

BEYOND THE CODE: LEVERAGING SOCIO-TECHNICAL
KNOWLEDGE TO IMPROVE THE PERFORMANCE OF AUTOMATED
APPROACHES TO SUPPORT LOGGING ACTIVITIES

by

YOUSSEF ESSEDDIQ OUATITI

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

Queen's University

Kingston, Ontario, Canada

May 2026

Copyright © Youssef Esseddiq Ouatiti, 2026

Abstract

LOGS are generated through logging statements inserted within software code and serve as a valuable resource for identifying execution anomalies, as well as preventing and debugging issues. Due to the complex nature of logging, several machine learning approaches have been proposed to guide developers on where, what, and at which level to log. However, modern software systems, with long maintenance histories, multi-component structures, and multiple maintaining teams lead to diverse and evolving logging strategies across components, teams, and over time, which complicates ML-based automation efforts. This thesis investigates whether automated approaches for supporting logging activities can be improved by incorporating socio-technical knowledge that exists beyond the code. Using data from large, modern software systems, we study three socio-technical signals: components, ownership, and evolution over time and examine their effect on automated tools that assist developers in selecting the appropriate logging levels.

Overall, our work shows that leveraging software engineering knowledge beyond the code can significantly improve the predictive performance of automated tools to support logging activities.

Dedication

To my grandmother (Mino)

Co-authorship

For each of the chapters and related publications of this thesis, my contributions are: the drafting of the research idea; researching the background material and related work; the collection of the data; the proposal of the research methods; the analysis of the data; and the writing of the manuscripts. My co-contributors supported me in refining the initial research ideas; providing suggestions to refine my research methods; and providing feedback on manuscript drafts. The work presented in this thesis is submitted as listed below:

- Youssef Esseddiq Ouatiti, Mohammed Sayagh, Nouredine Kerzazi, and Ahmed E. Hassan. 2023. An Empirical Study on Log Level Prediction for Multi-Component Systems. IEEE Transactions on Software Engineering. PP. 1-1. This work is described under Chapter [4](#).

- Youssef Esseddiq Ouatiti, Mohammed Sayagh, Nouredine Kerzazi, Bram Adams, and Ahmed E. Hassan. 2024. The impact of concept drift and data leakage on log level prediction models. Empirical Software Engineering. 29. 10.1007/s10664-024-10518-9. This work is described under Chapter 5.
- Youssef Esseddiq Ouatiti, Mohammed Sayagh, Bram Adams, and Ahmed E. Hassan. 2026. Retrieval Supersedes Scale: Efficient Log Level Prediction via Multiplex Socio-Technical Context. This work is submitted for review to the IEEE Transactions on Software Engineering journal and is described under Chapter 6.

Statement of Originality and Disclosure of Use of Artificial Intelligence

1. Use of AI in research methodology (Experimental objects): A part of this thesis investigates the utility of Large Language Models (LLMs) in software engineering tasks. As such, AI models (specifically CodeLlama, DeepSeek, and OpenAI GPT) were utilized as experimental subjects to generate data for evaluation. The outputs of these models (e.g., log level suggestions, code embeddings) serve as the dataset for this study and are analyzed critically in Chapters 6.

2. Use of AI in Thesis Preparation (Editorial Assistance): In the preparation of the written manuscript, Generative AI tools (namely Google’s Gemini and OpenAI’s ChatGPT) were used exclusively for linguistic refinement and editorial proofreading, which is consistent with the submission guidelines of major software engineering venues (e.g., IEEE, ACM, Springer). Specifically, AI tools were employed for the following limited purposes:

- Grammar and Syntax: To identify and correct grammatical errors, typos, and punctuation mistakes.
- Readability: To improve the flow and clarity of complex technical sentences.
- Formatting: To assist in generating LaTeX code for tables and bibliography management.

3. Statement of Originality and Human Oversight: I declare that no AI tool was used to generate the intellectual content, scientific hypotheses, architectural designs, or conclusions presented in this thesis. The core ideas, analyses, and final text structure are entirely the original work of the author. All suggestions made by AI tools regarding language and style were manually reviewed, verified, and approved by the author before inclusion.

Acknowledgments

I am deeply grateful to my supervisor, Prof. Ahmed E. Hassan, for the opportunity to pursue this PhD under his guidance. I appreciate his unwavering support throughout this journey. Our discussions were not only insightful but often transformative, constantly challenging my perspective and providing new lenses through which to view problems. I deeply admire the high standards Prof. Ahmed sets to establish the SAIL lab as a world-class environment through his constant push to raise the bar, which undoubtedly made me a better researcher. I am also grateful to Prof. Ahmed for facilitating an internship which allowed me to bridge the gap between academic research and industry standards, making my PhD experience truly unique and well-rounded.

I would also like to thank Prof. Mohammed Sayagh, who transitioned from post-doc to professor just as I joined the lab. Mohammed has been a strict but fair big brother figure to me. I cherish our research discussions (often held in between

tennis games) and his indispensable guidance. I am grateful for his availability, his willingness to answer every question, his meticulous attention to detail while reviewing my drafts, and his openness to bouncing around research ideas.

My sincere thanks go to Prof. Bram Adams, who taught the first and arguably most influential course of my doctoral studies. Thank you for your availability, your insightful ideas, your help in reviewing drafts, and your overall encouragement and support.

I would like to dedicate a special note of gratitude to Prof. Nouredine Kerzazi, my Master's professor. Nouredine was the one who believed in me, encouraged me and helped me embark on this PhD career. His inspiration, encouragement, and support were the spark that started this journey.

My endless gratitude and appreciation goes to my mom and dad, who have always believed in me, sacrificed for me, and supported me in every way possible. To my younger sisters, Salma and Ikhlas thank you for being such joy-bringers and supporting me throughout. Finally, to Khaoula, my best friend: I appreciate your patience, support, and help during both the easy and the hard times.

I dedicate this thesis to the memory of my grandmother (Mino).

Table of Contents

Abstract	i
Dedication	iii
Co-authorship	iv
Statement of Originality and Disclosure of Use of Artificial Intelligence	vi
Acknowledgments	viii
List of Tables	xiii
List of Figures	xv
Chapter 1 Introduction	1
1.1 Observability in Software Engineering	1
1.2 The State of Software Logging	3
1.3 Socio-technical Knowledge in Open Source Software	4
1.4 Research Hypothesis	6
1.5 Contributions	7
1.6 Thesis Structure	10

Chapter 2 Background	12
2.1 Software Logging Fundamentals	13
2.2 Socio-Technical Dimensions of Software Engineering	16
2.3 AI/ML Paradigms in Logging Level Prediction	18
Chapter 3 Literature Survey	22
3.1 Empirical studies on logging practices	23
3.2 Automated Support for developer logging decisions	29
Chapter 4 An Empirical Study on Logging Level Prediction for Multi- component Systems	41
4.1 Introduction	43
4.2 Methodology	49
4.3 Results	55
4.4 Threats to Validity	73
4.5 Chapter Summary	74
Chapter 5 The Impact of Concept Drift and Data Leakage on Logging Level Prediction Models	76
5.1 Introduction	77
5.2 Methodology	85
5.3 Results	93
5.4 Threats to Validity	127
5.5 Chapter Summary	128

Chapter 6 Retrieval Supersedes Scale: Efficient Logging Level Prediction via Multiplex Socio-Technical Context	130
6.1 Introduction	132
6.2 Methodology	138
6.3 Empirical evaluation setup	149
6.4 Results	155
6.5 Threats to validity	168
6.6 Chapter Summary	170
Chapter 7 Conclusion	172
7.1 Closing the Loop: Beyond the Code for Logging Automation	172
7.2 Summary of Key Findings and Contributions	173
7.3 Implications for Practice: What “Socio-technical” Means Operationally	176
7.4 Limitations and Threats to Validity	177
7.5 Future Work	180
7.6 Closing Remarks	182
Bibliography	183
Appendices	200
A Appendix - Logging level prediction for multi-component software systems	200

List of Tables

5.1	Features that are used to train the Shallow (Li et al., 2017a) and DL (Li et al., 2021b) LLPs	87
5.2	Studied Projects historical information	88
6.1	Summary of Large Language Models Selected for Evaluation	148
6.2	Summary of the studied projects	150
6.3	Summary of the performance of the multiplex retrieval compared to other approaches (metrics are obtained from 5 bootstrap runs and for CodeLlama-7B)	157
6.4	Overview of the mispredictions using OFA-LLP (multiplex mode)	158
6.5	Ownership-based clustering quality metrics	160
6.6	Summary of the performance of the ownership-only retrieval compared to other approaches (metrics are obtained from 5 bootstrap runs and for CodeLlama-7B)	161
6.7	Semantic-based clustering quality metrics	162
6.8	Summary of the performance of the semantic-only retrieval compared to other approaches (metrics are obtained from 5 bootstrap runs and for CodeLlama-7B)	163

6.9 Summary of the performance of the semantic-only retrieval compared to other approaches (metrics are obtained from 5 bootstrap runs and for CodeLlama-34BB)	164
6.10 Predictive Performance (AUC): Local Models vs. Commercial Baselines	166
6.11 Operational Cost Comparison: Self-Hosted vs. Commercial APIs (per 1k Logs)	167
6.12 The Scaling Analysis: Performance Gains vs. Operational Cost Increases (4x H100)	168
A.1 Important features rankings for Hadoop’s local and global models	220
A.2 Important features rankings for Spring’s local and global models	221
A.3 Important features rankings for Jupyter’s local and global models	222
A.4 Important features rankings for Elasticsearch’s local and global models	223
A.5 Important features rankings for OpenStack’s local and global models	224

List of Figures

2.1	Example of an added logging statement to the Hadoop project ¹	14
2.2	logging levels verbosity in Java (Left) and Python (Right) projects . .	14
3.1	Overview of the discussed research directions	23
4.1	An illustration of the models that we evaluated in our study according to data from different components. Note that each of these experiments are repeated 100 times using different bootstrap dataset for the training/testing.	50
4.2	An overview of our training and testing methodology	51
4.3	The AUC performance of the global model on components and on the OpenStack project	56
4.4	AUC scores of the global, local and mixed-effect models on the OpenStack components	60
4.5	Brier Score of the global, local and mixed-effect models on the OpenStack project	61
4.6	Corrected AIC of the global, local and mixed-effect models on the OpenStack project	62
4.7	The AUC performance of the global and peer-local models on the Aodh component	66

4.8	The Brier Score performance of the global and peer-local models on the Aodh component	67
5.1	An overview of the methodology for training our models. Both training and testing datasets are further detailed in the approaches of our research questions.	88
5.2	Overview of the modeling approaches for a single studied time frame.	94
5.3	An overview of the methodology for quantifying data leakage	96
5.4	AUC performance of the random and time-aware Shallow-LLPs on four-month testing time frames. R indicates time frames where the random approach is the best performing, T indicates time frames where the time-aware approach model is the best performing and N indicates time frames where there is no significant difference between the performance of the two models.	98
5.5	AUC performance of the random and time-aware DL-LLPs on four-month testing time frames. R indicates time frames where the random approach is the best performing, T indicates time frames where the time-aware approach model is the best performing and N indicates time frames where there is no significant difference between the performance of the two models.	99
5.6	Overview of model evaluation under concept drift for a specific time frame size.	103

5.7	AUC performance of Hadoop’s Shallow-LLPs across time. the lines represent the 1st quantile, median and 3rd quantile for the baseline AUC performance. Red boxplots show time frames with a performance statistically below the baseline, blue boxplots show no statistical difference and green boxplots show time frames with a performance statistically better than the baseline.	104
5.8	AUC performance of Spring’s Shallow-LLPs across time. the lines represent the 1st quantile, median and 3rd quantile for the baseline AUC performance. Red boxplots show time frames with a performance statistically below the baseline, blue boxplots show no statistical difference and green boxplots show time frames with a performance statistically better than the baseline.	105
5.9	AUC performance of OpenStack’s Shallow-LLPs across time. the lines represent the 1st quantile, median and 3rd quantile for the baseline AUC performance. Red boxplots show time frames with a performance statistically below the baseline, blue boxplots show no statistical difference and green boxplots show time frames with a performance statistically better than the baseline.	106
5.10	AUC performance of Hadoop’s DL-LLPs across time. the lines represent the 1st quantile, median and 3rd quantile for the baseline AUC performance. Red boxplots show time frames with a performance statistically below the baseline, blue boxplots show no statistical difference and green boxplots show time frames with a performance statistically better than the baseline.	107

5.11 AUC performance of Spring’s DL-LLPs across time. the lines represent the 1st quantile, median and 3rd quantile for the baseline AUC performance. Red boxplots show time frames with a performance statistically below the baseline, blue boxplots show no statistical difference and green boxplots show time frames with a performance statistically better than the baseline.	108
5.12 AUC performance of OpenStack’s DL-LLPs across time. the lines represent the 1st quantile, median and 3rd quantile for the baseline AUC performance. Red boxplots show time frames with a performance statistically below the baseline, blue boxplots show no statistical difference and green boxplots show time frames with a performance statistically better than the baseline.	109
5.13 Comparison of contextual and all-knowing models. We compare the models shown in the figure just on their first following testing time frame. We train new contextual and all-knowing models for each testing time frame.	114
5.14 The AUC Performance of contextual and all-knowing Shallow-LLPs on a selection of testing time frames (two-month long) - The remaining time frames (e.g., 1 to 35 for Hadoop) are available in the appendix.	116
5.15 Percentage of testing time frames in which one of our time-aware DL-LLPs (i.e., contextual or All-knowing) is the statistically significantly best performing model in terms of AUC.	117
5.16 Percentage of testing time frames in which the contextual Shallow-LLPs perform the best in terms of AUC.	118

5.17 The AUC Performance of the varying all-knowing Shallow-LLP on all testing time frames (two-month long)	120
5.18 Overview for comparison of interpretability across time	123
5.19 Feature survival rate for logging level prediction models. For clarity purpose, we only show the four, six and eight months based models.	126
6.1 Logging statement template	132
6.2 Overview of our Ownership-Functionality Aware retrieval approach for logging level prediction.	139
6.3 Overview of the semantic-based clustering	140
6.4 Overview of the ownership-based clustering	143
6.5 Overview of multiplex graphs. Nodes are Java files and edges are labeled with cosine similarity scores between the vectors representing the connected files.	144
6.6 Prompt creation process for the LLM-LLP.	147
6.7 Predictive Accuracy vs. Operational Cost Analysis	167
A.1 The AUC performance of the global model on components and on the entire project	201
A.2 The Brier Score performance of the global model on different components and on the entire project	202
A.3 AUC scores of the global, local and mixed-effect models	203
A.4 Brier scores of the global, local and mixed-effect models	204
A.5 AICc scores of the global, local and mixed-effect models (log-scale) .	205
A.6 OpenStack - Blazar	205
A.7 OpenStack - Cyborg	206

A.8 OpenStack - ec2-api	206
A.9 OpenStack - Horizon	206
A.10 OpenStack - Masakari	207
A.11 OpenStack - Placement	207
A.12 OpenStack - Qinling	207
A.13 OpenStack - Solum	208
A.14 OpenStack - Zaqar	208
A.15 Spring - Batch	208
A.16 Spring - Cloud-Commons	209
A.17 Spring - Data-Commons	209
A.18 Spring - Ldap	209
A.19 Spring - Roo	210
A.20 Spring - Session	210
A.21 Spring - Statemachine	210
A.22 Spring - Vault	211
A.23 Spring - Ws	211
A.24 OpenStack - Blazar	212
A.25 OpenStack - Cyborg	212
A.26 OpenStack - ec2-api	213
A.27 OpenStack - Horizon	213
A.28 OpenStack - Masakari	213
A.29 OpenStack - Placement	214
A.30 OpenStack - Qinling	214
A.31 OpenStack - Solum	214

A.32 OpenStack - Zaqar	215
A.33 Spring - Batch	215
A.34 Spring - Cloud-Commons	215
A.35 Spring - Data-Commons	216
A.36 Spring - Ldap	216
A.37 Spring - Roo	216
A.38 Spring - Session	217
A.39 Spring - Statemachine	217
A.40 Spring - Vault	217
A.41 Spring - Ws	218
A.42 Level of Correlation between the rankings of best performing peer- local models across the data lacking components of OpenStack.	218
A.43 Level of Correlation between the rankings of best performing peer- local models across the data lacking components of Spring.	219

CHAPTER 1

Introduction

1.1 Observability in Software Engineering

1.1.1 The Epistemology of Observability and the Telemetry Triad

THE operational reliability of software systems relies directly on how observable they are. Borrowed from control theory, observability is formally defined as the measure of how well internal system states can be inferred from knowledge of their external outputs ([Kalman, 1960](#)). In the domain of software engineering, this theoretical property is realized through a telemetry framework often referred to as the "Three Pillars": metrics, traces, and logs ([Sridharan, 2018](#)). While metrics provide an efficient, quantitative signal of system health (e.g.,

latency, throughput) and traces visualize the structural propagation of requests, it is logging that provides the granular, qualitative context required for *root cause analysis*. In fact, logs serve as the immutable narrative of execution flow, capturing the specific variable states and logic branches necessary to diagnose complex software behaviors that metrics and traces alone cannot explain (Chen and Jiang, 2017; Sridharan, 2018; Yuan et al., 2010).

1.1.2 Observability Challenges From Personal Experience

This critical nature of logging became particularly evident from experience as a research associate intern, where I worked on the profiling and observability of distributed applications. This experience revealed a sharp contrast between the theoretical utility of telemetry and its practical implementation. While automated tracing and metric collection were often robust for monitoring infrastructure health, they frequently lacked the granularity to diagnose domain-specific failures. For instance, standard profiling tools often struggled to isolate performance bottlenecks at the sub-function level in distributed workers, and default metrics were insufficient for detecting anomalies (e.g., only reporting loss for a reinforcement learning job). Consequently, the detection of root causes for critical incidents depended heavily on ad-hoc logging statements manually placed by developers to capture niche execution trajectories, as well as other task-relevant events. This experience highlighted that observability is not merely a systems challenge but a human-centric software engineering challenge, defined by the difficult decisions that developers must make to bridge the gap between generic tooling and application logic.

1.2 The State of Software Logging

Modern software systems produce large amounts of operational data. Logs, which are generated through the logging statements that developers insert within software code, are a key part of such data and help with identifying execution anomalies, preventing issues, and debugging failures (Li et al., 2021a; Shang and Hassan, 2015; Yuan et al., 2010). Prior work proposed machine learning tools to assist developers with logging activities such as what to log, where to log and at which level. However, logging data is unstable (Kabinna et al., 2016) and can change with development goals and performance expectations. For example, during debugging activities, developers typically opt to increase the verbosity of their logs, in order to resolve bugs. Conversely, in another period of the lifetime of a project, developers might decide to reduce the verbosity if their logs become overwhelming, thereby leading to a high performance overhead (Yuan et al., 2014).

Beyond such general instability, modern software projects bring in factors that are not visible from the code alone. In fact, different parts of a system have different responsibilities and purposes (e.g., operational observability vs. performance cost). Additionally, different teams own different code areas and follow their own logging conventions and habits. Furthermore, projects evolve over long periods of time, and logging practices change with that evolution (e.g., early alpha vs. robust long lived code). These factors shape how developers log and therefore affect the performance of automated tools that are trained only on code data.

1.3 Socio-technical Knowledge in Open Source Software

In practice, the components' purposes, the code ownership patterns, and the evolution of the project through time all impact logging practices. For instance, components serving different purposes often choose different logging levels, as OpenStack's object-storage component (Swift) deliberately reduced verbosity to control the operational cost of storing logs ¹, while the compute component (Nova) increased verbosity to improve operational visibility.² Furthermore, different developers may have different logging preferences, which are a social aspect rather than a technical one. For instance, seniority (Rong et al., 2023) and prior experiences across varying domains shape logging habits (e.g., structured key-value logs vs. prose) and lead to varied logging practices within the code pieces of the same software system. Finally, the logging practices change as the software and its ecosystem change. For instance, new standards and tools might require rethinking what to log and at what level. Additionally, shifts in software architecture, deployment approaches, data sensitivity, or performance expectations often lead to changes in logging practices across time.

