] D. Olewicki, S. Habchi, M. Nayrolles, M. Faramarzi, S. Chandar, and B. Adams, "On the costs and benefits of adopting lifelong learning for software analytics-empirical study on brown build and risk prediction," in *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*, 2024, pp. 275–286.

[3] D. Olewicki, S. Habchi, and B. Adams, "An empirical study on code review activity prediction and its impact in practice," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 2238–2260, 2024.

[4] A. K. Turzo and A. Bosu, "What makes a code review useful to opendev developers? an empirical investigation," *Empirical Software Engineering*, vol. 29, no. 1, p. 6, 2024.

[5] M. Fowler and M. Foemmel, "Continuous integration," 2006.

[6] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "An empirical study of the impact of modern code review practices on software quality," *Empirical Software Engineering*, vol. 21, pp. 2146–2189, 2016.

[7] M. Leppänen, S. Mäkinen, M. Pagels, V.-P. Eloranta, J. Itkonen, M. V. Mäntylä, and T. Männistö, "The highways and country roads to continuous deployment," *Ieee software*, vol. 32, no. 2, pp. 64–72, 2015.

[8] O. Kononenko, O. Baysal, and M. W. Godfrey, "Code review quality: How developers see it," in *Proceedings of the 38th international conference on software engineering*, 2016, pp. 1028–1038.

[9] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, "Usage, costs, and benefits of continuous integration in open-source projects," in *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*, 2016, pp. 426–437.

[10] G. Sun, S. Habchi, and S. McIntosh, "Ravenbuild: Context, relevance, and dependency aware build outcome prediction," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 996–1018, 2024.

[11] D. Olewicki, "On the cost and value of build automation," Master's thesis, Polytechnique Montréal, 2020.

[12] R. Tufano, L. Pascarella, M. Tufano, D. Poshyvanyk, and G. Bavota, "Towards automating code review activities," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 163–174.

[13] Y. Hong, C. K. Tantithamthavorn, and P. P. Thongtanunam, "Where should i look at? recommending lines that reviewers should pay attention to," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022, pp. 1034–1045.

[14] A. K. Arani, T. H. M. Le, M. Zahedi, and M. A. Babar, "Systematic literature review on application of learning-based approaches in continuous integration," *IEEE Access*, 2024.

[15] M. Borg, O. Svensson, K. Berg, and D. Hansson, "Szz unleashed: an open implementation of the szz algorithm-featuring example usage in a study of just-in-time bug prediction for the jenkins project," in *Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation*, 2019, pp. 7–12.

[16] M. Nayrolles and A. Hamou-Lhadj, "Clever: combining code metrics with clone detection for just-in-time fault prevention and resolution in large industrial projects," in *Proceedings of the 15th International Conference on Mining Software Repositories*, 2018, pp. 153–164.

[17] D. A. Da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, and A. E. Hassan, "A framework for evaluating the results of the szz approach for identifying bug-introducing changes," *IEEE Transactions on Software Engineering*, vol. 43, no. 7, pp. 641–657, 2016.

[18] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *Proceedings of the 28th international conference on Software engineering*, 2006, pp. 361–370.

[19] F. Hassan and X. Wang, "Change-aware build prediction model for stall avoidance in continuous integration," in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2017, pp. 157–162.

[20] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software testing, verification and reliability*, vol. 22, no. 2, pp. 67–120, 2012.

[21] H. K. Dam, T. Tran, and A. Ghose, "Explainable software analytics," in *Proceedings of the 40th international conference on software engineering: New ideas and Emerging results*, 2018, pp. 53–56.

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

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

[24] S. Wang, S. Schlobach, and M. Klein, "Concept drift and how to identify it," *Journal of Web Semantics*, vol. 9, no. 3, pp. 247–265, 2011.

[25] B. Liu, "Lifelong machine learning: a paradigm for continuous learning," *Frontiers of Computer Science*, vol. 11, no. 3, pp. 359–361, 2017.

[26] M. Tan, L. Tan, S. Dara, and C. Mayeux, "Online defect prediction for imbalanced data," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2. IEEE, 2015, pp. 99–108.

[27] G. I. Parisi, R. Kemker, J. L. Part, C. Kanan, and S. Wermter, "Continual lifelong learning with neural networks: A review," *Neural Networks*, vol. 113, pp. 54–71, 2019.

[28] P. Thongtanunam, C. Tantithamthavorn, R. G. Kula, N. Yoshida, H. Iida, and K.-i. Matsumoto, "Who should review my code? a file location-based code-reviewer recommendation approach for modern code review," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2015, pp. 141–150.