Taken together, these observations suggest that logging practices are shaped by both technical (e.g., component structure and system evolution) and social/organizational factors (e.g., ownership and developer preferences). In this thesis, we use the term socio-technical knowledge to mean information that captures both the

¹<https://github.com/openstack/swift/pull/15>

²<https://bugs.launchpad.net/nova/+bug/1715785>

software system’s technical structure and evolution (e.g., components, code context, change history) and the social/organizational context around the code (e.g., ownership, team conventions, developer preferences), as it relates to logging practices.

Specifically, we leverage Logging Level Predictors (LLPs) as a case study to examine how socio-technical knowledge affects tools that automate the logging activity. Logging levels are central to logging practices as they control verbosity, influence what surfaces during operations, and affect operational cost. Furthermore, developers often revisit and change logging levels as a system evolves or incidents occur, which makes determining the appropriate logging level a hard and error-prone task (Oliner et al., 2012; Yuan et al., 2010). This mix of importance, difficulty, and frequent change makes LLPs a suitable candidate to study the impact of socio-technical signals on logging practices. That said, our key message of the importance of considering socio-technical knowledge to better support the automation of logging practices is not limited to LLPs and is expected to be relevant to other logging automation tools as well.

While prior studies on logging level prediction (Anu et al., 2019; Heng et al., 2024; Kim et al., 2020; Li et al., 2017a, 2021b) analyzed logging practices and proposed code-powered tools to assist developers, they largely overlooked socio-technical signals that lie beyond the code. Considering these aspects can improve the effectiveness of automation. In this thesis, we study the effect of the following signals on the performance of LLPs in large, modern software systems:

- **Component (responsibility and purpose):** Parts of a system serve different functions, and those functions influence what gets logged and at which level. These characteristics are not directly captured by code-level representations.
- **Ownership (teams and authors):** Teams and groups of authors adopt logging conventions that reflect their experience, preferences, and workflows. Such practices often live as 'tribal knowledge' and are not explicit in the code.
- **Evolution over time:** Projects change across versions, phases, and maturity levels of code. Logging practices shift with these changes.

Specifically, we investigate how these socio-technical signals influence the performance and interpretability of LLPs. Then, we propose and evaluate approaches tailored to harness such signals, with the aim of enhancing LLP effectiveness and providing practical insights for software developers.

1.4 Research Hypothesis

In summary, prior research on logging automation tools led us to formulate the following research hypothesis:

Current automated approaches for supporting logging activities typically rely on leveraging the source code solely. This thesis argues that their performance can be improved by incorporating a broader spectrum of socio-technical knowledge that exists beyond the code, including the purpose of software components, their ownership, and their evolutionary history.

The goal of this thesis is to validate this hypothesis through studying various logging level prediction approaches and across several open source projects. Specifically, we design several approaches to harness the socio-technical knowledge present in open-source software projects. Our work can assist:

- Developers in choosing the suitable logging levels for newly added logging statements, as well as in verbosity calibration for existing logging statements. Hence, improving the quality of the generated logs.
- Managers in predicting faults and understanding their root causes. Thanks to the improved observability provided by high quality logs.

1.5 Contributions

The conceptual contributions of this thesis center around the development of techniques and approaches to demonstrate the value of leveraging socio-technical knowledge (specifically component boundaries, evolutionary history, and code ownership) in assisting developers with automated logging activities. The technical contributions of this thesis focus on the development of specialized modeling strategies and retrieval-augmented approaches to robustly automate the logging level prediction process for large, long-lived, and multi-component software systems. The empirical contributions of this thesis are the extensive application and evaluation of all proposed approaches on large historical datasets from several large-scale open-source systems, quantifying the limitations of traditional code-centric approaches and validating the efficacy of socio-technical signals.

The main contributions of this thesis are as follows:

1.5.1 An Empirical Characterization of the Impact of Software Architecture on Logging Level Prediction

We conducted the first large-scale empirical study on the transferability of logging level Predictors (LLPs) across the architectural boundaries of multi-component software systems. By analyzing five large-scale open-source software systems (e.g., OpenStack, Hadoop), we demonstrated that the one-size-fits-all assumption of global prediction models is flawed. Specifically:

- We quantified the performance degradation of global models, revealing that local (component-specific) models outperform global models on 60% to 100% of components across all performance metrics (AUC, Brier score, and cAIC).
- We introduced and validated the concept of "Peer-Local" modeling, demonstrating that for data-lacking components (e.g., new components), models trained on a mature peer component (i.e., another component from the same software system) outperform models trained on the entire system history.
- We provided evidence that the interpretability of global models is often misleading when applied locally, necessitating component-aware modeling to provide actionable insights to developers.

1.5.2 A Longitudinal Analysis of Concept Drift for Logging Level Prediction

We formalized the impact of software evolution on automated logging tools, identifying time as a critical confounding variable that degrades model reliability. Through

an analysis of long-lived software systems, we established that logging data exhibits significant concept drift, rendering static models obsolete shortly after training. Specifically:

- We quantified the shelf-life of different LLPs, finding that the performance of deep learning LLPs significantly drops after a median of just 1 to 1.5 testing time frames (approx. 2–3 months) due to evolving logging practices.
- We proposed and evaluated a contextual training strategy, proving that forgetting old data (training only on the recent N months) is at least statistically equivalent to "all-knowing" models (training on the full history) while being significantly more computationally and logistically efficient.
- We demonstrated that the drivers of logging level decisions (through feature importance analysis) fluctuate over time, suggesting that explainable AI for logging must be temporally-scoped to be valid.

1.5.3 A Socio-Technical Retrieval for LLM-powered Logging Level Prediction

We proposed a novel methodology to integrate socio-technical knowledge into LLM-powered logging level predictors, moving beyond the standard existing In-Context-Learning sampling strategies. We developed a Multiplex Graph approach that fuses two distinct signals i.e., Code Functionality (what the code does) and Code Ownership (who maintains the code) to optimize In-Context-Learning (ICL) retrieval. Specifically:

- We demonstrated that retrieving in-context examples based on ownership signals alone (files maintained by the same team) statistically significantly enhances the precision of LLM predictions by a median of 2% to 7% compared to state-of-the-art baselines.
- We showed that our fused socio-technical approach enables a small, open-source model (7B parameters) to achieve a median AUC between 0.90 and 0.96, outperforming commercial state-of-the-art models (e.g., GPT-4o, DeepSeek-V3.2) by margins of 17% to 40%.
- We provided empirical evidence that in the context of logging level prediction, precise context-aware retrieval is more critical than model scale, proving that smaller, self-hosted models equipped with socio-technical context can surpass substantially larger proprietary models, thereby offering a cost-effective path for industrial adoption.

1.6 Thesis Structure

The remainder of this thesis is organized as follows: Chapter 2 introduces background concepts related to logging and logging level prediction. Chapter 3 surveys prior work on logging practices and logging automation while identifying key research gaps. Chapter 4 presents the first study on logging level prediction while taking into consideration the potential differences in logging practices from one component to the other. Chapter 5 details our second empirical study on the impact of the evolution of software on the performance of logging level predictors. Chapter 6 introduces multiplex-retrieval, an approach that combines both the functionality

and ownership knowledge to improve the performance of LLM-powered logging level predictors. Finally, Chapter 7 concludes the thesis by summarizing key contributions, discussing practical implications, and outlining promising directions for future research on leveraging socio-technical knowledge in the context of logging automation.

CHAPTER 2

Background

THIS chapter establishes the foundational concepts and technical details relevant to the rest of the thesis. It begins with an overview of software observability, distinguishing logging as the primary mechanism for capturing the immutable narrative of system execution. We then introduce key logging concepts, including the hierarchy of logging levels (verbosity), the anatomy of logging statements, and the critical trade-off between diagnostic visibility and performance overhead. Special attention is given to the socio-technical nature of software development, exploring how factors beyond the code (specifically multi-component architectures, evolutionary history, and code ownership patterns) influence logging practices. We explore the evolution of automated logging support, tracing the progression from static analysis tools to modern data-driven approaches.

The latter sections of the chapter focus extensively on the specific machine learning paradigms employed in this work: Ordinal Regression, Deep Learning architectures for code representation, and Large Language Models (LLMs), with a specific focus on Retrieval-Augmented Generation (RAG) and multiplex graph-based clustering strategies for optimizing in-context learning.

2.1 Software Logging Fundamentals

2.1.1 The Anatomy of a Logging Statement

A logging statement is an imperative instruction manually inserted by developers to capture operational data during software execution. While the specific syntax varies across programming languages and logging libraries (e.g., Log4j, SLF4J), a standard logging statement is composed of three primary components: the logging level, the static message, and optional variables, as shown in Figure 2.1.

- **The Logging Level:** This is a categorical tag attached to the statement to indicate the severity or urgency of the recorded event. It acts as a filtering mechanism, allowing operators to control the verbosity of the output by enabling or suppressing logs based on their importance (e.g., suppressing DEBUG logs in production while retaining ERROR logs). Logging levels are ordered by severity: from *trace* (lowest severity) up to *fatal* (highest severity) in common Java libraries, and from *debug* up to *critical* in Python (see Figure 2.2).

```

    } catch (Exception e) {
-     return; // message logged in renewDT method
+     LOG.error("Exception renewing token" + token + ". Not rescheduled", e);
+     removeFailedDelegationToken(dttr);
    }

```

Figure 2.1: Example of an added logging statement to the Hadoop project¹

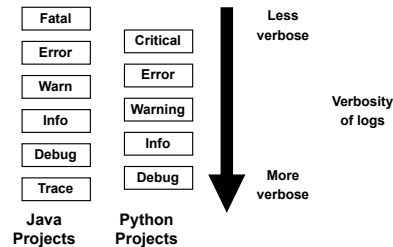


Figure 2.2: logging levels verbosity in Java (Left) and Python (Right) projects

- The Static Message: This is the constant textual component of the log (e.g., "Transaction failed" or "User logged in"), which serves as the human-readable template describing the event's nature.
- The Variables: These are the placeholders within the statement that capture the dynamic system context (e.g., User IDs, error codes, or request latencies) at the moment of execution.

It is important to distinguish between the logging statement and the log entry. On the one hand, the logging statement is a static code artifact that resides within the software repository and is subject to code maintenance and evolution. On the other hand, log entries are the dynamic, runtime data that is generated when the execution flow encounters a logging statement. A single static logging statement can generate millions of unique log entries during the system's operation, each sharing the same static message template but differing in timestamps and variable

¹<https://github.com/apache/hadoop/commit/002dd6968b89ded6a77858ccb50c9b2df074c226>

values. Consequently, while the logging statement represents the developer's intent to observe a behavior, the log entry represents the empirical realization of that behavior in a production environment. This thesis focuses on logging statements rather than log entries.

2.1.2 Current Industry Practices and Tools for Logging

Software developers implement logging continuously during the coding and review phases. In practice, they rely on a combination of integrated development environment (IDE) features, static analysis tools, and modern AI assistants. These tools reduce the manual effort required to format log messages and select appropriate logging levels. In fact, IDEs provide the first layer of logging assistance. Developers frequently use auto-completion features to insert standard logging templates. For example, a developer might type a short shortcut (e.g., `logw`) that the IDE automatically expands into a complete `logger.warn()` invocation. This practice accelerates development and ensures syntactic consistency across a codebase. Furthermore, static analysis tools enforce corporate logging guidelines before code reaches production. Organizations configure linters to scan source code for common logging anti-patterns. These tools can automatically flag issues such as missing variables in parameterized log messages, the use of `System.out.println` instead of a proper logging framework, or the inclusion of sensitive data in explicit text. Consequently, linters act as an automated quality gate during the continuous integration process. More recently, developers have begun using AI-powered coding assistants to generate and refine logging statements. Tools embedded directly in the IDE analyze the surrounding code context to suggest complete log messages and relevant variables.

Additionally, automated AI review assistants scan pull requests in repositories to suggest missing logging statements or recommend verbosity adjustments. These AI tools help bridge the gap between rigid static rules and the contextual needs of specific code blocks.

2.2 Socio-Technical Dimensions of Software Engineering

Software development is inherently a socio-technical endeavor where the resulting artifacts (e.g., Code, Documentation, Models, etc.) are shaped not only by technical requirements but also by the social dynamics of the development teams and the evolutionary history of the project. In the context of logging, these dimensions manifest as distinct signals that exist beyond the artifact/code.

2.2.1 Concept Drift and Software Evolution

Software systems are not static entities as they evolve continuously over long life-cycles to accommodate new requirements, bug fixes, and architectural shifts. This evolution introduces a phenomenon known in machine learning as Concept Drift ([G.Ditzler et al., 2015](#); [Gama et al., 2004](#)), where the statistical properties of the target variable change over time, degrading the performance of predictive models trained on historical data.

In the domain of logging, data is shown to be unstable. In fact, empirical studies indicate that 20% to 45% of the logging statements undergo modifications during

their lifetime (Kabinna et al., 2016). This instability is driven by shifting development goals as developers often increase log verbosity during active debugging or early development phases (e.g., alpha releases) to maximize visibility, only to later decrease verbosity in mature, long-lived code to reduce performance overhead and storage costs. Consequently, a machine learning model trained on logging practices made during one phase of the software’s lifecycle may fail to generalize to future phases, as the ground truth for what constitutes an appropriate logging level is temporally dependent.

2.2.2 Component-Based Architecture

Modern large-scale software systems are typically architected as multi-component systems where distinct subsystems are responsible for specific functional domains (e.g., storage, compute, networking). This architectural separation may lead to heterogeneous logging practices across the same system. Specifically, different components serve different operational purposes, which dictate their logging requirements. For instance, a storage component (e.g., OpenStack Swift) may prioritize low verbosity to minimize the I/O cost, whereas a compute scheduling component (e.g., OpenStack Nova) may prioritize high verbosity to trace complex state transitions. This variation implies that logging practices are often local rather than global. Treating a multi-component system as a uniform entity might obscure these distinct, purpose-driven logging practices.

2.2.3 Code Ownership and Developer Conventions

Code ownership refers to the relationship between software artifacts and the specific developers or teams responsible for their maintenance. Research suggests that logging is heavily influenced by the tacit knowledge and social conventions of the owning team rather than explicit technical standards (Pecchia et al., 2015). Typically, different teams possess unique logging preferences/styles shaped by their collective experience, seniority, and domain background (Rong et al., 2023). For example, a security team maintaining files across multiple components will likely enforce a consistent, rigorous logging style for auditability, whereas a UI team might adopt a different convention focused on user interactions. These social clusters often transcend directory structures and files that are architecturally distant but maintained by the same group of authors can exhibit higher similarity in logging practices than files in the same directory maintained by different authors. Capturing these ownership signals is therefore essential for understanding the inconsistencies in logging datasets that purely technical analysis cannot explain.

2.3 AI/ML Paradigms in Logging Level Prediction

The automation of logging activities has evolved in parallel with advancements in artificial intelligence. While early approaches relied on static analysis and heuristics (Yuan et al., 2011), the field has progressively adopted data-driven techniques to model the complex dependencies between source code and logging practices. This section outlines the three primary machine learning paradigms employed in

this domain: Classical Machine Learning (specifically Ordinal Regression), Deep Learning, and Large Language Models (LLMs).

2.3.1 Classical Machine Learning and Ordinal Regression

The earliest generation of automated tools, often referred to in our thesis as Shallow logging level predictors (aka. Shallow-LLP), approached the problem through the lens of feature engineering. These models rely on extracting a set of manually defined metrics from the source code, including structural features (e.g., cyclomatic complexity), textual features (e.g., length of the log message), and evolutionary features (e.g., code churn).

A critical distinction in this paradigm is the modeling of the target variable. Unlike standard classification tasks where classes are nominal (e.g., "Buggy" vs. "Not-Buggy"), logging levels possess an inherent order (TRACE < DEBUG < INFO < WARN < ERROR < FATAL). To respect this order, researchers employ Ordinal Regression models (such as Ordinal Logistic Regression) rather than multi-class classifiers. This approach ensures that the penalty for misclassifying a "DEBUG" statement as "INFO" is lower than misclassifying it as "FATAL," thereby preserving the semantic severity scale of the logging framework.

2.3.2 Deep Learning and Semantic Code Representation

To overcome the limitations of manual feature engineering, research shifted toward Deep Learning (DL) architectures capable of learning representations directly from

raw code. These approaches, referred to in this thesis as Deep logging level Predictors (DL-LLP), treat source code not as a bag of metrics but as a sequence of semantic and syntactic tokens.

Typically, these models utilize embedding techniques to encode two distinct contexts:

- **Syntactic Context:** Represented by sequences of Abstract Syntax Tree (AST) nodes surrounding the logging statement, capturing the control flow and structural logic.
- **Semantic Context:** Represented by the natural language tokens within the log message itself and the surrounding code identifiers.

By fusing these representations, deep learning models can capture complex, non-linear patterns—such as the correlation between specific exception types and "ERROR" levels—that shallow models frequently miss.

2.3.3 Large Language Models and In-Context Learning

The advent of Generative AI has introduced a new paradigm based on Large Language Models (LLMs). Unlike discriminative models that classify an input into a pre-defined label, an LLM-powered predictor (e.g., LLM-powered LLP) treats logging level prediction as a text generation task.

In this paradigm, the model is presented with a prompt containing the source code where the logging level is masked, along with the log message. The model then generates the appropriate logging level token (e.g., generating

the word "INFO") as a completion of the prompt. A key capability of this approach is In-Context Learning (ICL), or few-shot prompting, where the model is provided with a small set of correct logging examples within the LLM's context size. This allows the model to adapt to specific project conventions or domain contexts without the need for fine-tuning (e.g., LoRA) or extensive model retraining.

Establishing the Ground Truth Across Paradigms Across all three of these machine learning paradigms, establishing a reliable ground truth logging level for training and evaluation remains a universal challenge. To train and evaluate the predictive models in this thesis, we define the ground truth as the most recent and stable logging level found in the version control history, rather than the level assigned during the initial commit. Developers frequently revise logging statements during the code review process ([Kabinna et al., 2016](#)). Furthermore, continuous integration (CI) pipelines and production monitoring often reveal performance bottlenecks caused by overly verbose logging, prompting subsequent changes. Consequently, the initial logging level chosen by a developer may be suboptimal. By extracting the final, stable logging level of a statement after it has survived the CI/CD pipeline and community review, we ensure that our ground truth reflects the established, and tested consensus of the project.