[29] M. M. Rahman, C. K. Roy, and J. A. Collins, "Correct: code reviewer recommendation in github based on cross-project and technology experience," in *Proceedings of the 38th international conference on software engineering companion*, 2016, pp. 222–231.

[30] P. Thongtanunam, R. G. Kula, A. E. C. Cruz, N. Yoshida, and H. Iida, "Improving code review effectiveness through reviewer recommendations," in *Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering*, 2014, pp. 119–122.

[31] T. Baum, K. Schneider, and A. Bacchelli, "On the optimal order of reading source code changes for review," in *2017 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 2017, pp. 329–340.

[32] Z. Li, S. Lu, D. Guo, N. Duan, S. Jannu, G. Jenks, D. Majumder, J. Green, A. Svyatkovskiy, S. Fu *et al.*, "Automating code review activities by large-scale pre-training," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 1035–1047.

[33] A. Bosu, M. Greiler, and C. Bird, "Characteristics of useful code reviews: An empirical study at microsoft," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 2015, pp. 146–156.

[34] M. M. Rahman, C. K. Roy, and R. G. Kula, "Predicting usefulness of code review comments using textual features and developer experience," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 215–226.

[35] V. Efstathiou and D. Spinellis, "Code review comments: language matters," in *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*, 2018, pp. 69–72.

[36] R. Tufano, S. Masiero, A. Mastropaolo, L. Pascarella, D. Poshyvanyk, and G. Bavota, "Using pre-trained models to boost code review automation," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2291–2302.

[37] J. K. Siow, C. Gao, L. Fan, S. Chen, and Y. Liu, "Core: Automating review recommendation for code changes," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 284–295.

[38] A. Gupta and N. Sundaresan, "Intelligent code reviews using deep learning," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'18) Deep Learning Day*, 2018.

[39] R. Tufano, O. Dabić, A. Mastropaolo, M. Ciniselli, and G. Bavota, "Code review automation: Strengths and weaknesses of the state of the art," *IEEE Transactions on Software Engineering*, 2024.

[40] Y. Hong, C. Tantithamthavorn, P. Thongtanunam, and A. Aleti, "Commentfinder: a simpler, faster, more accurate code review comments recommendation," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 507–519.

[41] S. Koch, "Software evolution in open source projects—a large-scale investigation," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 19, no. 6, pp. 361–382, 2007.

[42] A. Z. Henley, K. Muçlu, M. Christakis, S. D. Fleming, and C. Bird, "Cfar: A tool to increase communication, productivity, and review quality in collaborative code reviews," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, 2018, pp. 1–13.

[43] T. Baum and K. Schneider, "On the need for a new generation of code review tools," in *Product-Focused Software Process Improvement: 17th International Conference, PROFES 2016, Trondheim, Norway, November 22-24, 2016, Proceedings 17*. Springer, 2016, pp. 301–308.

[44] M. Shahin, M. A. Babar, and L. Zhu, "Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices," *IEEE access*, vol. 5, pp. 3909–3943, 2017.

[45] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, 2012.

[46] E. Shihab, A. E. Hassan, B. Adams, and Z. M. Jiang, "An industrial study on the risk of software changes," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, 2012, pp. 1–11.

[47] L. Qiao and Y. Wang, "Effort-aware and just-in-time defect prediction with neural network," *PloS one*, vol. 14, no. 2, p. e0211359, 2019.

[48] T. Chen, T. He, M. Benesty, V. Khotilovich, Y. Tang, H. Cho *et al.*, "Xgboost: extreme gradient boosting," *R package version 0.4-2*, vol. 1, no. 4, pp. 1–4, 2015.

[49] B. Adams, H. Tromp, K. De Schutter, and W. De Meuter, "Design recovery and maintenance of build systems," in *2007 IEEE International Conference on Software Maintenance*. IEEE, 2007, pp. 114–123.

[50] M. Zolfagharinia, B. Adams, and Y.-G. Guéhénuc, "Do not trust build results at face value-an empirical study of 30 million cpan builds," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 312–322.

[51] G. Rothermel and M. J. Harrold, "A safe, efficient regression test selection technique," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 6, no. 2, pp. 173–210, 1997.

[52] M. Sherriff, M. Lake, and L. Williams, "Prioritization of regression tests using singular value decomposition with empirical change records," in *The 18th IEEE international symposium on software reliability (ISSRE'07)*. IEEE, 2007, pp. 81–90.

[53] T. Mende and R. Koschke, "Effort-aware defect prediction models," in *2010 14th European Conference on Software Maintenance and Reengineering*. IEEE, 2010, pp. 107–116.

[54] J. Lampel, S. Just, S. Apel, and A. Zeller, "When life gives you oranges: detecting and diagnosing intermittent job failures at mozilla," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1381–1392.