CHAPTER 3

Literature Survey

IN recent years, the increasing scale and distributed nature of modern software systems have heightened the reliance on observability for maintaining system reliability and diagnosability. Software logging serves as the primary mechanism for this observability, providing the immutable narrative of execution flow required to identify anomalies and debug failures. However, the quality of this data is heavily dependent on the often ad-hoc decisions made by developers, spurring a significant body of research into both the empirical characterization of logging practices and the development of automated support tools. This chapter presents a comprehensive literature review organized along two primary research directions (see Figure 3.1). First, we examine empirical studies on logging practices, covering (i) developer behavior and framework choices , (ii) the challenges

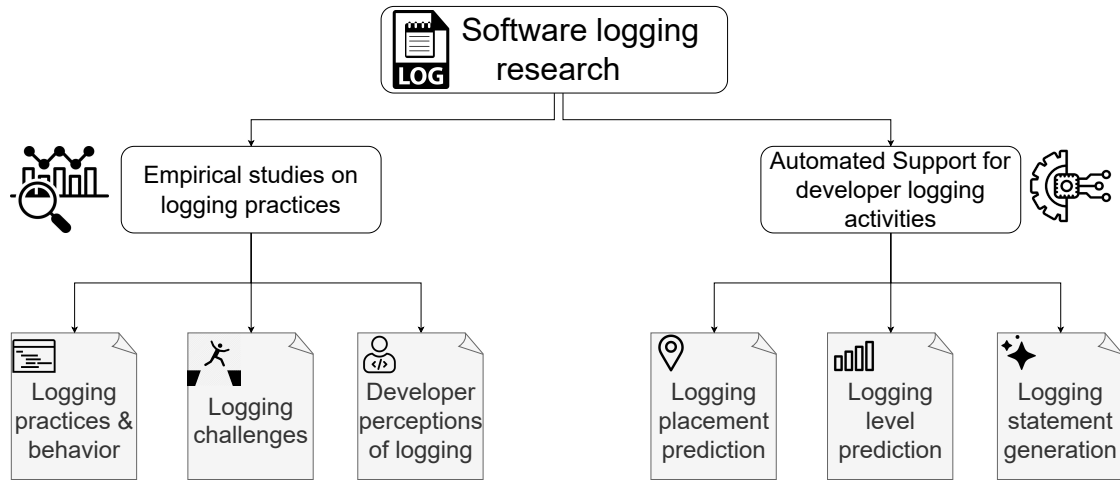


Figure 3.1: Overview of the discussed research directions

of log maintenance and instability , and (iii) the qualitative perceptions of developers regarding the costs and benefits of logging. Second, we survey automated support for logging decisions, detailing advancements in predicting log placement, generating and enriching logging statements , and predicting appropriate logging levels. Finally, we synthesize these works to identify the limitations of current code-centric approaches, positioning our contributions to address critical gaps regarding the influence of socio-technical factors on logging automation.

3.1 Empirical studies on logging practices

Understanding logging practices is important for effective software maintenance and debugging. Research in this area covers empirical studies that aim to understand how developers implement, evolve and maintain logging in real-world systems and the implications of such logging practices.

3.1.1 Developer logging practices and behavior

Several empirical studies aimed to characterize how software developers approach logging activities. For instance, [Fu et al. \(2014\)](#) explored logging practices within Microsoft's large industrial software systems. Their investigation combined source code analysis of two major Microsoft systems (totaling millions of lines of code) with qualitative insights gathered from 54 experienced engineers through surveys. [Fu et al. \(2014\)](#) revealed a clear yet limited set of scenarios in which developers commonly log, such as handling exceptions, verifying return values from function calls, and critical logic branches. In a complementary study, [Pecchia et al. \(2015\)](#) analyzed logging practices in an industrial safety-critical system context. Specifically, they conducted detailed inspections of source code, analyzed more than two million log entries collected during operational runs, and engaged with development teams. [Pecchia et al. \(2015\)](#) discovered logging to be largely informal, driven more by individual developers' expertise and preferences rather than by guidelines or standards. Addressing a different angle, [Chen and Jiang \(2020\)](#) focused on logging framework selection practices in large open-source Java projects. By analyzing the GitHub repositories of such high-profile projects such as Apache Software Foundation projects, they characterized the used logging frameworks (e.g., Log4j, Logback, or custom-built solutions) and the rationale behind the use of those specific frameworks. They observed a notable trend where projects favored creating custom logging solutions tailored for performance optimization and flexibility, especially in highly active and frequently changing codebases. A recent study by [Foalem et al. \(2024\)](#) extended logging research into the emerging domain of machine learning-based software systems. By examining logging practices across 110

ML-based software projects, they investigated the frequency, location, and content of logging statements within ML and non-ML components. Their quantitative analysis revealed significant differences in logging behavior between ML and non-ML contexts. ML components tend to have specialized logging practices focused on capturing information about model performance metrics, hyperparameter settings, and data processing steps, which reflects the unique operational needs of ML systems. [Patel et al. \(2022\)](#) conducted an extensive qualitative analysis of logging practices within the Linux kernel. Through a mixed-method approach involving repository mining and manual inspection, they studied logging statements across kernel subsystems, focusing on where and why logging occurred. [Patel et al. \(2022\)](#) highlighted that Linux kernel logging predominantly facilitated debugging, performance analysis, and anomaly detection, serving crucial roles given the kernel's complexity. [Chen et al. \(2025\)](#) examined how logging is used in GPU-accelerated deep learning projects by analyzing logging statements in 33 CUDA-based DL repositories. The authors found that most logs are written during model training and that logging in these systems is used primarily for monitoring and tracking model-related information (e.g., loss or resource usage). The study by [Chen et al. \(2025\)](#) highlights that developers mix general-purpose logging frameworks and DL-specific logging tools, and provides guidance on when to use each. Finally, the importance of studying logging practices from a broad perspective was highlighted by [Yuan et al. \(2010\)](#), who analyzed real-world failure data from several widely used software projects (e.g., Hadoop). Their study revealed frequent and critical gaps in logging coverage, noting instances where software failures occurred without generating any log entries.

3.1.2 Logging challenges, issues, and maintenance

Beyond examining developer practices, several empirical studies highlighted critical challenges and issues developers encounter when logging. Specifically, problems related to logging anti-patterns, the complexity of log maintenance, and issues related to software evolution, which all hurt software maintainability and reliability. For example, a study by [Pecchia et al. \(2013\)](#) systematically explored challenges in log instrumentation for dependable systems. This extensive analysis of event logs from industrial systems identified several recurring logging problems, notably missing or insufficient contextual information and inconsistencies in log messages. The authors argued that many logs lacked sufficient context or clarity to point to the root-cause of failures. [Pecchia et al. \(2013\)](#) developed a rule-based approach aimed at improving the effectiveness of logs for failure debugging activities. Their approach provided tangible advice, such as specifying key runtime variables to log during exceptional conditions and standardizing log message formats to streamline automated parsing, thus directly tackling widespread logging issues. Another logging issue was reported by [Li et al. \(2022\)](#) who conducted an extensive empirical analysis targeting duplicate logging statements (containing identical static messages) throughout software systems. Their large-scale analysis of Apache projects revealed widespread occurrences of logging duplication, which not only contributed to redundancy but complicated automated log analysis due to inflated log volumes and duplicated diagnostic information. Exploring maintenance difficulties, [Kabinna et al. \(2016\)](#) investigated challenges associated with logging framework migrations in software projects. By analyzing Apache Software Foundation (ASF) repositories, they observed frequent migrations between logging frameworks (e.g., from Log4j

to SLF4J or Logback). Such migrations, though aimed at improving logging capabilities or performance, often introduced substantial effort and risks of incomplete or incorrect migration. In another work, [Kabinna et al. \(2016\)](#) empirically investigated the stability of logging statements and their implications on log maintenance and log processing tools. Their study analyzed four large open-source projects, revealing that 20% to 45% of logging statements undergo changes over their lifetime. Notably, many logging statements experience their first modification shortly after their introduction (often within a median interval of just 1–17 days). Another work by [Chen and Jiang \(2019\)](#) targeted logging-related issues through an analysis of historical Logging-Code-Issue-Introducing (LCII) changes across six widely-used Java open-source projects. By mining historical logging code changes, [Chen and Jiang \(2019\)](#) found that logging-related fixes often occur both in co-changed and independently changed code blocks. Additionally, their analysis revealed that the complexity involved in fixing logging code issues was similar to that of regular logging updates. Recently, [Foalem et al. \(2025\)](#) targeted the issue of responsible AI by investigating logging practices in 85 open-source ML projects (with 20 responsible-AI libraries) and conducted a practitioner survey to see how well developers log information related to fairness, transparency, privacy, and safety. They found a significant gap in current logging practices as important Responsible-AI metrics (e.g. fairness metrics, explainability indicators like SHAP values) are rarely logged in these projects. For example, while 89% of the analyzed instrumented calls by [Foalem et al. \(2025\)](#) were related to privacy (due to use of privacy libraries), only 2% were related to fairness which is an uneven emphasis that leaves many fairness aspects unmonitored.

3.1.3 Developer perceptions and experiences towards/with logging

Empirical research on logging practices has not only analyzed software artifacts and code repositories but also explored logging from developers' personal perspectives. One contribution was made by [Li et al. \(2021a\)](#), who conducted an extensive qualitative investigation into developers' experiences and perceptions regarding logging. This study combined multiple approaches, including surveys involving 66 software developers and an in-depth manual examination of 223 logging-related issue reports drawn from popular open-source systems. [Li et al. \(2021a\)](#) findings revealed that developers strongly recognize logging's considerable benefits for debugging, anomaly detection, and system monitoring. Logging is indeed perceived as indispensable for gaining visibility into system behavior, especially during failure diagnosis and troubleshooting scenarios. However, their study also unveiled the significant perceived costs that are associated with logging, notably increased code complexity, potential performance overhead, and substantial maintenance burdens. More recently, [Rong et al. \(2023\)](#) conducted a comprehensive mixed-method study investigating industrial software developers' logging practices, intentions, and concerns (i.e., I&Cs). Their research confirmed previously reported challenges related to logging, notably highlighting troubleshooting and monitoring as the primary logging intentions and performance overheads of I/O and storage as the most significant concerns. Furthermore, [Rong et al. \(2023\)](#) found that developers' professional profiles significantly influenced their logging practices.

Prior studies provide insights into how logging is perceived, practiced, and experienced by developers in both open-source and industrial projects. Collectively, this research area emphasizes the developer-recognized necessity and utility of logging, and the significant challenges that developers face when incorporating logging into their everyday practices. Developers consistently highlight issues related to complexity, logging verbosity choices, maintenance burdens, and the lack of effective guidelines. Such insights and concerns have directly informed and motivated subsequent research efforts aimed at creating practical, effective logging support tools that address real-world developer pain points and facilitate more consistent and efficient logging practices.

3.2 Automated Support for developer logging decisions

The complexity and variability of logging practices have led researchers to develop automated tools, approaches and models to assist developers with logging decisions. These efforts range from log placement predictions to logging level prediction and automatic logging statement generation.

3.2.1 Prediction of logging placement

Identifying where logging statements should be placed within a codebase is an important yet tricky decision that developers face regularly. Research addressing this issue focus primarily on predicting suitable locations for logging statements based

on patterns learned from existing logging practices, typically leveraging machine learning and static analysis methods. An early contribution in this domain was the work by [Zhu et al. \(2015\)](#), who proposed LogAdvisor, a framework that leverages machine learning to automatically recommend logging placement. Their work began by thoroughly analyzing existing log locations within several large-scale Microsoft software systems. Next, they extracted features such as control flow statements, exception handling blocks, and commit histories. Then, they trained classifiers to distinguish code blocks that should include logging statements from those that should not. Their evaluation demonstrated promising accuracy, which suggests that the typically manual logging decisions made by experienced developers could be systematically replicated and even improved upon through automation. Building upon that foundation, [Cândido et al. \(2021\)](#) addressed several limitations in the automation of logging placement research, particularly emphasizing dataset imbalance, where some locations (e.g., classes) have more dense logging than other locations. Specifically, they explored multiple machine learning models, applied to industrial software codebases. [Cândido et al. \(2021\)](#) results demonstrated that balancing the training dataset significantly improved prediction accuracy. Another contribution is the work of [Li et al. \(2020\)](#), who took a different perspective by examining log placement prediction at the block level. Specifically, their deep learning model leveraging different block-level features (syntactic, semantic, and structural) showed that syntactic block features for log placement consistently achieved the highest accuracy, outperforming baseline methods significantly. Similarly, [Mastropaolo et al. \(2024\)](#) introduced LEONID, which is a deep learning model built on

top of the T5 (Raffel et al., 2020) text-to-text transformer to create logging statements and decide on their placement given a Java method. LEONID first determines whether a method needs a logging statement in the first place, then inserts as much logging statements as needed. The evaluation of LEONID across multiple open-source projects reveals that it can detect the need for logging statements reliably and can accurately place the suggested logging statements in the input method code. Recently, Tan et al. (2025) introduced AL-Bench, a large-scale benchmark dataset and evaluation methodology for automated logging tools. It focuses on systematically evaluating how well tools can predict where to insert logging statements under realistic conditions. AL-Bench contributes a diverse dataset from 10 popular projects and proposes a novel dynamic evaluation that compares the runtime logs produced by tool-generated statements against ground truth logs. Using this benchmark, the authors discovered significant drops in performance when moving from static code-level evaluation to runtime evaluation. For example, on average the accuracy of state-of-the-art tools in predicting log positions dropped by 37%, and for logging levels by 23%, compared to their originally reported results. Moreover, 20%–84% of generated logging statements failed to compile or execute properly, indicating robustness issues. Beyond the notion of log placement, Li et al. (2017b) proposed an approach focused on logging timing by providing “just-in-time” suggestions for log changes. By examining and classifying reasons for log modifications such as block changes, logging improvements, dependency-driven adjustments, and logging issues, the authors identified a comprehensive set of explanatory features

capturing both historical log changes and characteristics of the current code snapshot. Using these features, [Li et al. \(2017b\)](#) trained random forest classifiers to predict the necessity of log changes at the exact time developers committed new code. Evaluated on four well-known open-source project, their classifiers achieved accuracies ranging from 0.76 to 0.82 within projects and 0.76 to 0.80 across projects, demonstrating significant potential in assisting developers by immediately flagging code commits that likely require logging adjustments.

3.2.2 Automated logging statement generation and enrichment

Beyond recommending log placement, another research direction targets the problem of automatically generating complete logging statements, including both the textual content and logged variables. Automating this aspect of logging reduces developer effort, ensures consistency, and potentially enhances the informativeness of logs, which directly benefits tasks such as debugging and failure diagnosis. One of the contributions in automatic logging statement generation was LogEnhancer proposed by [Yuan et al. \(2011\)](#). Recognizing the common inadequacies of existing log messages, such as missing context and insufficient runtime data, the authors developed a static analysis-based approach to automatically enrich logs by adding important runtime variables into logging statements. Their method identified causally related runtime information needed for effective diagnosis and automatically modified logging statements to include additional context. Evaluated across several widely-used open-source systems, LogEnhancer significantly improved the informativeness and diagnostic utility of the logs. [Liu et al. \(2020\)](#)

explored the generation of descriptive texts in logging statements using a retrieval-based question-answering (Q&A) approach. Specifically, they framed log message generation as a retrieval task, proposing two strategies: Code&Code, where both queries and answers are derived from source code snippets, and Code&Log, where queries originate from code snippets and answers come from existing log messages. They investigated various similarity-measurement techniques, comparing traditional information retrieval-based methods against more advanced neural network-based approaches. They systematically evaluated these methods using automatic metrics and human assessments, uncovering several practical insights into the effectiveness of retrieval-based log generation. Furthermore, they constructed and released a substantial dataset containing over 138,000 valid log messages extracted from 85 Apache Java projects, supporting future research into automated logging statement generation. Furthering this direction, [Ding et al. \(2022\)](#) proposed LoGenText, a neural machine translation based approach aimed at automatically generating logging texts directly from source code. LoGenText leverages transformer-based sequence-to-sequence models, which are neural network architectures known for effective natural language translation. The approach generates logging texts by translating the preceding source code (the "pre-log code") and considers additional contextual information, including the AST structure surrounding the logging statement, subsequent code snippets ("post-log code"), and logging texts from similar code. Evaluations across ten diverse open-source Java projects

demonstrated that LoGenText substantially outperformed existing methods, achieving significantly higher BLEU (up to 41.8) and ROUGE-L (up to 53.9) scores compared to previous state-of-the-art approaches. Additionally, human evaluations involving 42 software developers confirmed the practical utility and high quality of the logging texts generated by LoGenText. Building on their previous LoGenText approach, [Ding et al. \(2023\)](#) further advanced automated logging text generation by proposing LoGenText-Plus, an enhanced two-stage neural machine translation NMT-based approach. While LoGenText directly translated source code into logging texts, LoGenText-Plus introduced a decomposition strategy that first generates the syntactic template of the logging text, then generates the complete logging text based on both the source code and the generated template. Specifically, LoGenText-Plus leverages two separate Transformer-based NMT models: the first model predicts a high-level syntactic template of the target logging message, and the second model utilizes this generated template alongside the source code as inputs to produce the final logging text. Evaluated across the same 10 diverse open-source Java projects, LoGenText-Plus demonstrated clear performance improvements over the original LoGenText approach, surpassing it in 9 out of the 10 projects studied. A human evaluation involving software developers further validated these improvements, confirming that logging texts generated by LoGenText-Plus exhibit notably higher quality compared to those produced by LoGenText and the prior state-of-the-art baseline approach. Inspired by breakthroughs in generative language models, several studies have explored using large language models (LLMs) to automate logging tasks comprehensively. For instance, [Xu et al. \(2024\)](#) introduced UniLog,

an LLM-powered solution that employs in-context prompting to automatically generate log messages, recommend appropriate logging statement placements, and predict their verbosity levels. By leveraging a fine-tuned LLM trained on annotated logging datasets, UniLog captures the semantic intent of logging contexts and generates high-quality logging statements that closely matched developers' logging practices. The evaluation of UniLog indicated strong predictive performance. Most recently, [Li et al. \(2024\)](#) conducted a study on the application of different LLMs for the generation of logging statements. The authors intended to explore the potential of LLMs in automating the logging statement generation process. To do so, Li et al ([Li et al., 2024](#)) created a dataset called LogBench, which consists of two parts: LogBench-O and LogBench-T. LogBench-O contains logging statements derived from 3,870 methods collected from GitHub repositories, while LogBench-T consists of transformed (using Condition-Swap for instance), thus, unseen code instances from LogBench-O, providing a test set for evaluating generalization capabilities. LogBench is then used to evaluate the performance of 13 top-performing LLMs, with model sizes ranging from 60 million to 405 billion parameters. These models were assessed for their ability to generate high quality logging statements. The results revealed that, while LLMs performed reasonably well in determining logging levels and variables, they achieved a maximum BLEU score of only 0.249.

3.2.3 Prediction of logging levels

3.2.3.1 General survey

Historically, practitioners relied on manual guidelines and static analysis tools to manage logging levels ([Fu et al., 2014](#)). Teams typically used rule-based linters and

peer code reviews to enforce basic verbosity standards. Early automated approaches focused primarily on anomaly detection rather than direct prediction. These tools used static heuristics to flag inconsistent log levels across structurally identical code snippets (Yuan et al., 2012). However, these strict rule-based methods struggled to adapt to the evolving, project-wide context of large software systems. Consequently, researchers transitioned to statistical models.