[55] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 643–653.

[56] T. A. Ghaleb, D. A. da Costa, Y. Zou, and A. E. Hassan, "Studying the impact of noises in build breakage data," *IEEE Transactions on Software Engineering*, 2019.

[57] K. Gallaba, C. Macho, M. Pinzger, and S. McIntosh, "Noise and heterogeneity in historical build data: an empirical study of travis ci," in *Proceedings of the*

33rd ACM/IEEE International Conference on Automated Software Engineering, 2018, pp. 87–97.

[58] Z. Dong, A. Tiwari, X. L. Yu, and A. Roychoudhury, "Concurrency-related flaky test detection in android apps," *arXiv preprint arXiv:2005.10762*, 2020.

[59] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "Deflaker: Automatically detecting flaky tests," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 433–444.

[60] K. Herzig and N. Nagappan, "Empirically detecting false test alarms using association rules," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2. IEEE, 2015, pp. 39–48.

[61] A. Alshammari, C. Morris, M. Hilton, and J. Bell, "Flakeflagger: Predicting flakiness without rerunning tests," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1572–1584.

[62] G. Haben, S. Habchi, M. Papadakis, M. Cordy, and Y. Le Traon, "A replication study on the usability of code vocabulary in predicting flaky tests," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, 2021, pp. 219–229.

[63] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le *et al.*, "Program synthesis with large language models," *arXiv preprint arXiv:2108.07732*, 2021.

[64] D. Zan, B. Chen, F. Zhang, D. Lu, B. Wu, B. Guan, W. Yongji, and J.-G. Lou, "Large language models meet nl2code: A survey," in *Proceedings of the 61st*

Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), 2023, pp. 7443–7464.

[65] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, "Fuzzing deep-learning libraries via large language models," *arXiv preprint arXiv:2212.14834*, 2022.

[66] E. Giger, M. D'Ambros, M. Pinzger, and H. C. Gall, "Method-level bug prediction," in *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*, 2012, pp. 171–180.

[67] M. Barnett, C. Bird, J. Brunet, and S. K. Lahiri, "Helping developers help themselves: Automatic decomposition of code review changesets," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 134–144.

[68] O. Ben Sghaier and H. Sahraoui, "Improving the learning of code review successive tasks with cross-task knowledge distillation," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 1086–1106, 2024.

[69] W. Fu and T. Menzies, "Easy over hard: A case study on deep learning," in *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, 2017, pp. 49–60.

[70] V. J. Hellendoorn and P. Devanbu, "Are deep neural networks the best choice for modeling source code?" in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 763–773.

[71] S. Majumder, N. Balaji, K. Brey, W. Fu, and T. Menzies, "500+ times faster than deep learning: A case study exploring faster methods for text mining

stackoverflow," in *Proceedings of the 15th International Conference on Mining Software Repositories*, 2018, pp. 554–563.

[72] D. Angluin, "Queries and concept learning," *Machine learning*, vol. 2, no. 4, pp. 319–342, 1988.

[73] G. Webb, R. Hypde, H. Cao, H. L. Nguyen, and F. Petitjean, "Characterizing concept drift," in *Data Mining and Knowledge Discovery*, 2016, pp. 964–994.

[74] B. Turhan, "On the dataset shift problem in software engineering prediction models," *Empirical Software Engineering*, vol. 17, no. 1, pp. 62–74, 2012.

[75] S. McIntosh and Y. Kamei, "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*, 2018, pp. 560–560.

[76] G. Ditzler and R. Polikar, "Incremental learning of concept drift from streaming imbalanced data," *IEEE transactions on knowledge and data engineering*, vol. 25, no. 10, pp. 2283–2301, 2012.

[77] J. Ekanayake, J. Tappolet, H. C. Gall, and A. Bernstein, "Tracking concept drift of software projects using defect prediction quality," in *2009 6th IEEE International Working Conference on Mining Software Repositories*. IEEE, 2009, pp. 51–60.

[78] M. Ma, S. Zhang, D. Pei, X. Huang, and H. Dai, "Robust and rapid adaption for concept drift in software system anomaly detection," in *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2018, pp. 13–24.

[79] P. Refaeilzadeh, L. Tang, and H. Liu, "Cross-validation." *Encyclopedia of database systems*, vol. 5, pp. 532–538, 2009.

[80] N. D. Brenowitz, B. Henn, J. McGibbon, S. K. Clark, A. Kwa, W. A. Perkins, O. Watt-Meyer, and C. S. Bretherton, "Machine learning climate model dynamics: Offline versus online performance," *arXiv preprint arXiv:2011.03081*, 2020.