To address these limitations, researchers transitioned from rigid heuristics to statistical models. Rather than simply detecting anomalies, these new models aimed to directly predict the appropriate logging level (e.g., TRACE, DEBUG, INFO, WARN, or ERROR) for new statements. An early machine learning contribution was made by Li et al. (2017a) who introduced an approach we refer to in this thesis as the Shallow logging level Predictor (Shallow-LLP). This approach leveraged an ordinal logistic regression model, specifically designed to predict logging levels using a set of static, change-related and historical code features (e.g., log message length, logging statement churn, number of revisions in history). The empirical evaluation of the Shallow-LLP across several large open-source Java projects demonstrates strong predictive performance. Additionally, the Shallow-LLP's interpretability provided insights into factors influencing developers' logging level choices, such as file-level logging metrics (e.g., average logging level in file). Another study that leveraged, classic machine learning models was the one proposed by Anu et al. (2019) who introduced `VerbosityLevelDirector`, an automated approach based on random-forest designed to help developers assign appropriate logging levels based on logging intention embedded within the code context. `VerbosityLevelDirector` leverages contextual features such as triggered methods, exception types, logging

messages, post-processing code actions, and related comments. The evaluation of VerboseLevelDirector on open-source Apache projects, showed notably higher accuracy in predicting log verbosity levels compared to baseline methods. Meanwhile, [Kim et al. \(2020\)](#) introduced a new perspective for recommending appropriate logging levels based on the semantic and syntactic context of logging statements. Their approach builds two feature vectors: a semantic vector that quantifies (using a word2vec model) the similarity between logging levels and the terms that are associated with such levels (e.g., 'exception' or 'failure' for ERROR levels), and a syntactic vector capturing structural context surrounding the logging statement in the code (e.g., average logging level in a class). Using these combined semantic and syntactic vectors, [Kim et al. \(2020\)](#) employed and compared four different machine learning classifiers: K-nearest neighbors (KNN), Random Forest, Support Vector Machine (SVM), and Decision Trees. Among these, the Random Forest classifier yielded the best performance. Moreover, developers accepted the recommender's suggested logging level changes in 72% of the cases. More recently, [Li et al. \(2021b\)](#) introduced the first deep learning take on logging level prediction. Their approach which we refer to in this thesis as DL-LLP leveraged a neural network architecture that encoded syntactic and semantic contexts surrounding logging statements as well as the contents of the logging statements, significantly improved the prediction accuracy of LLPs. The DL-LLP encoded sequences of abstract syntax tree (AST) nodes from the containing method and the natural language tokens of the log message, capturing deeper contextual information about the logging decision. [Li et al. \(2021b\)](#) evaluation demonstrated superior performance compared to shallow methods. Finally, the breakthroughs in generative large language models (LLMs)

motivated [Heng et al. \(2024\)](#) work that explored logging level prediction using an LLM-powered approach (referred to as LLM-powered LLP in this thesis), employing various LLMs with different sizes (e.g., CodeLlama-7B). Given a source-code snippet (where the logging statement is masked) as a context, and the contents of the log message, the LLM predicts the appropriate logging level by generating an appropriate textual response. This generative approach yielded promising results, showcasing the adaptability and semantic awareness of modern generative LLMs, especially the CodeLlama-7B model that performed the best.

Most prior efforts for automating logging activities are code-powered and overlook the socio-technical context of modern software projects. First, they do not account for the long maintenance lifetimes and the resulting changes in logging practices over time. They also omit that modern software is multi-component, thus, missing component-specific differences in logging approaches. Finally, they ignore ownership and developer conventions differences (e.g., style, seniority) that can shape how logs are written. This gap motivates our work as we aim to study how socio-technical signals impact the performance of LLPs in large, modern software systems.

Since we use logging level predictors (LLPs) as our case study for logging automation, the next section provides detailed background on the logging level prediction models leveraged in this thesis.

3.2.3.2 Baseline logging level predictors

In order to assist developers with the logging level decision, machine learning models were proposed by prior research (Anu et al., 2019; Heng et al., 2024; Kim et al., 2020; Li et al., 2017a, 2021b; Xu et al., 2024). While we discussed all these approaches in detail in Section 3.2.3, we specifically focus our empirical analyses in this thesis on three representative, state-of-the-art approaches: the shallow logging level predictor (aka. Shallow-LLP) suggested by Li et al. (2017a), the state-of-the-art deep learning logging level prediction model (aka. DL-LLP) suggested by Li et al. (2021b), in addition to the state-of-the-art LLM-powered LLP (aka. LLM-powered LLP) suggested by Heng et al. (2024). The first two LLP models predict an ordinal variable (i.e., ordered categorical variable) ranging from 1 to N, where N is the number of logging levels supported by the logging framework (e.g., N=6 for Log4j). Meanwhile, the LLM-powered LLP leverages the CodeLLama-7B model, and operates by masking logging statements within the source code context, then providing both the masked code context and the corresponding log message as input to the LLM.

We selected these three models (Shallow-LLP, DL-LLP, and LLM-powered LLP) because each represents the state-of-the-art in its respective category, providing robust and diverse baselines for evaluating LLP performance. Specifically, the Shallow-LLP (Li et al., 2017a) offers an optimal balance between predictive performance and interpretability, making it suitable for understanding the drivers of logging level decisions across components and over time. The DL-LLP (Li et al., 2021b), unlike the shallow model, does not rely on static code features (e.g., cyclomatic complexity).

Instead, it leverages code context by directly encoding sequences of AST nodes surrounding the logging statement along with the log message tokens, thus providing an overhaul to the LLP design and capturing deeper contextual patterns. Lastly, the LLM-powered LLP, based on the CodeLlama-7B model ([Heng et al., 2024](#)), introduces a generative approach, leveraging advancements in natural language understanding and generation, which offers improved flexibility and adaptability in logging level predictions without relying on manually engineered or pre-defined code features. This combination ensures that our thesis has comprehensive coverage of the spectrum of existing approaches and supports a thorough exploration of modern software's socio-technical signals influence various logging level prediction methodologies.

CHAPTER 4

An Empirical Study on Logging Level Prediction for Multi-component Systems

This chapter is published in the Transactions on Software Engineering journal (TSE) ([Ouatiti et al., 2023](#)).

LOGGING statements are used to trace the execution of a software system. Practitioners leverage different logging information (e.g., the content of a log message) to decide for each logging statement an appropriate logging level, which is leveraged to adjust the verbosity of logs so that only important log messages are traced. Deciding for the logging level can be done differently from one to another component of a multi-component system, such as OpenStack and its 28 components. For example, a component might aim for increasing the verbosity of its log messages, while another component for the same multi-component system

might aim at decreasing such a verbosity. Such different logging strategies can exist since each component can be developed and maintained by a different team. While a prior work leveraged an ordinal regression model to recommend the appropriate logging level for a new logging statement, their evaluation did not consider the particularities that each component can have within a multi-component system. For instance, their model might not perform well at each component level of a multi-component system. The same model's interpretability can mislead the developers of each component that has its unique logging strategy. In this chapter, we quantify the impact of the particularities of each component of a multi-component system on the performance and interpretability of the logging level prediction model of prior work. We observe that the performance of the logging level prediction models that are trained at the whole project level (aka., global models) have lower performances (AUC) on 72% to 100% of the components of our five evaluated multi-component software systems, compared to the same models when evaluated on the whole multi-component system. We observe that the models that are trained at the component level (aka., local models) statistically outperform the global model on 33% to 77% of the components of our evaluated multi-component software systems. Furthermore, we observe that the rankings of the most important features that are obtained from the global models are statistically different from the feature importance rankings of 50% to 87% of the local models of our evaluated multi-component software systems. Finally, we observe that 60% and 35% of the Spring and OpenStack components do not have enough data points to train their own local models (aka., data lacking components). Leveraging a peer-local model for such type of components is more promising than using the global model.

4.1 Introduction

Developers insert logging statements into the source code of a software system to collect its runtime information. Typically, a logging statement consists of a logging function and its parameters, which include a text message, variables, and a verbosity level (e.g., fatal/error/info) that specifies the severity of the logged events (i.e., Log (level, “logging message”, variable)). The logging levels are ordered by their level of verbosity from Trace (most verbose) to Fatal (least verbose) in most common Java logging libraries and they range from Debug to Critical in Python projects. The logs provide valuable information for different stakeholders, such as software operators to spot abnormal execution (Li et al., 2021a; Shang and Hassan, 2015; Yuan et al., 2010), developers to diagnose failures and to better maintain a software system (Li et al., 2021a; Lin et al., 2018; Yuan et al., 2011), and release managers to assess deployment activities (Chen et al., 2019; El-Sayed et al., 2017; Xu et al., 2018).

However, assigning the appropriate logging level for a logging statement is an important activity (Li et al., 2017a; Pecchia et al., 2015). For instance, the lack of logging causes a lack of information about the runtime and a reduced ability for diagnosis (Yuan et al., 2010). Logging too much, however, causes system overhead and makes logs full of noisy data and challenging to analyze (Yuan et al., 2014). Thus, coupled with the framework logging level (works as a threshold), logging levels allow the suppressing of lower level (more verbose) log messages from being traced during the execution of a system. For example, if a developer sets the verbosity level at the “warn” level, only the logging statements with the “warn” level or less verbose logging level (“error” and “fatal”) would be traced in a log file.

Identifying the appropriate logging level for a logging statement is a challenging task (Oliner et al., 2012; Yuan et al., 2010), since the developers cannot be sure how the code will be used in later stages of the development activity (Oliner et al., 2012). Consequently, developers spend much effort adjusting the logging levels of their logging statements (Yuan et al., 2010).

The logging practices (e.g., the length of a log message, the variables to trace) as well as choosing the appropriate logging level can be different from one to another component of a multi-component system. That is since each component is typically developed by a different team, which can follow specific logging strategies that depend on the purpose of the component. A typical example of such a difference is manifested by two (i.e., ‘Swift’ and ‘Nova’) of the 28 components of OpenStack, in which the ‘Swift’ component aimed at reducing the verbosity of logging levels as the logs were growing fast without any execution anomaly¹. That was the opposite for the ‘Nova’ component, which aimed at increasing the verbosity to have detailed monitoring².

While an ordinal regression machine learning model to predict the logging level for a new logging statement was proposed by Li et al. (2017a), their study does not take into account the differences that exist between the components of a multi-component system. Li et al.’s model is an ordinal regression model – which is an extension of logistic regression for ordinal dependent variables – that uses different metrics related to a new logging statement to predict its appropriate logging level (shown in Figure 1). The metrics of Li et al. consider five dimensions; i.e., logging statement metrics (e.g., number of variables), containing bloc metrics (e.g., number

¹<https://github.com/openstack/swift/pull/15>

²<https://bugs.launchpad.net/nova/+bug/1715785>

of lines of code in the bloc), file metrics (e.g., logging statement density), change metrics (e.g., logging statement churn) and historical metrics (e.g., number of revisions in history). While that model might perform well for some components, its performance can be as low as a random guess for other components. Similarly, that model might mislead its users when leveraged for interpretation at the component level.

Therefore, we empirically quantify in this chapter the impact of the variation between different components on the performance and interpretation of the logging level prediction model of [Li et al. \(2017a\)](#), so practitioners can better understand how to leverage that model to predict logging level in the context of multi-component software systems. While we expect that machine learning models that are trained on focused data (i.e., single component) will perform differently compared to the models that are trained on heterogeneous data (i.e., from different components), the objective of our chapter is to quantify the differences between the performance and interpretability of these two types of models. As prior studies ([Fu and Menzies, 2017](#); [Rudin, 2019](#); [Liu et al., 2018](#)) suggest using simple and interpretable models over complex ones, we focus on the ordinal regression model that is proposed by [Li et al. \(2017a\)](#), rather than using a complex one (e.g., a deep learning model).

In particular, we first quantify how a model that is trained using all the components data (aka., *global model*) performs on each component of a multi-component system. We also compare the global model to models that are trained at a component level (aka., *local models*) in terms of performance and interpretability. Finally,

we compare the global and each local model on data-lacking components (i.e., components with few data points). We summarize our contribution in the following research questions:

RQ1. How global models perform at the component level?

While the Hadoop, Spring, OpenStack, Jupyter and Elasticsearch global model show an AUC of 75%, 76%, 77%, 81% and 76% respectively, that global model shows a median AUC of 71.5%, 73.5%, 75%, 74% and 72% and as a low AUC as 69%, 67%, 64%, 65% and 64% when evaluated at each component level. We observe that the global model performs statistically better on only 12.5% (1 out of 8) and 11% (2 out of 18) of Spring and OpenStack components when compared to the same global model tested on the whole multi-component system.

RQ2. How local models perform on the component level compared to the global model?

60%, 75%, 77%, 33% and 75% of the local models statistically significantly outperform the global model (in terms of the AUC) for Hadoop, Spring, OpenStack, Jupyter and Elasticsearch respectively, while 40%, 12%, 5%, 0% and 0% of the local models have a statistically significantly lower performance compared to the global model. Furthermore, 100%, 78%, 83%, 100% and 100% of Hadoop, Spring, OpenStack, Jupyter and Elasticsearch local models that statistically outperform their respective global model do so with a large difference. Meanwhile, the few global models that outperform their respective local models do so with a negligible difference in 50%, 50%, 100%, 100% and

100% of the cases for Hadoop, Spring, OpenStack, Jupyter and Elasticsearch respectively. Our observation holds for other performance metrics (i.e., Brier Score & AICc).

Since 60% and 35% of the Spring and OpenStack components do not have enough data points (i.e., less than 300 observations through their history) for training a local model, we evaluate cross-component logging level prediction models. In particular, we evaluate whether any of the local models for peer-components with enough data points (aka., *peer-local models*) outperforms the global model when evaluated on data-lacking components. Note that these data-lacking components are excluded from our first three research questions as they do not have enough data to train local models. In particular, we address the following research question:

RQ3. How cross-component models perform compared to the global model?

Data-lacking components have a median of 25% (2 out of 8) and 27% (5 out of 18) of their peer-local models that outperform the global model in terms of AUC for Spring and OpenStack, respectively. Furthermore, 88% and 100% of the Spring and OpenStack data-lacking components have at least one peer-local model that significantly outperforms the global model in terms of the AUC. These results further encourage local modeling as it provides better performance on data-lacking components. However, no specific peer-local model seems to perform the best for all the data-lacking components.

While prior research questions re-evaluate the model of [Li et al. \(2017a\)](#) from a performance perspective, the following research question focuses on the interpretability perspective:

RQ4. How different is the interpretation of global and local models?

The most important features in deciding the appropriate logging level are different between the global and local models. Only one of each of Hadoop, Jupyter and Elasticsearch local models shows an important feature ranking that is strongly correlated with the feature ranking of the global model. In fact, the rankings of the most important features obtained from the global model and 80%, 87%, 83%, 67% and 50% of the Hadoop, Spring, OpenStack, Jupyter and Elasticsearch local models have a very weak to a weak correlation.

Our findings suggest the use of local models for better performance and interpretability. We observe that the evaluation of the prior study is overestimated on the components of multi-component software systems, so one needs to consider local models as well as peer-local models for data-lacking components. However, identifying which peer-component to consider is not straightforward. We suggest future work to investigate approaches that identify the peer-local model to use for data-lacking components. Finally, we observe that using the global model's most important features might mislead practitioners when selecting logging levels for a given component.

The chapter is structured as follows. Section 2 covers the methodology. Section 3 presents our findings. Section 4 discusses the threats to validity. Finally, Section 5 concludes the Chapter.

4.2 Methodology

The goal of this chapter is to investigate the usage of the logging level prediction models in the context of multi-component software systems. In particular, we wish to quantify how the models that are trained on a whole multi-component system (Global model) perform at each component in terms of performance (RQ1) and interpretability (RQ4), as well as explore local models to the components of a multi-component system (RQ2 and RQ3). To do so, we consider two machine learning models to train a global model. We first consider the same ordinal regression model that was evaluated by [Li et al. \(2017a\)](#). That model leverages data from all the components to train a single global model (as shown in [Figure 4.1](#)). We also consider a mixed-effect model, which uses data from all the components, but takes into consideration the particularities that each component has. Additionally, we compare how different is the performance of the local models and the global model (RQ2). The local models are trained on data from one component and evaluated on data from the same component, as shown in [Figure 4.1](#). We evaluated local models for all the components with sufficient data to train a model. To consider a component for training, we used a threshold of 300 observations in order to guarantee 10 observations per feature ([Harrell, 2001](#)). For any other data lacking-component, we also evaluated peer-local models (RQ3) and compared them to the global model. Each peer-local model is trained on one component and tested on a data-lacking component, as shown in [Figure 4.1](#). The same Figure also shows that the training of a model uses a bootstrap sample of data from each component for the training and the rest of the data for testing. We further discuss our modeling

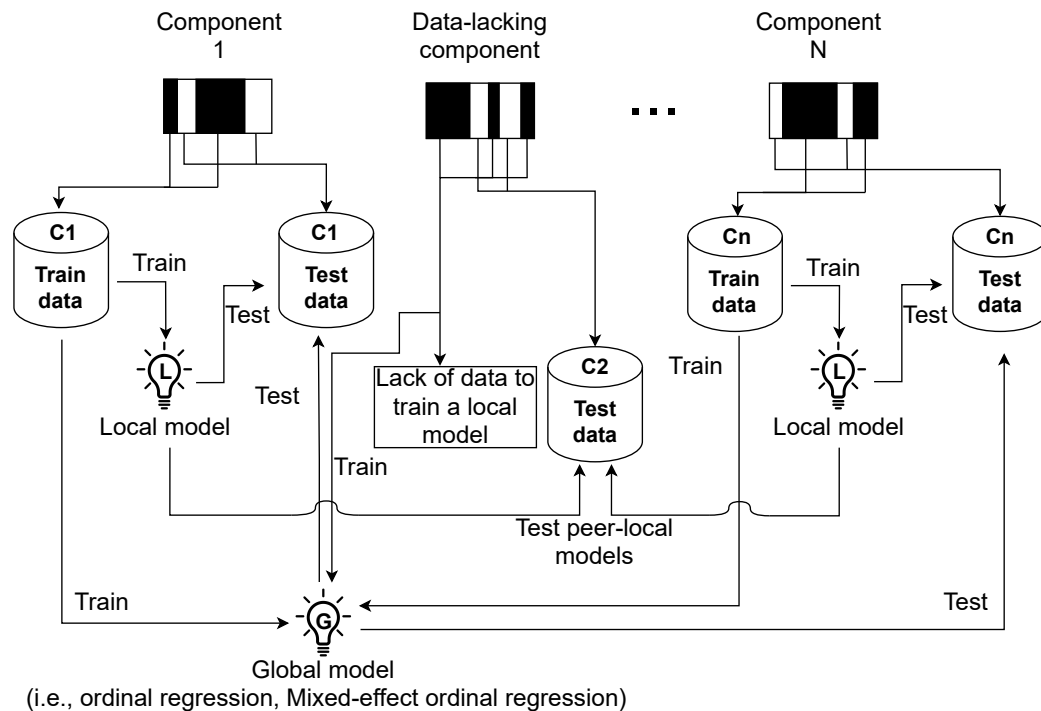


Figure 4.1: An illustration of the models that we evaluated in our study according to data from different components. Note that each of these experiments are repeated 100 times using different bootstrap dataset for the training/testing.

approach in Section 3.1 and the approach of each of our research questions in the results section.

Our study considers five multi-component software systems and their components: Jupyter is web-based interactive computing platform that has 3 components, Elasticsearch is a search engine, Hadoop is a distributed computing system, Spring is a project that offers a large number of services for Java developers, and OpenStack is a cloud computing platform. In this chapter, we define a component boundary by deriving it directly from the official documentation and architectural guidelines of the analyzed projects. Rather than artificially dividing the codebase, we adopt the developers' explicit architectural delineations. For example, within the

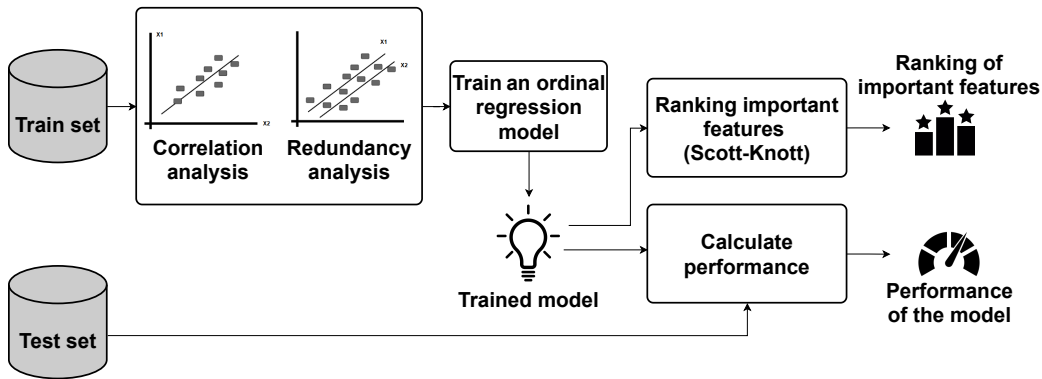


Figure 4.2: An overview of our training and testing methodology

Hadoop system, the official documentation explicitly separates the distributed file system (HDFS) from the processing engine (MapReduce).

4.2.1 Modeling approach

In this chapter, we evaluate ordinal regression models for logging level prediction in the context of multi-component software systems. In particular, we followed the approach summarized in Figure 4.2 to train and test each model, including the global model (i.e., the model that is trained using the whole project data) and the local models (i.e., the model that uses each component’s data separately). The testing set depends on each of our experiments, which are detailed in the approach section of each research question. Our training and testing approach leverages a 100 bootstrap iterations process that leverages 100 bootstrap samples in order to guarantee that our results are statistically robust. Before training each of our models and for each bootstrap sample, we do the following:

4.2.1.1 Correlation Analysis

Before constructing the ordinal regression model, we conduct a correlation analysis on both training datasets (i.e., global and local training datasets) in order to avoid having highly correlated features in the model. Correlated features can interfere when interpreting a model as we cannot separate the individual effect of collinear features. In fact, prior work (Jiarpakdee et al., 2019; Tantithamthavorn and Hassan, 2018) show that correlated metrics can lead to inaccurate ranking of the important features for defect prediction models. In addition, removing these correlated metrics guarantees a consistent ranking of the highly ranked features (Jiarpakdee et al., 2019). To conduct our correlation analysis, we use Spearman rank correlation test as it is more resilient to data that is not normally distributed compared to other correlation tests. Furthermore, we set the features' correlation threshold to 0.7 similarly to prior work (Lee et al., 2020; McIntosh et al., 2014). Finally, for each pair of highly correlated features (i.e., $\rho > 0.7$) we keep only one of the features.

In order to guarantee that the choice of features to keep is consistent throughout all of our experiments, we define a priority list of features. This priority list is important since we compare the interpretability of different models (e.g., global vs local models), so our comparison should exclude correlated features in the same way for any of our models. Therefore, given two correlated features, we keep the one with a higher priority. In addition, our priority list prioritize logging-related metrics over other metrics. For example, when “code churn” and “log churn” features are correlated, we keep “log churn” for the training of the model.

4.2.1.2 Redundancy Analysis

Correlation analysis does not eliminate, but just reduces the collinearity between the independent features. In fact, the correlation analysis detects just the pair of correlated independent features, whereas redundancy analysis identifies which independent features can be predicted by other independent features. To conduct the redundancy analysis, we fit preliminary models, each of which explains one of the independent features using the remaining independent features. We use the R^2 value of a *preliminary model* to measure how well a feature can be explained by the remaining independent features. We remove each independent feature that can be explained by the remaining independent features (i.e when the associated preliminary model has an $R^2 >$ a threshold). For our chapter, we chose the default 0.9 as a cutoff threshold, similarly to prior work ([Lee et al., 2020](#); [Tantithamthavorn et al., 2020](#)).

4.2.1.3 Training and Testing

We train our ordinal regression models using the remaining features that are obtained from the two prior steps. Our training and testing datasets depend on each of our experiments, as discussed in the approach of each research question. For example, to evaluate how the global model performs on a given component (RQ1), we train 100 global models based on 100 bootstrap data samples from all the components. We test each of these models on data from our target component. Note that our testing dataset is not included in our training dataset.

We consider different performance metrics to evaluate the performance of our models. In particular, we consider the AUC³ (Hanley and McNeil, 1982), Brier Score⁴ (Wilks, 2011), and the AICc⁵ (Hurvich and Tsai, 1989) to evaluate the performance and the quality of fit of our trained prediction models. The AUC indicates the discrimination ability of the model, whereas the AICc is a measure that evaluates how the model fits the data. We opt for the corrected AIC (AICc) instead of the regular AIC as the number of observations for local modeling is not very large compared to the number of features for most of the components. The general rule thumb for using the corrected version is having $n < p^2$ (n: number of observations, p: number of features) (Hurvich and Tsai, 2008). Brier Score measures the ability of the model to predict the right class accurately. An AUC higher than 50% is better than a random guess. Brier score ranges between 0 and 2 and the lower it is the better the model is. A random guess logging level predictor model achieves a Brier Score of 0.80. Finally, the lower the AICc is, the better the model fits the data.

To guarantee robustness of our findings, our training and testing approach is based on the out-of-sample bootstrap validation technique, similarly to previous work (Lee et al., 2020; Thongtanunam and Hassan, 2018; Tantithamthavorn et al., 2019a). This technique starts by generating a bootstrap sample of size N with replacement. While the bootstrap sample is used for training the model, we evaluate the performance of the trained model using the sample of observations that do not

³Area under the ROC curve: the metric summarizes a classifier performance over all possible thresholds (i.e, cutoff probability to select a class).

⁴Brier Score is a loss function that measures the accuracy of probabilistic predictions. This score quantifies the ability of the model to predict the right class accurately.

⁵Corrected Akaike Information Criterion measures the fit of a statistical prediction model on a given dataset.

appear in the bootstrap sample. We repeat the process of training and testing for 100 generated bootstrap samples.

4.2.1.4 Feature Importance

We used Wald's χ^2 to evaluate the impact of each metric on predicting the logging level similarly to prior work (Lee et al., 2020; Thongtanunam and Hassan, 2018). The larger the χ^2 is for a feature, the larger the explanatory power of that feature is. Since we train 100 models by leveraging 100 bootstrap samples, we obtain 100 different rankings of the most important features. To obtain a final ranking, we used the Scott-Knott (Tantithamthavorn, 2018) clustering technique similarly to previous studies (Rajbahadur et al., 2019; Jiarpakdee et al., 2019). This technique uses an iterative process to generate a ranking of groups of features according to their importance.

4.3 Results

In this section, we present the results for each of our RQs. For each RQ, we discuss the motivation, the approach we used to address the RQ, and our findings.

4.3.1 RQ1. How global models perform at the component level?

Motivation: The goal of this research question is to quantify the performance of the global model (the model that is trained on the whole multi-component system) on each component. While it is expected that the model behave differently from one to another component, this research question quantifies such differences so that

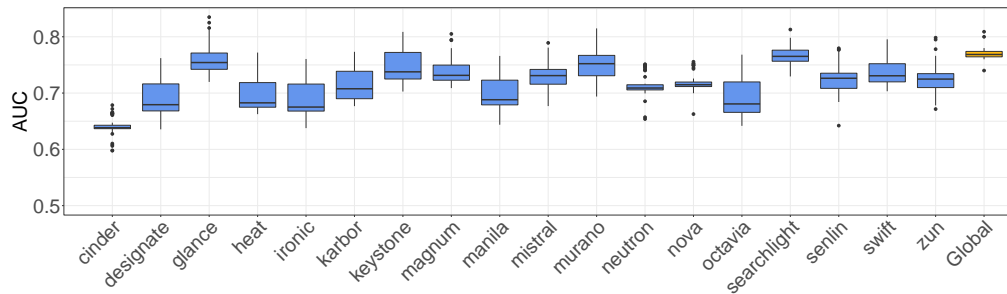


Figure 4.3: The AUC performance of the global model on components and on the OpenStack project

we can understand to which extent practitioners can leverage the global model for their components.

Approach: To evaluate how the global model performs on each component of a multi-component system compared to the same model on the whole multi-component system, as evaluated by [Li et al. \(2017a\)](#), we train the global model using a bootstrap sample of observations from all the components. We then test that model on two separate datasets:

- **global testing:** using the remaining observations that belong to the whole project out of the training bootstrap sample.
- **local testing:** using a component's observations that are not used for training the global model.

This process is done for 100 times as explained in Section 4.2.1 and for every component in order to obtain the performance of the global model on the whole project and on each of the components.

Results: The performance of the global model on each component is lower than the performance of the same model when tested on the dataset that is obtained from all the components, as shown in Figure 4.3 for OpenStack and

Figure A.1 in Appendix A for our other studied projects. That indicates that the performance obtained in the prior study (Li et al., 2017a) is overestimated for the components of a multi-component software system. The AUC performance of the global model on 80%, 75%, 72%, 67% and 75% of the components of Hadoop, Spring, OpenStack, Jupyter and Elasticsearch is statistically significantly lower than the AUC performance of the same model when tested on data from the whole project. In fact, while the Hadoop, Spring, OpenStack, Jupyter and Elasticsearch global model shows a median AUC of 75%, 76%, 77%, 81% and 76% respectively, that global model shows a median AUC of 71.5%, 73.5%, 75%, 74% and 72% when evaluated at each component level. Such an AUC can be as low as 69%, 67%, 64%, 65% and 64%. Similarly, the global model has a lower Brier Score when tested on all of OpenStack, Spring and Jupyter components, 60% of Hadoop's components and 75% of Elasticsearch components, as shown in Figure A.2 in Appendix A. The global model shows a Brier Score performance up to 0.41, 0.78, 0.66, 0.6 and 0.74 on Hadoop, Spring, OpenStack, Jupyter and Elasticsearch components respectively.

Such performance variation can be explained by the statistically significant differences (Chi-square; $\alpha = 0.01$) between the distribution of the dependent feature (i.e., logging level) in each component compared to the whole multi-component system. We observe such a statistically significant distribution difference for 3 out of 5, 5 out of 8, 11 out of 18, 4 out of 4 and 3 out of 3 of Hadoop, Spring, OpenStack, Jupyter and Elasticsearch respective components.

Finally, the amount of changes to the logging statements –as few as they are– is not correlated with the performance of the global model. While we cannot systematically identify whether traces generated within components exhibit a buggy

behavior (e.g., difficult to read), we find that the median percentage of the logging statements for which the logging level (dependent variable) was changed is 7.5%, 3.8%, 3.9%, 3% and 4.7% for Hadoop, Spring, OpenStack, Jupyter and Elasticsearch respectively. We also observe that the median percentage of the logging statements for which at least one feature was changed is 9.5%, 6.6%, 7%, 8.3% and 8.8% for Hadoop, Spring, OpenStack, Jupyter and Elasticsearch respectively. Moreover, we do not observe any correlation (Spearman, $\rho = -0.04$) between the performance of the global model on the components and the number of the logging changes that the components exhibit.

Summary of RQ1

The global model shows a lower AUC performance when evaluated on 80%, 75%, 72%, 67% and 75% of the components compared to the same model when evaluated globally (on testing data from the whole system) for Hadoop, Spring, OpenStack, Jupyter and Elasticsearch. **Our results caution about the usage of a global model for local components.**

RQ2. How local models perform on the component level compared to the global model?

Motivation: The goal of this research question is to investigate whether one should use a local model (i.e., a model trained for each component) instead of global models (i.e., a model trained on the whole components of a multi-component system). To do so, we quantify the performance differences between global and local models. We also evaluate a global model that takes into consideration the particularities of

each component of a multi-component system. In particular, we evaluated on top of local models and global models a mixed-effect model; this model uses in addition to the features used by the global models (i.e., fixed effects), the information about the component to which a logging statement belongs as a random effect.

Approach: In this RQ we compare the performance of the local models (i.e., trained on components), the global model (i.e., trained on data from the entire project), and the mixed-effect model. To do so, we train and test our models following the approach explained in Section 4.2.1. For each component, we train our three models on a bootstrap sample and test these models on the same component's data. Our testing data is not included in any of the bootstrap samples that are used to train our models. We repeat this process 100 times for each component in order to obtain a 100 measure distribution for each performance metric (i.e., AUC, AICc, and Brier Scores) and for each model.

Results: Local models outperform the global model and the mixed-effect model in terms of performance metrics (i.e., AUC and Brier Score) and quality of fit (i.e., AICc). 60%, 75%, 77%, 33% and 75% of the Hadoop, Spring, OpenStack, Jupyter and Elasticsearch local models statistically (Wilcoxon test, $\alpha = 0.01$) outperform their respective global model in terms of AUC, as shown in Figure 4.4 for OpenStack and Figure A.3 in Appendix A for the rest of our studied projects. Such differences have a large effect size (Cohen's d ; $d > 0.7$) for 100%, 78%, 83%, 100% and 100% of the components of Hadoop, Spring, OpenStack, Jupyter and Elasticsearch respectively. Furthermore, none of Elasticsearch and Jupyter local models show a statistically significantly lower AUC performance compared to the global model, and only 20% (1 out of 5), 12% (1 out of 8) and 5% (1 out of 18) of the

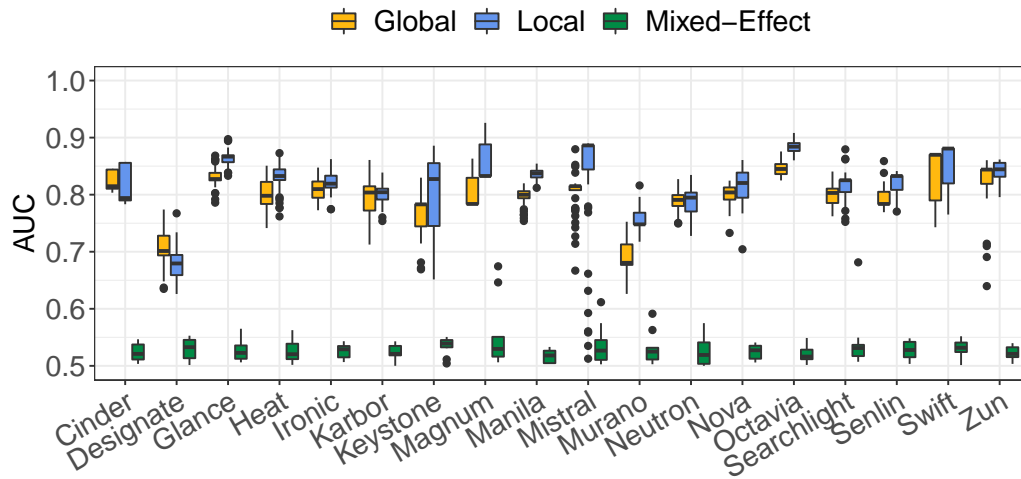


Figure 4.4: AUC scores of the global, local and mixed-effect models on the OpenStack components

Hadoop, Spring and OpenStack local models show a statistically significantly lower AUC performances compared to the global model.

Similarly, we observe that local models show better Brier Score and AICc compared to the global model for all the components, as shown in Figures 4.5 and 4.6 for OpenStack and Figures A.4 and A.5 in Appendix A for the other studied projects. We observe that all the local models (except one from Elasticsearch and one from Jupyter) provide a significantly large Brier Score improvement (Cohen’s d ; $d > 0.7$) compared to their respective global model for Hadoop, Spring, OpenStack, Jupyter and Elasticsearch. On top of that, we observe that the global model of the Spring project reaches a median Brier Score that is worse than a random guess ($BS \geq 0.8$) on three Spring components.

Although the mixed-effect models are supposed to consider the particularities of each component while using data from all the components, we observe that such

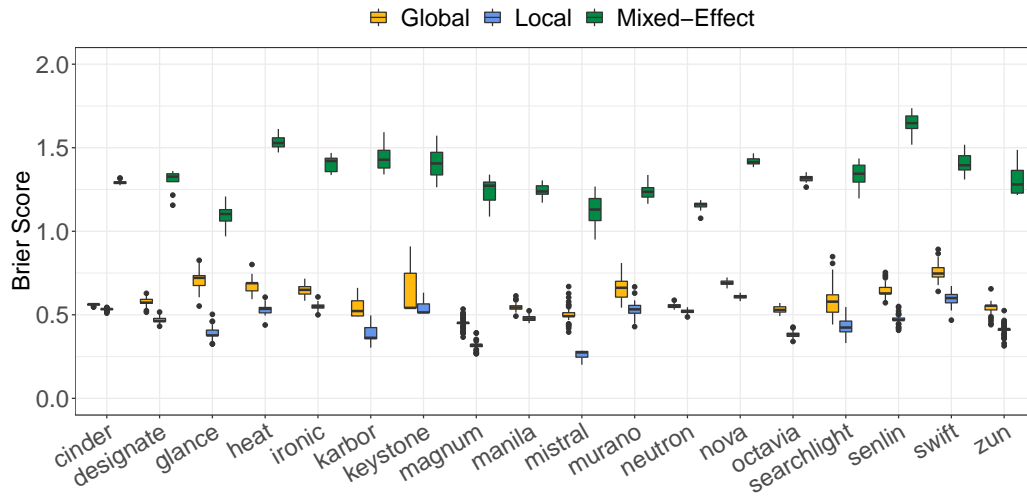


Figure 4.5: Brier Score of the global, local and mixed-effect models on the OpenStack project

models do not perform well for any of our performance metrics, as shown in Figures 4.4, 4.5, and 4.6 for OpenStack and Figures A.3, A.4, and A.5 in Appendix A for the other studied projects. In fact, the mixed-effect models have an AUC lower than 60% when tested on 87% of Spring’s components and all the other projects’ components. Similarly, the Brier Score of our evaluated mixed-effect models exceeds the Brier Score of random guess (i.e., BS=0.8) on all the components of our studied project. Finally, we find that the mixed-effect models also provide the worst fit to the training dataset.

We observe certain logging practices that are common within the eight different groups of OpenStack components that are for the same functionality, while the same logging practices are different from one group of components to another. For instance, the components of 6 out of the 8 groups do not have a statistically significantly different (Wilcoxon test, $\alpha = 0.01$) length of the logging statements, while a median of 90% of the pair of components that belong to different groups have

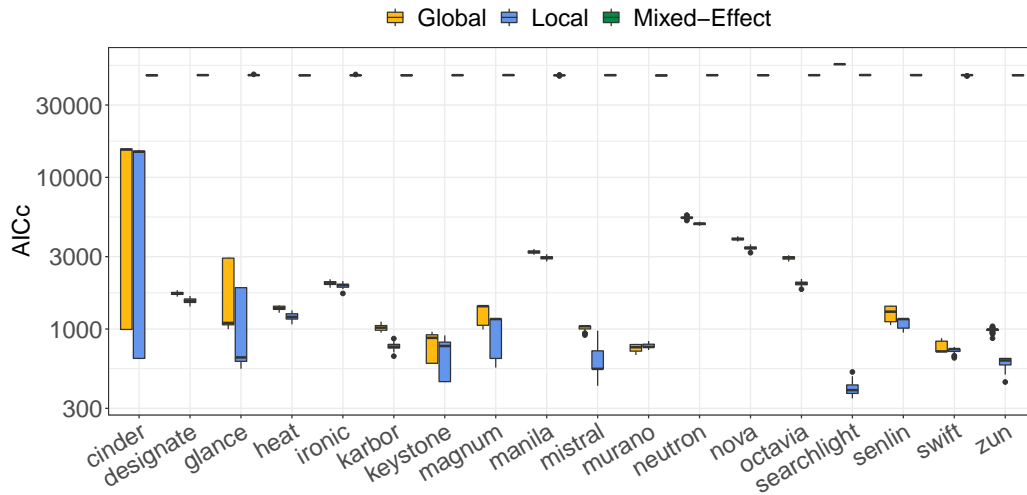


Figure 4.6: Corrected AIC of the global, local and mixed-effect models on the OpenStack project

statistically significantly different lengths of the log messages. For example, the components that belong to the hardware management of OpenStack tend to have longer (double) log messages compared to the orchestration components. Such a difference is because hardware management components (68% of the times) need to mention the node (i.e., the hardware) in use, its current and expected state in the logging statement. Similarly, the components within each of the 8 groups leverage similar logging levels, which is not the case for components that belong to different groups. For example, the workload provisioning components have between 61% to 90% of the logging statements with the verbose logging levels (i.e., Info and Debug), while the same percentage is between 47% and 59% for the storage (e.g., Manilla) components. In fact, we find that one of the workload components had an audit of logging statements, where the maintainers “changed many [logging levels]

from LOG.Info to LOG.Debug”⁶ in order “to conform with logging standards”⁷. These within group similarities and across group differences have an impact on the logging level prediction models. For instance, the differences between the AUC performance of the local models for within-group components have a low to medium effect size except for the differences between a median of 25% of the within-group components. Additionally, we observe a large effect size when comparing the AUC performance of a median of 91% of local model pairs for components that belong to different groups.