[81] S. Tabassum, L. L. Minku, D. Feng, G. G. Cabral, and L. Song, "An investigation of cross-project learning in online just-in-time software defect prediction," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020, pp. 554–565.

[82] M. Weyssow, X. Zhou, K. Kim, D. Lo, and H. Sahraoui, "On the usage of continual learning for out-of-distribution generalization in pre-trained language models of code," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1470–1482.

[83] M. Weyssow, C. Di Sipio, D. Di Ruscio, and H. Sahraoui, "Codell: A lifelong learning dataset to support the co-evolution of data and language models of code," in *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*. IEEE, 2024, pp. 637–641.

[84] G. G. Cabral and L. L. Minku, "Towards reliable online just-in-time software defect prediction," *IEEE Transactions on Software Engineering*, 2022.

[85] I. J. Goodfellow, M. Mirza, D. Xiao, A. Courville, and Y. Bengio, "An empirical investigation of catastrophic forgetting in gradient-based neural networks," *arXiv preprint arXiv:1312.6211*, 2013.

[86] M. Barry, A. Bifet, and J.-L. Billy, "Streamai: Dealing with challenges of continual learning systems for serving ai in production," in *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2023, pp. 134–137.

[87] R. Kemker, M. McClure, A. Abitino, T. Hayes, and C. Kanan, "Measuring catastrophic forgetting in neural networks," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, no. 1, 2018.

[88] G. Ditzler, M. Roveri, C. Alippi, and R. Polikar, "Learning in nonstationary environments: A survey," *IEEE Computational Intelligence Magazine*, vol. 10, no. 4, pp. 12–25, 2015.

[89] D. Isele and A. Cosgun, "Selective experience replay for lifelong learning," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, no. 1, 2018.

[90] M. Riemer, T. Klinger, D. Bouneffouf, and M. Franceschini, "Scalable recollections for continual lifelong learning," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, no. 01, 2019, pp. 1352–1359.

[91] S. Lanka and T. Wu, "Archer: Aggressive rewards to counter bias in hindsight experience replay," *arXiv preprint arXiv:1809.02070*, 2018.

[92] B. Eysenbach, R. Salakhutdinov, and S. Levine, "Search on the replay buffer: Bridging planning and reinforcement learning," *arXiv preprint arXiv:1906.05253*, 2019.

[93] G. I. Parisi, J. Tani, C. Weber, and S. Wermter, "Lifelong learning of human actions with deep neural network self-organization," *Neural Networks*, vol. 96, pp. 137–149, 2017.

[94] M. K. Benna and S. Fusi, "Computational principles of synaptic memory consolidation," *Nature neuroscience*, vol. 19, no. 12, pp. 1697–1706, 2016.

[95] J. Kirkpatrick, R. Pascanu, N. Rabinowitz, J. Veness, G. Desjardins, A. A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska *et al.*, "Overcoming catastrophic forgetting in neural networks," *Proceedings of the national academy of sciences*, vol. 114, no. 13, pp. 3521–3526, 2017.

[96] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[97] T. Chen, Z. Zhang, S. Liu, S. Chang, and Z. Wang, "Long live the lottery: The existence of winning tickets in lifelong learning," in *International Conference on Learning Representations*, 2020.

[98] G. M. Van de Ven and A. S. Tolias, "Three scenarios for continual learning," *arXiv preprint arXiv:1904.07734*, 2019.

[99] Z. Zheng, K. Ning, J. Chen, Y. Wang, W. Chen, L. Guo, and W. Wang, "Towards an understanding of large language models in software engineering tasks," *arXiv preprint arXiv:2308.11396*, 2023.

[100] I. Ozkaya, "Application of large language models to software engineering tasks: Opportunities, risks, and implications," *IEEE Software*, vol. 40, no. 3, pp. 4–8, 2023.

[101] L. Beurer-Kellner, M. Fischer, and M. Vechev, "Prompting is programming: A query language for large language models," *Proceedings of the ACM on Programming Languages*, vol. 7, no. PLDI, pp. 1946–1969, 2023.

[102] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[103] N. Nashid, M. Sintaha, and A. Mesbah, "Retrieval-based prompt selection for code-related few-shot learning," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 2450–2462.

[104] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[105] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models,"

Advances in neural information processing systems, vol. 35, pp. 24 824–24 837, 2022.

[106] J. White, Q. Fu, S. Hays, M. Sandborn, C. Olea, H. Gilbert, A. Elnashar, J. Spencer-Smith, and D. C. Schmidt, "A prompt pattern catalog to enhance prompt engineering with chatgpt," *arXiv preprint arXiv:2302.11382*, 2023.