We believe that one other reason for the difference of logging practices between components might have to do with how prone they are to bugs. In fact, [Shang and Hassan \(2015\)](#) indicate that the bugginess of a source code has a relation with the logging activity in the same code, as files with excessive logging typically reflect developers’ concerns about defects. For instance, we observe that components such as Nova for OpenStack is more complex (i.e., double the cyclomatic complexity) than other components such as Ironic (designed for hardware lifecycle management), hence it is likely that Nova is more prone to bugs according to previous research on the relation between complexity and bugginess ([D’Ambros et al., 2012](#); [Shihab et al., 2010](#)). Consequently, according to [Shang and Hassan \(2015\)](#) it is likely that Nova logs differently than Ironic. Similarly, we observe that the Spring AMQP component has six times more revisions than Spring Framework, which indicates that Spring AMQP is more susceptible to bugs ([D’Ambros et al., 2012](#); [Graves et al.,](#)

⁶<https://github.com/openstack/trove/commit/cf7694f0dbf9b5f81057f00c35cfe626f22a71ec>

⁷<https://github.com/openstack/trove/commit/cf7694f0dbf9b5f81057f00c35cfe626f22a71ec>

2000; Hassan, 2009), according to prior work on the relation between revisions and bugginess. Therefore, logging between these two components might be different.

Summary of RQ2

Local models outperform the global model on 60%, 75%, 77%, 100% and 75% of the components in terms of all the performance metrics (i.e., AUC, Brier score and AICc). We do not observe any global model that outperforms the local model on more than one performance metric. **Our results suggest the use of local models instead of a global model.**

RQ3. How cross-component models perform compared to the global model?

Motivation: The goal of this research question is to investigate which models might better fit components with insufficient data points for training a model, which we refer to as data-lacking components. For instance, 60% and 35% of Spring and Openstack respective components have less than 300 logging statements. Leveraging these components' data can lead to overfitting and consequently a poor predictive ability on future logging statements within those lacking component. Therefore, we evaluate the performance of the global model on such data-lacking components, and we compare the global model to local models that are trained on peer-components, which we refer to as peer-local models.

Approach: To evaluate which models better fit data-lacking components, we train a global model similarly to the previous research questions, a global model that takes into consideration the particularities of each component (i.e., a mixed-effect

model), and local models for components with enough data points. Similarly to our previous research questions, we leverage 100 bootstrap samples for each of the three types of models. For instance, we evaluate 100 global models that leverages 100 bootstrap samples on a targeted data-lacking component. Similarly, we evaluate 100 mixed-effect models. Finally, for each peer-component, we train 100 models based on 100 bootstrap samples from that peer-component. Thus, by obtaining a distribution for each performance metric and model, we compare these distributions using Wilcoxon test. We quantify the differences by leveraging *Cohen's* effect size.

Results: At least one peer-local model statistically significantly outperforms the global model on 88% of Spring and all OpenStack respective data-lacking components, as shown in Figures 4.7 and 4.8 for one of OpenStack (Aodh) data-lacking components. The other data-lacking components figures can be found in Appendix A. A median of 2 and 5 peer-local models outperform the global model in terms of AUC for Spring and OpenStack, respectively. Similarly, a median of 4 and 5 peer-local models outperform the global model in terms of Brier Score for Spring and OpenStack, respectively. Note that Spring and OpenStack have 8 and 18 components with enough data points, which we used to train peer-local models.

We also observe that 6 out of 9 and 10 out of 10 Spring and OpenStack data-lacking components have at least one peer-local model that outperforms the global model in terms of AUC with a large effect size (i.e. $d > 0.7$). On the other side, none of the Spring and OpenStack global models statistically outperform all the peer-local models on data-lacking components. Finally, peer-local models have a

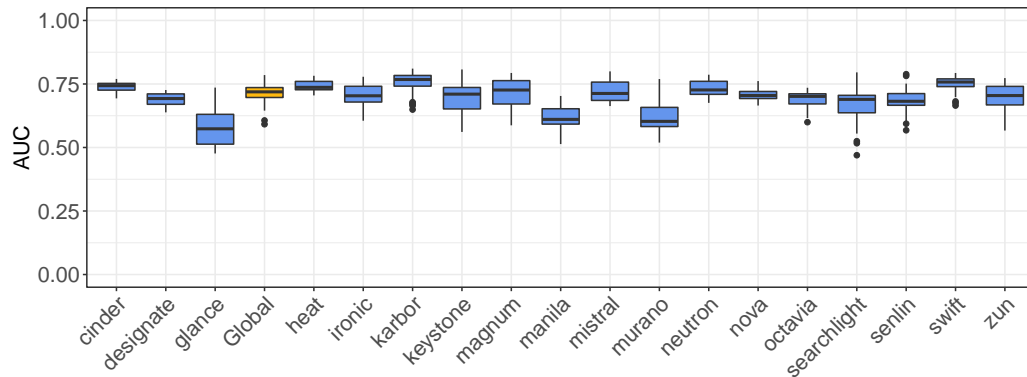


Figure 4.7: The AUC performance of the global and peer-local models on the Aodh component

maximum AUC of 0.99 and 0.88 and a minimum Brier Score of 0.33 and 0.24 for Spring and OpenStack.

While the global model can leverage knowledge from data lacking components, that knowledge is both limited (due to the lack of data) and not impactful (due to bias caused by other components that are different from the data-lacking component). In fact, a data-lacking component would share more common characteristics with a similar component compared to a dataset containing all components with their significant differences.

Our evaluation of the mixed-effect model tested on the data-lacking component shows a poor performance in terms of both the AUC and Brier Score. In fact, 78% and 100% of the mixed-effect models reach a median AUC lower than 60% for Spring and OpenStack, respectively. Similarly, all the mixed effect models of Spring and OpenStack exceed the Brier Score of 0.8 (i.e., equivalent to a random model). Our findings further emphasize the challenges of modeling logging decisions within multi-component software systems, especially for data-lacking components.

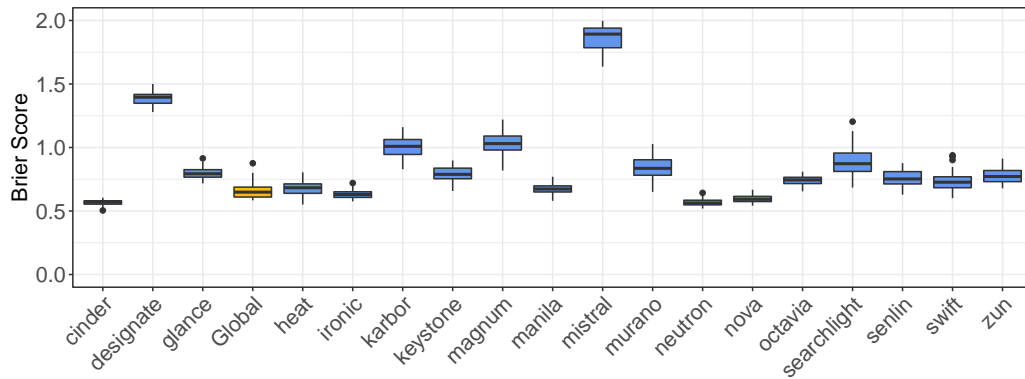


Figure 4.8: The Brier Score performance of the global and peer-local models on the Aodh component

We do not observe any specific peer-local model that consistently performs the best on all data-lacking components of a multi-component software system. For instance, 5 and 4 different peer-local models performed the best for all the Spring and OpenStack data-lacking components. While certain peer-local models show a good performance on one data-lacking component, they show a low performance on other data-lacking components. For example, the peer-local model that is trained on the *Glance* component shows a median AUC of 0.81 on the *Zaqar* data-lacking component, while the same model shows a low AUC of 0.5 on the *Qinling* data-lacking component. For each data-lacking component, we ranked the peer-local models from the most to the least performant ones. Then, we statistically compare the ranks between each pair of data-lacking components. Our statistical comparison (Spearman Correlation) of the ranks between the data-lacking components shows that 31% and 77% of the correlations are weak or very weak across all data-lacking components for Spring and OpenStack respectively, as shown in Figures A.42 and A.43 in Appendix A. Meanwhile, only 28% and 8% of the rankings

are strongly correlated across all data-lacking components of Spring and OpenStack respectively.

While we report on the potential that peer-local models have for logging level prediction on data lacking components (e.g., early-development components), identifying the suitable peer-local model might incur additional efforts especially for systems with a large number of components. These additional efforts only include training and testing the peer-local models, which can be fully automated. Additionally, we encourage future work to build approaches to identify the best peer-local model to use for a given data lacking component.

Summary of RQ3

At least one peer-local model outperforms the global model on 88% and 100% of the components of Spring and OpenStack. However, there is no unique peer-local model that fits all data-lacking components. **Our results suggest the use of peer-local models for data lacking components (such as early-development components, with limited to no historical data).**

RQ4. How different is the interpretation of global and local models?

Motivation: The goal of this research question is to quantify to which extent the interpretability of a global model is similar to the interpretability of a local model. While prior study (Li et al., 2017a) investigated the interpretability (i.e., the most important features) for the models that predict the logging level, their investigation

does not consider the particularities that a component might have. The interpretation of their model can mislead developers in the context of a multi-component system since the developers of each component might follow different logging strategies. Therefore, this research question quantifies the extent to which a developer might be misled when interpreting a global model compared to a local model.

Approach: To quantify the differences between the interpretability of the global model and the local models, we calculate the ranking of the most important features for the global model and the local models following the approach discussed in Section 4.2.1.4. In particular, we calculate a ranking of the most important features from each of the 100 bootstrap iterations to obtain 100 rankings. For instance, we obtain 100 rankings for the global model and 100 rankings for each of our local models. We classify the 100 rankings for each model by using the Scott-Knott clustering technique, to obtain a final ranking of the most important features for each model. We then compare the final rankings of each local model with the ranking for the global model using Spearman rank-correlation test.

We also compare the global and local models based on the positive or negative impact that each feature might have on predicting the logging level. Given a global model GM and a local model LM , which both share an important feature F , we measure whether F has a similar impact, either positive or negative, on both GM and LM . To do so, we leverage the following steps for each feature F and logging level LL (e.g., predicting the debug level):

- We predict our targeted LL by setting all the feature at their median values.
- We increase our targeted feature F by one standard deviation above its median.

- We re-predict the same targeted LL .
- We compare whether the predicted probability increased or decreased when adding one standard deviation to the feature F for both GM and LM models.
- We repeat the same experiments for all the remaining features, local models, and predicted logging levels.

Results: We observe a low similarity between local models and the global model feature rankings. The ranking of the most important features of 80%, 87%, 83%, 67% and 50% of Hadoop, Spring, OpenStack, Jupyter and Elasticsearch local models have a very weak to a weak rank correlation with the ranking of important features of the global model. Furthermore, the ranking of the most important features of only 20% (1 out of 5), 0%, 5% (1 out of 18), 0% and 0% of Hadoop, Spring, OpenStack, Jupyter and Elasticsearch local models have a strong to very strong rank correlation with the global model's important features rankings. We observe that only 42%, 13%, 8%, 25% and 37% of the global model's important features are also important for all the local models of Hadoop, Spring, OpenStack, Jupyter and Elasticsearch, as shown in Tables [A.1](#), [A.2](#), [A.5](#), [A.3](#) and [A.4](#) in Appendix [A](#). As shown in the same tables, we also observe that the Hadoop, Spring, OpenStack, Jupyter and Elasticsearch global models do not consider an average (across all components) of 6.3%, 2%, 17%, 30% and 40% of the features that are important for at least one local model.

We can explain the few occasions where we have high correlation between the important features rankings of the global model and the local models by the fact that sometimes only the generic features (i.e., those that are relevant for the global

and more than 50% of the local models) are enough to model the logging level choice within a component and no component-specific features are needed. For example, the Common component model for which we had a strong rank correlation with the global model only uses 8 features (excluding Tokens), 87% of these features are generic as they are shared with more than 50% of the other local models and the global model. Meanwhile, the other Hadoop components' models (i.e., the ones with weak important features ranking correlation) use an average of 11 features for logging level prediction, with an average of only 54% of generic features. Similarly, Spring's 7 local models for which we found a weak to very weak important rank correlation use an average of 14 features, among which only an average of 22% are generic features. We observe that the only OpenStack local model with a strong to very strong rank correlation (Mistral) uses only 6 features among which 67% are generic features. Finally, OpenStack's local models with weak rank correlation use an average of 9 features with an average of only 40% of the generic features.

The common most important features between the global model and the local models can have contradictory effects on the prediction of the appropriate logging level for a logging statement. We observe a median of 38%, 83%, 84%, 12% and 12% features that exhibit a contradictory effect on the choice of at least one logging level between the global model of Hadoop, Spring, OpenStack, Jupyter and Elasticsearch and each of their respective local models. For example, a larger length of a log message increases the probability of a logging statement to have the *Info* logging level according to the global model of Hadoop, while that is the opposite for the local component of the *Common*'s component of Hadoop (i.e.,

the shorter a log message is, the more likely that logging statement's level is *Info*). Similarly, we observe that a logging statement with more variables is less likely to have the *Warn* logging level according to the global model of Spring project. Meanwhile, increasing the number of variables in a logging statement leads to increasing the probability of the *Warn* logging level when we use the *framework* component local model. Finally, we find that a larger logging statement containing bloc (i.e., more lines of code in the containing bloc) decreases the probability of a logging statement to have *Debug* logging level according to the global model of OpenStack, while that is the opposite for the local model of the *Nova* component (i.e., the larger the containing bloc is, the more likely that the logging statement's level is *Debug*).

We recommend users to train a local logging level prediction model for each component. While the performance of the global model that leverages data from all components is not as low as a random guess (AUC=0.5), interpretability of the local models can be different from that of the global model, which leads into erroneous interpretation of what factors impact logging level prediction within individual components.

Summary of RQ4

The interpretability of the global model can be misleading when leveraged for a local component. A maximum of just one local model per case study shows an important features ranking that is strongly correlated with the important features ranking of its respective global model. **Our results suggest the use of local models for better interpretability.**

4.4 Threats to Validity

In this section, we discuss the threats to the validity of our study.

4.4.1 External Validity

An external threat to the validity of our findings concerns the generalizability of our results. While we do not generalize our results to other multi-component software systems, our study covers five large and popular multi-component software systems. Even if the impact of the differences between the components of a given multi-component system can happen to be negligible, one has to study such an impact before using a logging level prediction model. Nonetheless, we encourage future work to replicate our work on other systems, as that might prove to be promising.

We do not generalize our results on other machine learning models. While our study focuses on predicting the logging level for logging statements, our study is a first investigation of the performance and interpretation of machine learning models in the context of multi-component software systems. We encourage future work to replicate our study on other machine learning models, e.g., for bug prediction.

4.4.2 Internal Validity

One internal threat to the validity of our results concern boundaries between components. In fact, we set the boundaries between components of a software project based on the major sub-projects within that project. Yet, a sub-project might be internally split into other components. Future work might instead re-evaluate our

findings on different component boundaries, such as files that are maintained by a specific group of developers.

Another internal threat to validity concerns the impact of the amount of modifications to the logging statements on the performance of our evaluated models. While inaccurate logging decisions (e.g., wrong logging level choice) can influence the quality of the training datasets, we observe that logging statements are not frequently changed in our studied projects. Additionally, we do not observe any correlation between the amount of logging changes and the performance of our evaluated logging level prediction models.

4.5 Chapter Summary

Since the identification of the appropriate logging level for a new logging statement is challenging, prior studies leveraged machine learning models (e.g., ordinal regression models) to predict the appropriate logging level for a new logging statement. However, these prior studies do not consider the characteristics of multi-component software systems. In such systems, each component can be developed by a different team that follows different logging strategies. That can have an impact on how prior studies' models perform on each component.

In this chapter, we quantify the impact of the variation between different components on the performance and interpretability of logging level prediction models. We observe that the global models show a statistically significantly lower performance when evaluated on each component compared to the same models when evaluated on the whole multi-component system. Leveraging local models that are dedicated to a component shows a better performance compared to the a global

model. The interpretability of the global model is different from the interpretability of the local model, which might mislead developers.

While we observe that leveraging a local model is better than a global model, 60% and 35% of the Spring and OpenStack components do not have enough data points to train a local model (aka., data-lacking components). Therefore, we evaluated a cross-component modeling approach, which consists of leveraging peer-local models for the data-lacking components (e.g., early-development components). Our evaluation shows that peer-local models outperform the global model, while there is no unique peer-local model that fits any data-lacking component. Despite the fully automated process for training multiple peer-local models, we encourage future work to propose approaches that identify the best suitable peer-local model.

Finally, we caution about the usage of a global model as it might not perform well in terms of performance and interpretability when evaluated on a component of a multi-component system. Instead, we suggest leveraging the appropriate local model for each component and peer-local models for data-lacking components.

CHAPTER 5

The Impact of Concept Drift and Data Leakage on Logging Level Prediction Models

This chapter is published in the ACM Empirical Software Engineering journal (EMSE) ([Ouatiti et al., 2024](#)).

DEVELOPERS insert logging statements to collect information about the execution of their systems. Along with a logging framework (e.g., Log4j), practitioners can decide which logging statement to print or suppress by tagging each log line with a logging level. Since picking the right logging level for a new logging statement is not straightforward, machine learning models for logging level prediction (LLP) were proposed by prior studies. While these models show good performances, they are still subject to the context in which they are applied, specifically to the way practitioners decide on logging levels in different

phases of the development history of their projects (e.g., debugging vs. testing). For example, Openstack developers interchangeably increased/decreased the verbosity of their logs across the history of the project in response to code changes (e.g., before vs after fixing a new bug). Thus, the manifestation of these changing log verbosity choices across time can lead to concept drift and data leakage issues, which we wish to quantify in this chapter on LLP models. In this chapter, we empirically quantify the impact of data leakage and concept drift on the performance and interpretability of LLP models in three large open-source systems. Additionally, we compare the performance and interpretability of several time-aware approaches to tackle time-related issues. We observe that both shallow and deep-learning-based models suffer from both time-related issues. We also observe that training a model on just a window of the historical data (i.e., contextual model) outperforms models that are trained on the whole historical data (i.e., all-knowing model) in the case of our shallow LLP model. Finally, we observe that contextual models exhibit a different (even contradictory) model interpretability, with a (very) weak correlation between the ranking of important features of the pairs of contextual models we compared. Our findings suggest that data leakage and concept drift should be taken into consideration for LLP models. We also invite practitioners to include the size of the historical window as an additional hyperparameter to tune a suitable contextual model instead of leveraging all-knowing models.

5.1 Introduction

The practice of inserting logging statements is an important component of the development activity as it provides insights about the execution of software systems (Li

et al., 2021a; Shang and Hassan, 2015; Yuan et al., 2010; Lin et al., 2018; Yuan et al., 2011), so practitioners (e.g., developers, release managers, operators) can prevent and easily fix errors. Each logging statement requires in addition to a message, a logging level (e.g., trace, info, warn, error, fatal) that decides the verbosity of that logging statement. Practitioners can then adjust the logging levels via a logging framework, to decide which logging statements to trace and which ones to ignore. Such a logging level adjustment suppresses unnecessary logging statements that might cause noise and overhead to the system. Inversely, the logging level can be set to show more verbose logging statements which provides further information during debugging tasks.

Given that choosing the suitable logging level for logging statements is not a straightforward task (Oliner et al., 2012; Yuan et al., 2010) and that developers typically make initial poor logging level choices (Li et al., 2021a; Oliner et al., 2012), prior studies (Li et al., 2017a, 2021b) suggested machine learning models that leverage different metrics to predict the right logging level for a logging statement. Such metrics quantify properties about the logging statement (e.g., length of the logging statement), the containing block (e.g., type of block), the containing file (e.g., number of logging statements), the change in the file (e.g., code churn) and the history of the file (e.g., number of revisions). Such models achieve an average AUC performance ranging from 0.75 to 0.81 (Li et al., 2017a) and from 0.79 to 0.85 (Li et al., 2021b) across the evaluated projects.

However, neither of these studies (Li et al., 2017a, 2021b) takes into account that logging practices can change over time for a variety of reasons not captured by

earlier metrics, which can impact the performance and interpretation of these models. In fact, logging strategies are unstable (Kabinna et al., 2016) and can change depending on the development phase and the performance of the system. Such changes in the logging practices reflect a global change in the logging strategy from one period of time to another and are not limited to the update of the logging level for certain logging statements. For instance, during debugging activities OpenStack developers typically opt to increase the verbosity of their logs, in order to resolve bugs ¹. Inversely, at another period of the history of the OpenStack project, developers might decide to reduce the verbosity as their logs are getting noisy without any abnormal system activity ². Additionally, performance monitoring activities can also motivate log verbosity tweaks as excessive logging from a previous time period can negatively impact the performance of systems (Yuan et al., 2014).

The impact of changing logging practices can lead to two well-known time-issues for LLPs. On the one hand, concept drift refers to the phenomenon according to which the statistical properties of a variable (dependent or independent) unexpectedly change over time (Gama et al., 2014; G.Ditzler et al., 2015). This can occur as the logging practices on which LLP models were trained become obsolete with time (e.g., after the adoption of a new logging strategy). On the other hand, this drift of continuously changing logging levels also leads to a higher impact of so-called data leakage on model performance. Data leakage refers to situations where the usage of random train/test splits biases the model by giving it access to future information (Kaufman et al., 2011). This can occur if future logging practices (e.g., logging data when debugging an issue in March) were used to predict past logging

¹<https://bugs.launchpad.net/nova/+bug/1715785>

²<https://github.com/openstack/swift/pull/15>

level decisions (e.g., when debugging an issue in January). While a known issue for other software analytics approaches, the restless nature of logging levels can be expected to only exacerbate the impact of data leakage.

While prior studies measured the impact of concept drift and/or data leakage on bug prediction models (Ekanayake et al., 2009; Bennin et al., 2020) and AIOps models (Lyu et al., 2021a), no prior studies focused on the impact of these time-related issues on LLP models. In fact, given the context-dependent nature of machine learning (Agrawal and Menzies, 2019; Chahar and Kaur, 2020), the impact of time-related issues (i.e., concept drift and data leakage) cannot be directly deducted from the impact on other software engineering machine learning models. For instance, the domain of logging is different than previously studied domains from one side, and the type of data and model are different between logging level prediction and the existing studies from another side. These last differences are in terms of the type of data used (stream vs. batch data, in comparison with AIOps data) and the type of predicted outcomes (binary vs. ordinal predicted response, in comparison with defect prediction data). Lastly, while LLP models predict decisions controlled by developers (i.e., the developers choose the logging level), prior studied models (e.g., defect prediction) are out of the developer’s control and depend mainly on the system characteristics (e.g., bug proneness).

Due to these two differences (i.e., the domain and the type of data), different optimization and fine-tuning routes (e.g., different hyperparameters) might be followed to achieve the best-performing model for each domain, as suggested by Agrawal and Menzies (2019). These problem-specific optimizations –combined

with the domain and data differences– make software engineering models vary significantly (Zhang et al., 2016). For example, Bennin et al. (2020) reported that –even within the context of defect prediction– some models are more robust to time-related issues than others.

Thus, our work aims to study the impact of the phenomena of data leakage and concept drift on logging level prediction models, which are not explored yet, and involve substantially different types of data compared to data on which the two time issues were evaluated in the past (i.e., bug prediction and AIOps). On the one hand, LLPs might not be as impacted as models based on stream data (e.g., AIOps models), as logging strategy changes can be infrequent in time (e.g., projects with long release cycle). On the other hand, one could assume that these logging level prediction models might be severely impacted by time-related issues in case of abrupt changes in the logging practice.

Specifically, we evaluate both a state-of-the-art shallow logging level predictor (aka. Shallow-LLP) proposed by Li et al. (2017a) and a state-of-the-art deep logging level predictor (aka. DL-LLP) proposed by Li et al. (2021b). Note that we evaluate both models as they can have different advantages over each other. In particular, the DL-LLP can be used for its high prediction performances, while the Shallow-LLP as it can be also used for prediction it is more importantly used for the interpretation and explanation of the important factors related to the selection of logging levels. Furthermore, we evaluate different time-aware modeling techniques to deal with the data leakage and concept drift observed in logging level prediction. To do so, we focus on the following research questions:

RQ1: What is the impact of data leakage on the performance of logging level prediction models?

We observe that randomly splitting the training and testing data (i.e., random splitting approach) overestimates the AUC performance of the Shallow-LLP by a median³ of 7% (Hadoop), 3% (Spring) and 3% (OpenStack). Similarly, the DL-LLP trained using a random-approach overestimates the AUC performance by a median³ of 9%, 2% and 3.5% for Hadoop, Spring and OpenStack respectively. Such an overestimation can lead up to a maximum of 17% (Shallow-LLP) and 27% (DL-LLP). We also observe that models based on random splitting statistically significantly outperform the models that are based on a time-aware splitting approach in a median³ of 90% (Hadoop), 75% (Spring) and 71% (OpenStack) of the testing time frames for the Shallow-LLP, and in a median of 57% (Hadoop), 67% (Spring) and 100% (OpenStack) of the testing time frames for the DL-LLP.

RQ2: What is the impact of concept drift on the performance of logging level prediction models? Shallow-LLP and DL-LLP see statistically significant performance drops occur as early as a median³ of 1.5, 1 and 1 time frames (two months length) after the end of the training period for Hadoop, Spring and OpenStack respectively. The magnitude of this AUC performance decrease is estimated at a median⁴ of 3.6% (Hadoop), 5.3% (Spring), and 2.8% (OpenStack) for the Shallow-LLP and at a median⁴ of 4.5% (Hadoop), 8.2% (Spring) and 3.2% (OpenStack) for the DL-LLP. Across all the testing time frames of a trained model, we observe that the median⁴ AUC performance decreases are

³Median over the time frame sizes from 4 to 24 months.

⁴Median over the time frame sizes from 4 to 24 months.

8.2% (Hadoop), 5.9% (Spring) and 3.8% (OpenStack) for the Shallow-LLP and 7.7% (Hadoop), 8.7% (Spring) and 7.3% (OpenStack) for the DL-LLP.

Since we observe that both logging level predictors (Shallow-LLP and DL-LLP) suffer from the data leakage and concept drift problems, we further investigate the following RQ:

RQ3: What is the best performing time-based modeling strategy for logging level prediction models?

Shallow-LLP contextual models trained on a window of historical data statistically significantly outperform (in terms of AUC) all-knowing models (i.e., trained using the entire history) in a median (over different time frame sizes) of 94%, 82% and 86% of the testing time frames (i.e., testing datasets of a given size) for Hadoop, Spring and OpenStack respectively. Meanwhile, we do not observe a statistically significant difference between the performance of contextual DL-LLPs and that of the all-knowing DL-LLP, which is likely due to the trade-off between data quality and quantity that is more pronounced in the case of data-hungry models like the DL-LLP. Furthermore, we find that in the context of the Shallow-LLP no training time frame size is consistently better than the other sizes in all testing time frames. Therefore, the size of the training data for contextual models should be considered as a hyperparameter to tune for LLPs.

While the prior research questions evaluate the impact of time-issues on the performance of LLPs, the following research question investigates the impact of time on the interpretation of LLPs, as time-changing interpretation can be

unreliable and confusing. We only investigate Shallow-LLPs, since they are more interpretable compared to the DL-LLPs.

RQ4: How does the interpretability of the logging level prediction models change over time? The drivers of logging level choice change over time, a phenomenon captured by the fluctuations in our shallow model's interpretability metrics. In fact, the correlation between the ranking of the most important features of different contextual models is very weak to weak for a median (across different time frame sizes) of 88%, 80% and 79% of the pairs of compared contextual models (e.g., 6 months contextual model trained on time frame T1 and 6 months contextual model trained on time frame T2) for Hadoop, Spring and OpenStack, respectively. Up to 40% (Hadoop), 30% (Spring) and 22% (OpenStack) of the most important features that are shared between different contextual models with the same training size can have a contradictory impact (i.e., an important feature has a positive impact on a given contextual model while the same feature has a negative impact on another contextual model).

We summarize our contribution as follows: (1) Quantifying the impact of data leakage and concept drift on state-of-the-art LLPs. (2) Studying the impact of time on models' interpretation. (3) Evaluating different time-aware modeling strategies designed to eliminate/mitigate time-related issues. Our results suggest that logging level prediction models (both Shallow-LLP and DL-LLP) suffer from data leakage and concept drift. While such time-related issues can be mitigated using contextual models for Shallow-LLPs, the identification of the right window size for

a contextual model should be considered as a hyperparameter to tweak when experimenting with logging level prediction models. Furthermore, our work provides developers with insights on how to mitigate time-related issues when leveraging existing LLPs, as well as a systematic approach to evaluate new LLPs for potential time-related issues. Such insights can improve the tooling efforts for future LLPs, and caution against the blind usage of existing LLPs that might not stand the test of live-environment usage.

Chapter structure: The chapter is structured as follows. Section 2 covers the approach used in this study. Section 3 presents our results. Section 5 discusses threats to the validity of our results. Finally, Section 6 concludes our study.

5.2 Methodology

In this Section, we discuss how we train, test and interpret our models, using a methodology similar to a large number of prior studies that leverage machine learning techniques to assist software engineering practitioners (Li et al., 2017a, 2021b; Ouatiti et al., 2023; Lyu et al., 2021a,b; Subramanyam and Krishnan, 2003; Herb-sleb and Mockus, 2003; Lee et al., 2020; Rajbahadur et al., 2021; Thongtanunam and Hassan, 2018).

In this chapter, we study the impact of concept drift and data leakage on logging level prediction models, so that we can understand how time context changes can affect models used to assist developers with logging activities. Logging level prediction models predict the appropriate logging level for a newly introduced logging statement, hence they require an ordinal predictor (i.e., categorical and ordered

dependent variable) such as the ordinal logistic regression model suggested by [Li et al. \(2017a\)](#) or the ordinal deep learning model suggested by [Li et al. \(2021b\)](#). For the purposes of our study, we leverage three popular, large and well-maintained open source software projects: Hadoop is a distributed computing system, Spring is a modular project that offers a vast pool of functionalities to Java Developers, and OpenStack is a cloud computing platform. Each of the studied projects has been developed and maintained for at least six years. Similar to prior work ([Li et al., 2017a, 2021b](#)), we collect all the revisions from the Github repositories of each of the studied projects. Next, we use the “git diff” command to collect code changes between revisions. The added logging statements along with their labels (i.e., logging levels) are then obtained using a regular expression used by prior works ([Li et al., 2017a, 2021b](#)). Finally, we extract the respective features (shown in [Table 5.1](#)) for the Shallow and DL-LPPs following each of the approaches from prior works ([Li et al., 2017a, 2021b](#)). We summarize information about the datasets used in our study in [Table 5.2](#).

To quantify concept drift and data leakage for logging level prediction models, we follow the modeling approach discussed in the remainder of this section. Our machine learning pipeline (i.e., data preparation, modeling, testing and feature importance) is similar to approaches followed by prior studies ([Li et al., 2017a; Lyu et al., 2021a,b; Ouatiti et al., 2023](#)). While our training and testing datasets differ based on each of our experiments (as discussed in the approach of each of our research questions), the following training and testing steps, also shown in [Figure 5.1](#), are common to all of the case studies of our chapter.

Model	Features class	Features description
Shallow-LLP	Logging statement	<ul style="list-style-type: none"> - Length of the logging statement. - Number of variables - Frequency of tokens in the logging statements.
	Containing block	<ul style="list-style-type: none"> - Number of LOC in the containing block. - Type of the containing block. - Exception type (if containing block is catch block).
	Containing file	<ul style="list-style-type: none"> - Logging statement density in the file. - Number of logging statements in the file. - Average logging statement length in the file. - Average number of variables in the logging statement of the file. - McCabe complexity. - Fan In.
	Change features	<ul style="list-style-type: none"> - Code churn - Logging statements churn. - Portion of changed logging statements among changed lines of code.
	Historic features	<ul style="list-style-type: none"> - Number of lines changed in the history of the containing file. - Number of revisions in history of the containing file. - Number of changed logging statements in the history of the containing file. - Portion of changed logging statements among changed lines of code in the history of the containing file. - Number of revisions that change logging statements.
DL-LLP	Syntactic context	- The sequence of AST nodes containing the logging statement (e.g., [Method Declaration , If statement, Logging statement, Method invocation]). The beginning of the sequence is marked by the start of the method and the end is marked by the end of the basic block (e.g., if block) that contains the logging statement.
	Logging message	<ul style="list-style-type: none"> - The sequence of natural language tokens in the logging message. * This sequence of tokens is inserted instead of the "Logging statement" tag in the example above (i.e., [Method declaration, If statement, startLogStmnt, this, is, a, logging, message, endLogStmnt, Method invocation])

Table 5.1: Features that are used to train the Shallow (Li et al., 2017a) and DL (Li et al., 2021b) LLPs

Project	Programming language	Span of data (days)	Number of added logging statements	Average per day
Hadoop	Java	3,790	16,841	4.44
OpenStack	Python	2,368	23,955	10.11
Spring	Java	3,207	6,018	1.87

Table 5.2: Studied Projects historical information

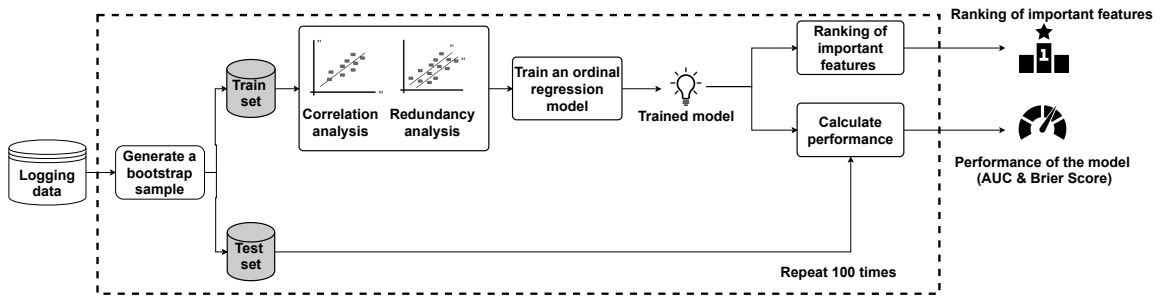


Figure 5.1: An overview of the methodology for training our models. Both training and testing datasets are further detailed in the approaches of our research questions.

Note that we leverage the same features (see Table 5.1) used by Li et al. (2017a) (ordinal regression) and Li et al. (2021b) (deep learning model) to train their respective models.

5.2.1 Bootstrap Sampling

Similar to prior work (Lee et al., 2020; Tantithamthavorn et al., 2019b) and our prior Chapter 4, we leverage the out-of-sample bootstrap validation technique to train and test our models. This approach consists of generating a sample with replacement (i.e., bootstrap sample) on which the model is trained. The model performance is then tested using the observations that were not selected in the bootstrap sample (i.e., out-of-bootstrap sample). This process of sampling, training and testing is repeated 100 times to guarantee robust findings. Note that for the

case of time-aware models (i.e., models taking into consideration the chronological order of data), our training bootstrap samples are always coming historically before the testing datasets, as explained in the approach section of RQ1. Note that this sampling strategy is used for both of our evaluated models.

Note that the two following steps (5.2.2 and 5.2.3) only apply to the Shallow-LLP trained using ordinal regression.

5.2.2 Correlation Analysis

For each training set, we conduct a correlation analysis to avoid erroneous model interpretation (Jiarpakdee et al., 2019; Tantithamthavorn and Hassan, 2018). Indeed, in order to guarantee a consistent ranking when interpreting models, it is recommended to discard one out of each pair of correlated features (Jiarpakdee et al., 2019). Similar to prior work (Lee et al., 2020; McIntosh et al., 2014), our correlation analysis leverages Spearman correlation with a threshold of 0.7. We opt for Spearman correlation due to its resilience to non-normally distributed data. Finally, we list our features by order of priority to guarantee a consistent correlation analysis across the different bootstrap iterations. The highly prioritized features are the ones related to exceptions (e.g., type of exception), the most important features reported by Li et al. (2017a) and the features related to logging activity (e.g., we keep log churn rather than code churn) as these features are characteristic to the logging activity. For each pair of highly correlated features (i.e., $\rho > 0.7$), we keep the feature with the highest priority.

5.2.3 Redundancy Analysis

As correlation analysis is not able to completely eliminate collinearity, we conduct a redundancy analysis similar to prior studies (Lee et al., 2020; Rajbahadur et al., 2021; Tantithamthavorn et al., 2020). The analysis consists of reducing the collinearity by iteratively identifying which independent feature is explainable by the other independent features. To do so, different preliminary models are built for each bootstrap sample, each of these models explains one independent feature with the other ones. We exclude an independent feature if its associated preliminary model has an $R^2 > 0.9$. We leverage the implementation of redundancy analysis provided by the *redun* function from the *rms* package ⁵.

5.2.4 Training and Testing

5.2.4.1 Shallow-LLP - Ordinal regression model

Using the remaining features from the previous steps, we train our ordinal regression models –which are an extension of logistic regression for ordinal dependent variables– that predict the suitable logging level (i.e., ordinal output variable) for a given logging statement. These models are tested on a different dataset, according to the specific RQ under analysis. To evaluate our model, we consider the AUC (Area under the ROC curve) and Brier Score as two standard performance metrics similar to prior studies (Li et al., 2017a; Rajbahadur et al., 2021; Tantithamthavorn et al., 2020). While the AUC measures the discrimination ability of the model (the higher above 0.5, the better), Brier Score captures how often a model can predict

⁵<https://cran.r-project.org/web/packages/rms/index.html>

the right class. An AUC of 50% is equivalent to a random guess. The Brier score ranges between 0 and 2 and the lower it is, the better the model is. A random guess model has a Brier Score of 0.8. While Brier Score is designed for multi-class evaluation, we leverage the multi-class AUC score generalization proposed by Hand and Till (Hand and Till, 2001), as it has been vastly used by prior work (Li et al., 2017a; Zhang et al., 2016; Sarro et al., 2022) to evaluate multi-class classifiers (e.g., Human expert effort estimation), and implemented by popular machine learning libraries (e.g., Scikit-learning⁶ and pROC⁷).

5.2.4.2 DL-LLP - Ordinal deep learning model

Using the features explained in Table 5.1, we train a deep learning model similar to the approach followed by Li et al. (2021b). This DL-LLP is a state-of-the-art logging level predictor, that uses the syntactic features (e.g., containing blocks) and log message features to predict an ordinally encoded output (i.e., the logging levels). The architecture of the DL-LLP consists of an embedding layer, an RNN and an output layer. While the embedding layer takes the input features (i.e., contextual and log message features) and represents them in the form of a sequential vector (i.e., the sequence of code statements within the block that contains the logging statement), the RNN layer (Bi-LSTM) is used to train the deep learning model. Finally, the output layer receives the output from the RNN layer and generates an ordinal output reflecting the predicted logging level. We refer to Li et al. (2021b)

⁶https://scikit-learn.org/stable/modules/generated/sklearn.metrics.roc_auc_score.html

⁷<https://www.rdocumentation.org/packages/pROC/versions/1.18.0>

for further details about the implementation. We leverage the AUC score to evaluate our DL-LLPs.