[107] P. Zhao, H. Zhang, Q. Yu, Z. Wang, Y. Geng, F. Fu, L. Yang, W. Zhang, and B. Cui, "Retrieval-augmented generation for ai-generated content: A survey," *arXiv preprint arXiv:2402.19473*, 2024.

[108] S. Zhou, U. Alon, F. F. Xu, Z. Wang, Z. Jiang, and G. Neubig, "Docprompting: Generating code by retrieving the docs," *arXiv preprint arXiv:2207.05987*, 2022.

[109] M. R. Parvez, W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Retrieval augmented code generation and summarization," *arXiv preprint arXiv:2108.11601*, 2021.

[110] L. Zheng, W.-L. Chiang, Y. Sheng, S. Zhuang, Z. Wu, Y. Zhuang, Z. Lin, Z. Li, D. Li, E. Xing *et al.*, "Judging llm-as-a-judge with mt-bench and chatbot arena," *Advances in Neural Information Processing Systems*, vol. 36, pp. 46 595–46 623, 2023.

[111] M. Weyssow, A. Kamanda, and H. Sahraoui, "Codeultrafeedback: An llm-as-a-judge dataset for aligning large language models to coding preferences," *arXiv preprint arXiv:2403.09032*, 2024.

[112] R. Compton, L. Zhang, A. Puli, and R. Ranganath, "When more is less: Incorporating additional datasets can hurt performance by introducing spurious correlations," in *Machine Learning for Healthcare Conference*. PMLR, 2023, pp. 110–127.

[113] G. König, C. Molnar, B. Bischl, and M. Grosse-Wentrup, "Relative feature importance," in *2020 25th International Conference on Pattern Recognition (ICPR)*. IEEE, 2021, pp. 9318–9325.

[114] Facebook Inc. Captum python library. Accessed December 2022. [Online]. Available: https://captum.ai/

[115] J. Ramos *et al.*, "Using tf-idf to determine word relevance in document queries," in *Proceedings of the first instructional conference on machine learning*, vol. 242, no. 1. Citeseer, 2003, pp. 29–48.

[116] G. Malinas, "Simpson's paradox: A logically benign, empirically treacherous hydra," *The monist*, vol. 84, no. 2, pp. 265–283, 2001.

[117] I. Saidani, A. Ouni, and M. W. Mkaouer, "Improving the prediction of continuous integration build failures using deep learning," *Automated Software Engineering*, vol. 29, no. 1, pp. 1–61, 2022.

[118] D. Marijan, "Comparative study of machine learning test case prioritization for continuous integration testing," *arXiv preprint arXiv:2204.10899*, 2022.

[119] S. Luan, D. Yang, C. Barnaby, K. Sen, and S. Chandra, "Aroma: Code recommendation via structural code search," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–28, 2019.

[120] M. E. H. Schumacher, K. T. Le, and A. Andrzejak, "Improving code recommendations by combining neural and classical machine learning approaches," in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, 2020, pp. 476–482.

[121] D. Sato, A. Wider, and C. Windheuser, "Continuous delivery for machine learning: Automating the end-to-end lifecycle of machine learning applications," 2019. [Online]. Available: https://martinfowler.com/articles/cd4ml.html

[122] N. Gift and A. Deza, *Practical MLOps*. " O'Reilly Media, Inc.", 2021.

[123] J. Lu, A. Liu, F. Dong, F. Gu, J. Gama, and G. Zhang, "Learning under concept drift: A review," *IEEE Transactions on Knowledge and Data Engineering*, vol. 31, no. 12, pp. 2346–2363, 2018.

[124] P. Tsangaratos and I. Ilia, "Comparison of a logistic regression and naïve bayes classifier in landslide susceptibility assessments: The influence of models complexity and training dataset size," *Catena*, vol. 145, pp. 164–179, 2016.

[125] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, 2016, pp. 785–794.

[126] F.-L. Chung, Z. Deng, and S. Wang, "From minimum enclosing ball to fast fuzzy inference system training on large datasets," *IEEE Transactions on Fuzzy Systems*, vol. 17, no. 1, pp. 173–184, 2008.

[127] S. Demeyer, S. Ducasse, and O. Nierstrasz, *Object-oriented reengineering patterns*. Elsevier, 2002.

[128] C. Rosen, B. Grawi, and E. Shihab, "Commit guru: analytics and risk prediction of software commits," in *Proceedings of the 2015 10th joint meeting on foundations of software engineering*, 2015, pp. 966–969.

[129] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *Proceedings of the 30th international conference on Software engineering*, 2008, pp. 181–190.

[130] M. W. Gardner and S. Dorling, "Artificial neural networks (the multilayer perceptron)—a review of applications in the atmospheric sciences," *Atmospheric environment*, vol. 32, no. 14-15, pp. 2627–2636, 1998.

[131] M. Li, T. Zhang, Y. Chen, and A. J. Smola, "Efficient mini-batch training for stochastic optimization," in *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2014, pp. 661–670.

[132] Graphviz. Graphviz website. Accessed August 2022. [Online]. Available: https://graphviz.org/

[133] GitLab B.V. Gitlab website. Accessed August 2022. [Online]. Available: https://about.gitlab.com/

[134] H. S. Wilf, *Algorithms and complexity*. AK Peters/CRC Press, 2002.

[135] S. Thuijsman, D. Hendriks, R. Theunissen, M. Reniers, and R. Schiffelers, "Computational effort of bdd-based supervisor synthesis of extended finite automata," in *2019 IEEE 15th International Conference on Automation Science and Engineering (CASE)*. IEEE, 2019, pp. 486–493.

[136] A. Chaudhry, P. K. Dokania, T. Ajanthan, and P. H. Torr, "Riemannian walk for incremental learning: Understanding forgetting and intransigence," in *Proceedings of the European conference on computer vision (ECCV)*, 2018, pp. 532–547.

[137] F. Zenke, B. Poole, and S. Ganguli, "Continual learning through synaptic intelligence," in *International conference on machine learning*. PMLR, 2017, pp. 3987–3995.

[138] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[139] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[140] K. Han, A. Xiao, E. Wu, J. Guo, C. Xu, and Y. Wang, "Transformer in transformer," *Advances in Neural Information Processing Systems*, vol. 34, pp. 15 908–15 919, 2021.

[141] P. Rigby, B. Cleary, F. Painchaud, M.-A. Storey, and D. German, "Contemporary peer review in action: Lessons from open source development," *IEEE software*, vol. 29, no. 6, pp. 56–61, 2012.

[142] M. Fagan, "A history of software inspections," *Software pioneers: contributions to software engineering*, pp. 562–573, 2002.

[143] C. Sadowski, E. Söderberg, L. Church, M. Sipko, and A. Bacchelli, "Modern code review: a case study at google," in *Proceedings of the 40th international conference on software engineering: Software engineering in practice*, 2018, pp. 181–190.

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

[145] Y. Huang, N. Jia, X. Chen, K. Hong, and Z. Zheng, "Salient-class location: Help developers understand code change in code review," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 770–774.

[146] G. Petrović and M. Ivanković, "State of mutation testing at google," in *Proceedings of the 40th international conference on software engineering: Software engineering in practice*, 2018, pp. 163–171.

[147] E. Fregnan, L. Braz, M. D'Ambros, G. Çalıklı, and A. Bacchelli, "First come first served: the impact of file position on code review," in *Proceedings of the*

30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2022, pp. 483–494.

[148] "Replication package," https://zenodo.org/records/10783562, accessed: 2023-09-28.

[149] N. Dahlbäck, A. Jönsson, and L. Ahrenberg, "Wizard of oz studies: why and how," in *Proceedings of the 1st international conference on Intelligent user interfaces*, 1993, pp. 193–200.

[150] "Countvectorizer python library," https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html, accessed: 2023-09-24.

[151] "Hugging face website," https://huggingface.co/, accessed: 2023-09-24.

[152] BigScience, "Bigscience language open-science open-access multilingual (bloom) language model," *International*, May 2021 - May 2022.

[153] "Sentence transformer," https://huggingface.co/sentence-transformers, accessed: 2023-09-24.

[154] J. Wang, M. Xu, H. Wang, and J. Zhang, "Classification of imbalanced data by using the smote algorithm and locally linear embedding," in *2006 8th international Conference on Signal Processing*, vol. 3. IEEE, 2006.

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

[156] Z. Chase Lipton, C. Elkan, and B. Narayanaswamy, "Thresholding classifiers to maximize f1 score," *arXiv e-prints*, pp. arXiv–1402, 2014.

[157] J. Cuzick, "A wilcoxon-type test for trend," *Statistics in medicine*, vol. 4, no. 1, pp. 87–90, 1985.

[158] G. Macbeth, E. Razumiejczyk, and R. D. Ledesma, "Cliff's delta calculator: A non-parametric effect size program for two groups of observations," *Universitas Psychologica*, vol. 10, no. 2, pp. 545–555, 2011.

[159] S. Wattanakriengkrai, P. Thongtanunam, C. Tantithamthavorn, H. Hata, and K. Matsumoto, "Predicting defective lines using a model-agnostic technique," *IEEE Transactions on Software Engineering*, vol. 48, no. 5, pp. 1480–1496, 2020.

[160] A. Sharma, A. Gupta, and M. Bilalpur, "Argumentative stance prediction: An exploratory study on multimodality and few-shot learning," *arXiv preprint arXiv:2310.07093*, 2023.

[161] R. Morales, S. McIntosh, and F. Khomh, "Do code review practices impact design quality? a case study of the qt, vtk, and itk projects," in *2015 IEEE 22nd international conference on software analysis, evolution, and reengineering (SANER)*. IEEE, 2015, pp. 171–180.

[162] M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens, "Modern code reviews in open-source projects: Which problems do they fix?" in *Proceedings of the 11th working conference on mining software repositories*, 2014, pp. 202–211.

[163] A. Bosu, J. C. Carver, C. Bird, J. Orbeck, and C. Chockley, "Process aspects and social dynamics of contemporary code review: Insights from open source development and industrial practice at microsoft," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 56–75, 2017.

[164] J. Wang, P. C. Shih, Y. Wu, and J. M. Carroll, "Comparative case studies of open source software peer review practices," *Information and Software Technology*, vol. 67, pp. 1–12, 2015.

[165] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 712–721.

[166] T. Pangsakulyanont, P. Thongtanunam, D. Port, and H. Iida, "Assessing mcr discussion usefulness using semantic similarity," in *2014 6th International Workshop on Empirical Software Engineering in Practice*. IEEE, 2014, pp. 49–54.

[167] L. Pascarella, D. Spadini, F. Palomba, M. Bruntink, and A. Bacchelli, "Information needs in contemporary code review," *Proceedings of the ACM on Human-Computer Interaction*, vol. 2, no. CSCW, pp. 1–27, 2018.

[168] F. Ebert, F. Castor, N. Novielli, and A. Serebrenik, "An exploratory study on confusion in code reviews," *Empirical Software Engineering*, vol. 26, pp. 1–48, 2021.

[169] P. Wurzel Gonçalves, G. Calikli, A. Serebrenik, and A. Bacchelli, "Competencies for code review," *Proceedings of the ACM on Human-Computer Interaction*, vol. 7, no. CSCW1, pp. 1–33, 2023.

[170] M. Chouchen, A. Ouni, R. G. Kula, D. Wang, P. Thongtanunam, M. W. Mkaouer, and K. Matsumoto, "Anti-patterns in modern code review: Symptoms and prevalence," in *2021 IEEE international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 2021, pp. 531–535.

[171] R. Paul, A. K. Turzo, and A. Bosu, "Why security defects go unnoticed during code reviews? a case-control study of the chromium os project," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1373–1385.

[172] T. Baum, K. Schneider, and A. Bacchelli, "Associating working memory capacity and code change ordering with code review performance," *Empirical Software Engineering*, vol. 24, pp. 1762–1798, 2019.

[173] S. Mujahid and M. Castelluccio, "Dataset of generated code review comments for Firefox developers," Oct. 2024. [Online]. Available: https://doi.org/10.5281/zenodo.14014324

[174] Mozilla, "Mozilla's bugbug code review project," 2024, accessed on 2024-10-14. See bugbug/tools/code_review.py. [Online]. Available: https://github.com/mozilla/bugbug

[175] H. S. Zheng, S. Mishra, X. Chen, H.-T. Cheng, E. H. Chi, Q. V. Le, and D. Zhou, "Take a step back: Evoking reasoning via abstraction in large language models," *arXiv preprint arXiv:2310.06117*, 2023.

[176] Y. Li, J. Shi, and Z. Zhang, "An approach for rapid source code development based on chatgpt and prompt engineering," *IEEE Access*, 2024.

[177] L. Ardito, L. Barbato, M. Castelluccio, R. Coppola, C. Denizet, S. Ledru, and M. Valsesia, "rust-code-analysis: A rust library to analyze and extract maintainability information from source codes," *SoftwareX*, vol. 12, p. 100635, 2020.

[178] Qdrant, "Qdrant vector database," 2024, accessed on 2024-10-08. [Online]. Available: https://qdrant.tech/

[179] F. Lin, D. J. Kim *et al.*, "When llm-based code generation meets the software development process," *arXiv preprint arXiv:2403.15852*, 2024.

[180] K. Muller, "Statistical power analysis for the behavioral sciences," 1989.

[181] L. Morchard, "Clustering ideas with llamafile," 2024, accessed on 2024-10-02. [Online]. Available: https://blog.lmorchard.com/2024/05/10/topic-clustering-llamafile/

[182] Sklearn, "Sklearn's kmeans," 2024, accessed on 2024-10-08. [Online]. Available: https://scikit-learn.org/1.5/modules/generated/sklearn.cluster.KMeans.html

[183] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and psychological measurement*, vol. 20, no. 1, pp. 37–46, 1960.

[184] J. L. Campbell, C. Quincy, J. Osserman, and O. K. Pedersen, "Coding in-depth semistructured interviews: Problems of unitization and intercoder reliability and agreement," *Sociological methods & research*, vol. 42, no. 3, pp. 294–320, 2013.

[185] M. B. Chaaben, L. Burgueño, I. David, and H. Sahraoui, "On the utility of domain modeling assistance with large language models," *arXiv preprint arXiv:2410.12577*, 2024.

[186] L. Dong, H. Zhang, L. Yang, Z. Weng, X. Yang, X. Zhou, and Z. Pan, "Survey on pains and best practices of code review," in *2021 28th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2021, pp. 482–491.

[187] R. Roscher, M. Rußwurm, C. Gevaert, M. Kampffmeyer, J. A. d. Santos, M. Vakalopoulou, R. Hänsch, S. Hansen, K. Nogueira, J. Prexl *et al.*, "Better, not just more: Data-centric machine learning for earth observation," *arXiv preprint arXiv:2312.05327*, 2023.

[188] D. J. MacKay, "Information-based objective functions for active data selection," *Neural computation*, vol. 4, no. 4, pp. 590–604, 1992.

Appendix A

Research Ethics Approval

For Study 3 and 4, we required ethical approval to perform the user studies, which we requested in one consolidated application. We provide the approval in the following pages.

Queen's University

Queen's University General Research Ethics Board (GREB)

GREB Initial Ethics Clearance

December 04, 2023

Ms. Olewicki
School of Computing
Queen's University

TRAQ #: 6040278
Study Title: "RevMate: Towards an optimized review process in AAA game development"
Date Ethics Clearance Issued: December 04, 2023
Ethics Clearance Expiry Date: December 04, 2024

Dear Ms. Olewicki:

The General Research Ethics Board (GREB), by means of a delegated board review, has cleared your proposal entitled **"RevMate: Towards an optimized review process in AAA game development"** for ethical compliance with the Tri-Council Guidelines (TCPS 2) and Queen's ethics policies. In accordance with the Tri-Council Guidelines (Article 6.14) and Standard Operating Procedures (405), your project has been cleared for one year.

Renewals: An annual renewal event form or a study closure event form must be submitted annually as per the TCPS 2 Article 6.14. As a courtesy, the Office of Research Ethics Compliance may send reminders 30 days in advance of the ethics clearance expiry date. All lapses in ethics clearance will be documented on the annual renewal clearance letter. A Suspension letter may be issued for lapses in ethics clearances, with subsequent termination and closure of the ethics file for lapses greater than 10 business days.

Completion/Termination: The GREB must be notified of the completion or termination of this study through the submission of a study closure event form in TRAQ. This should be submitted at the time of completion; there is no need to wait until the annual renewal due date.

Amendments: No deviation from, or changes to the protocol, informed consent form and conduct of study should be initiated without prior written clearance or an appropriate amendment event from the GREB, except when necessary to eliminate immediate hazard(s) to study participants or when the change(s) involves only administrative or logistical aspects of the study. For example, you must report changes to the level of risk, applicant characteristics, and implementation of new procedures. To submit an amendment form, access the application by at http://www.queensu.ca/traq/signon.html; click on "Events;" under "Create New Event" click on "General Research Ethics Board Request for the Amendment of Approved Studies." Once submitted, these changes will automatically be sent to the Ethics Coordinator, GREB, at University Research Services for further review and clearance by GREB or the Chair, GREB.

Adverse Events: You are reminded of your obligation to advise the GREB of any adverse event(s) that occur during this one-year period (access this form at http://www.queensu.ca/traq/signon.html/; click on "Events;" under "Create New Event" click on "General Research Ethics Board Adverse Event Form"). An adverse event includes, but is not

limited to, a complaint, a change or unexpected event that alters the level of risk for the researcher or participants or situation that requires a substantial change in approach to a participant(s). You are also advised that all adverse events must be reported to the GREB within 48 hours.

On behalf of the General Research Ethics Board, I wish you continued success in your research.

Sincerely,

Chair, General Research Ethics Board (GREB)
Associate Professor and Distinguished Faculty Fellow of Marketing
Master of Digital Product Management
Smith School of Business
Queen's University
chair.GREB@queensu.ca

Index

Conclusion, 157

Introduction, 1

List of Figures, xviii

List of Tables, xiv

Related work, 14

STUDY 1: Dynamic re-training scheduling strategies - Concept Drift detection for Brown Build Prediction, 28

STUDY 2: Lifelong learning setup for software analytics models , 50

STUDY 3: Hot-spot prediction-based file ordering for code review, 82

STUDY 4: Impact of code review comment generation, 121

Table of Contents, vii