Note that the approaches used to perform our empirical evaluation in RQ1, RQ2 and RQ3 (as explained in each RQ's approach) are the same regardless of the used model. For instance, we quantify concept drift for the Shallow-LLP in the same way that we quantify it for the DL-LLP.

5.2.5 Feature Importance

Similar to prior work ([Lee et al., 2020](#); [Thongtanunam and Hassan, 2018](#)), we use Wald's χ^2 to evaluate the impact of each feature on predicting the logging level. A large χ^2 for a given feature mirrors the large explanatory power of that feature. From each of our 100 bootstrap iterations, we obtain a ranking of features based on importance. We then apply a Scott-Knott ([Tantithamthavorn, 2018](#)) clustering technique on the 100 rankings, similar to previous work ([Lee et al., 2020](#); [Rajbahadur et al., 2021](#); [Thongtanunam and Hassan, 2018](#)), in order to aggregate those rankings into a final ranking. The Scott-Knott approach uses a hierarchical clustering to group the means of importance scores into groups that are statistically different. We leverage these generated rankings to compare the interpretability of the different models we train.

Note that for the context of our study we used approaches (i.e., analysis of variance) aiming for the global explainability of our evaluated logging level predictor ([Tantithamthavorn et al., 2021](#)), which are widely used in the field of software engineering ([Subramanyam and Krishnan, 2003](#); [Herbsleb and Mockus, 2003](#); [Lee et al., 2020](#); [Rajbahadur et al., 2021](#); [Thongtanunam and Hassan, 2018](#); [Ouatiti](#)

et al., 2023) and are similar to how the models we evaluate were interpreted by Lee et al. (2020). That is, we do not interpret how the model takes individual decisions (aka., model-agnostic techniques).

5.3 Results

In this section, we present the results for each of our RQs. For each RQ, we discuss the motivation, the approach we used to address the RQ, and our findings.

5.3.1 RQ1: What is the impact of data leakage on the performance of logging level prediction models?

Motivation: The goal of this research question is to quantify the impact of data leakage on the way logging level prediction models were evaluated by prior studies (Li et al., 2017a, 2021b). In fact, ML models suffering from data leakage typically offer unrealistic performance that might not be replicable in a life-environment setup. Having insights about data leakage for LLPs would help practitioners better evaluate their LLPs and safeguard against unrealistic performance expectations.

Approach: To quantify the impact of data leakage on logging level prediction models, we compare the AUC and Brier Score performance of models built using two separate approaches that are described as follows:

- *The random-based model:* trained using a random train/test split that is susceptible to data leakage. The random split does not take into consideration the chronological order of data, as shown in Figure 5.2. For example, a model

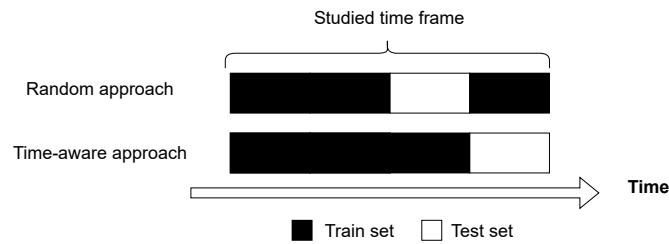


Figure 5.2: Overview of the modeling approaches for a single studied time frame.

that uses the first two batches and the last batch of data for training might suffer from data leakage when predicting on the testing batch (3rd batch). This potential data leakage is due to the fact that the model might gain knowledge coming from the future (4th batch) on the testing data (3rd batch). This approach was leveraged by [Li et al. \(2017a\)](#) to train their logging level prediction models.

- *The time-aware approach model*: trained taking into consideration the chronological order of data, as shown in Figure 5.2. The time-aware approach represents how the model would be used in practice (i.e., without having access to training data from the future).

Note that the difference between the two approaches is related to the train/test splits, while the leveraged features and the learning algorithm (i.e., ordinal regression) remain the same.

To train our two models, we follow the approach shown in Figure 5.3. In particular, we consider the following steps to compare the random-based models and the time-aware models:

First, we split the whole history of data equally. Each part of the history is hereafter referred to as a “time frame”. A time frame has a length of a number of months

(e.g., four-months time frame) and contains all the existing logging statements in those months. In our chapter, we evaluate different time frame sizes that range between four and 24 months to make our results generalizable to different time frame sizes. The lower bound that we used is a four-month time frame, since this is long enough to train and test our models without risking overfitting.

Then, we build both types of models:

- (i) The random-based model is trained on a bootstrap sample from a time frame (e.g., the first 4 months of a project) and tested on the out-of-bootstrap-sample portion of the same time frame. We consider the steps discussed in Section 5.2.4 to train our model. To guarantee that each time frame has enough data to train our model, we set a threshold of 300 observations for every training time frame, in order to have 10 features per observation (Harrrell, 2001).
- (ii) For the same time frame, we train a time-aware model on a bootstrap sample from the chronologically first 70% of the data and test that model on the following 30% of the same time frame data. Similarly to the random model, we consider time frames with at least 300 observations for training a model.

We repeat the previous two steps 100 times by leveraging different bootstrap samples to end up with a distribution of 100 performance (i.e., AUC and Brier Score) measurements for the random-based models and another distribution of 100 measurements for the time-aware models. We statistically compare these two distributions to identify whether they are different using a Wilcoxon test ($\alpha = 0.01$). If so, we also measure the amount of differences as well as the magnitude of such a difference using Cohen's d . Note that the median performance values reported

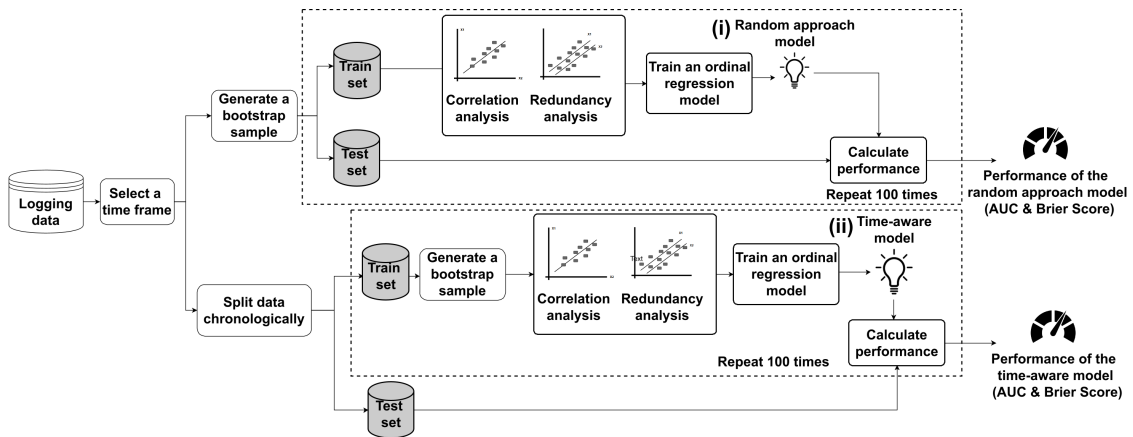


Figure 5.3: An overview of the methodology for quantifying data leakage

in our findings for individual models (e.g., LLP trained on 4 months worth of data) are calculated over the 100 bootstrap samples. Meanwhile, global findings about a given project (e.g., LLPs trained using Hadoop data) are aggregated based on the data frame size (e.g., median over the different time frame sizes used to train a contextual model).

We repeat the same experiments on the following time frames (e.g., the 2nd 4 months of a project) of the same size. Finally, we repeat everything for the other time frame sizes (e.g, a time frame of 5 months), up to 24 months.

We leverage the same approach to quantify data leakage for both of our evaluated logging level predictors (i.e., Shallow and Deep learning).

Results: Random-based models overestimate the AUC performance on the test sets by a median⁸ that ranges between 3% and 7% (Shallow-LLP) and between 2% and 9% (DL-LLP) compared to time-aware models across our evaluated time frame sizes and case studies. In fact, Hadoop’s Shallow-LLP trained using the random approach overestimates the time-aware models by a median⁸ of 2% (observed

⁸Median over the 100 bootstrap samples.

for four-month based models) to 17% (observed for seven-month based models). The overestimation ranges between 1% and 14% for Spring and between 1% and 12% for OpenStack. For example, our evaluated four-month random-based models inflate the AUC performance compared to the four-month time-aware models by a median⁸ AUC of 2%, 2%, and 3% for Hadoop, Spring and OpenStack respectively, as shown in Figure 5.4. We observe similar overestimation for the DL-LLP as shown in Figure 5.5 for our evaluated four-month based models. Such an overestimation indicates that one has to re-evaluate the LLP models using the time-aware approach as there is a chance that using a simple random-data splitting approach can overestimate the performances of these models, hence mislead practitioners that might consider the model as good when it has low performances (as close as a random guess as we observed in the case of Hadoop Shallow-LLP with six month time frames). Note that our observations stand for the Brier Score as well. The figures for the other time frame sizes are available in the online appendix (Ouatiti, 2024).

Across the different time frame sizes, a median of 90%, 75%, and 71% (Shallow-LLP) and a median of 57%, 67%, and 100% (DL-LLP) of our evaluated random-based models are statistically significantly (Wilcoxon test; $\alpha = 0.01$) better-performing than our evaluated time-based models for Hadoop, Spring and OpenStack respectively. For example, we observe that 67%, 37% and 67% of the four-month random-based Shallow-LLPs statistically significantly (Wilcoxon test; $\alpha = 0.01$) outperform the four-month time-aware models for Hadoop, Spring and OpenStack respectively. Similarly, we observe that 91%, 50% and 55% of the four-month random-based DL-LLPs statistically significantly (Wilcoxon test; $\alpha = 0.01$)

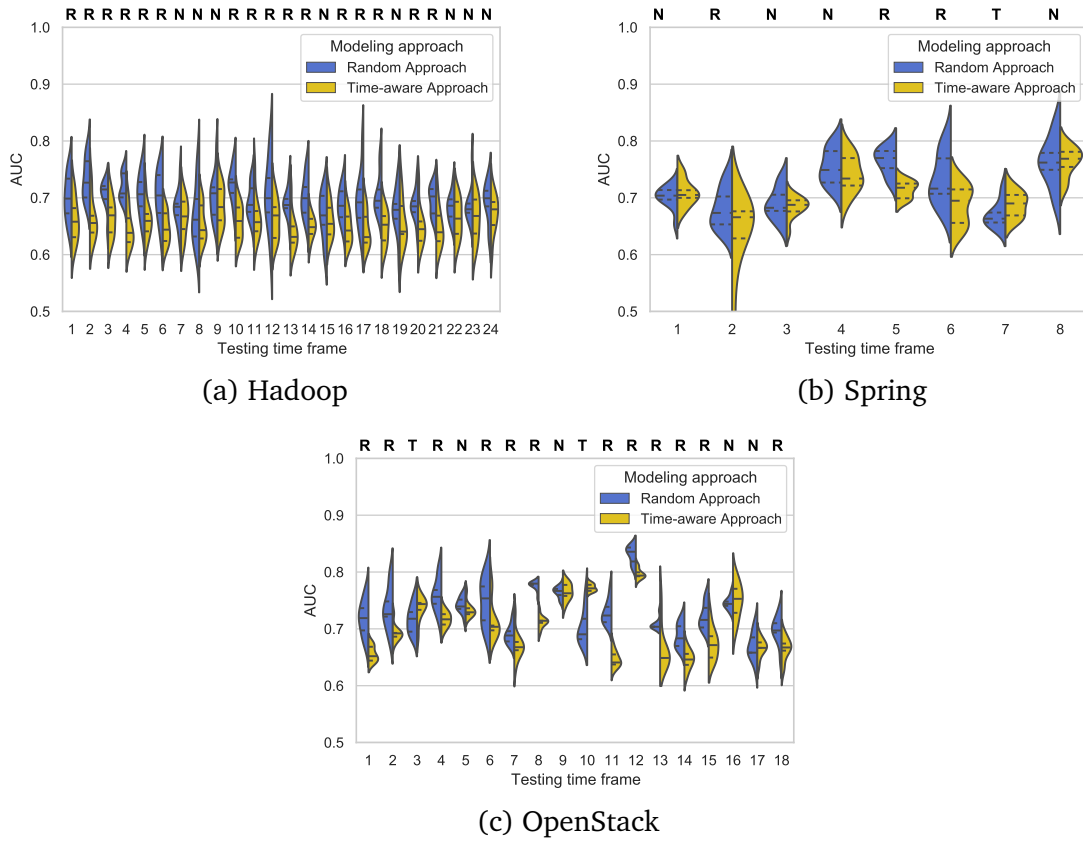


Figure 5.4: AUC performance of the random and time-aware Shallow-LLPs on four-month testing time frames. **R** indicates time frames where the random approach is the best performing, **T** indicates time frames where the time-aware approach model is the best performing and **N** indicates time frames where there is no significant difference between the performance of the two models.

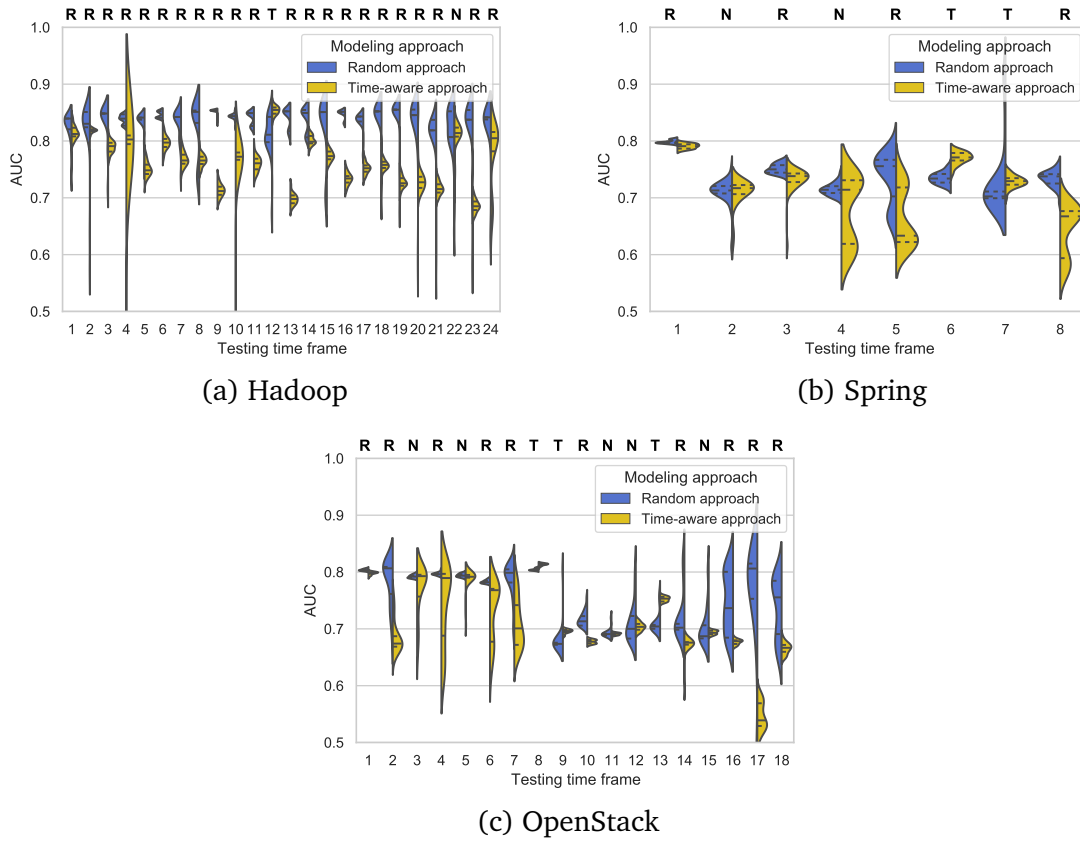


Figure 5.5: AUC performance of the random and time-aware DL-LLPs on four-month testing time frames. **R** indicates time frames where the random approach is the best performing, **T** indicates time frames where the time-aware approach model is the best performing and **N** indicates time frames where there is no significant difference between the performance of the two models.

outperform the four-month time-aware models for Hadoop, Spring and OpenStack respectively. All of these differences have a large effect size (Cohen's d , $d > 0.7$). On the other hand, only a median (over different time frame sizes) of 0%, 0% and 14% of the time-aware Shallow-LLPs statistically significantly outperform the random-based models. We observe similar results for the DL-LLPs, as only a median (over different time frame sizes) of 0%, 0% and 0% of the time-aware DL-LLPs statistically significantly outperform the random-based models.

These findings can be explained by the fact that the distributions of the independent features are different between the training and test sets that are leveraged for our time-aware models. We observe that a median of eight to 13 (depending on the evaluated time frame size) features are statistically significantly different (Wilcoxon test, $\alpha = 0.01$) between the training and test datasets that are used for our Hadoop time-aware models. This median number of statistically different features ranges between 8 and 14 for Spring and five and nine for OpenStack. For example, our four-month based model has a median⁹ of nine, 11 and seven features (out of 25) that are statistically significantly different between the training and testing sets of the time-aware model trained on Hadoop, Spring and OpenStack respectively. Note that the OpenStack project, which has the most time frames in which the time-aware model outperforms the corresponding random-based model (median of 14%), is also associated with the lowest number of significantly different features between the training and testing sets used by the time-aware model.

⁹Median over the 100 bootstrap samples.

Summary of RQ1

logging level prediction models trained using the random approach can overestimate the expected performance of a time-aware logging level predictor by a median AUC up to 17% (Shallow-LLP) and 27% (DL-LLP) higher. **Our findings suggest that one should leverage time-aware approaches for shallow and DL LLPs for more realistic performance estimations.**

5.3.2 RQ2: What is the impact of concept drift on the performance of logging level prediction models?

Motivation: The goal of this research question is to quantify concept drift on logging level prediction models. While log choice strategies might vary from one time period to another (e.g., when debugging vs. after fix), it is not clear whether a concept drift caused performance decrease exists for LLPs, and if such a performance decrease exists, how soon after the end of the training dataset the concept drift can be manifested. Such insights would warn practitioners on the importance of updating their models so they can better maintain their LLPs (e.g., plan for updates).

Approach: To quantify the impact of concept drift on logging level prediction models, we train a model (Deep and shallow LLPs) on a selected time frame and test that model on each of the following time frames, as described in the following steps and illustrated in Figure 5.6.

We split the whole existing data into equal time frames. For each time frame (TF), we perform the following:

- We train a model on a bootstrap sample from the first 70% of the observations of TF.
- We test our model on the remaining 30% of the observations to obtain a *baseline performance*.
- We test our model on each two-month time frame that chronologically follows TF and compare the obtained performance to the baseline performance. The two-month testing time frame guarantees enough observations (i.e., at least 50 observations) to have statistically significant findings. Note that we implement Bonferroni correction ([Haynes, 2013](#)), as we are performing multiple comparison tests.

We repeat the same analysis with 99 other bootstrap samples from the chronologically first 70% of the observations of TF. We end up with 100 baseline performance measurements and 100 performance measurements for each of the two month testing time frames.

We then repeat all previous steps with a different training window size. Starting with a time frame of four months (first 70% for training and remaining 30% to measure the baseline evaluation) up to 24 months, with an increment of two months.

Results: Across the different time frame sizes, the performance of our evaluated LLPs (Shallow and deep) drops significantly a median of 1.5 (Hadoop) or one (Spring and OpenStack) testing time frame after the end of the training data period. For example, we observe that the Shallow-LLP trained using four-month time frames (i.e., four-month based Shallow-LLP) takes a median¹⁰ of 1.5, 1

¹⁰Median over the 100 bootstrap samples.

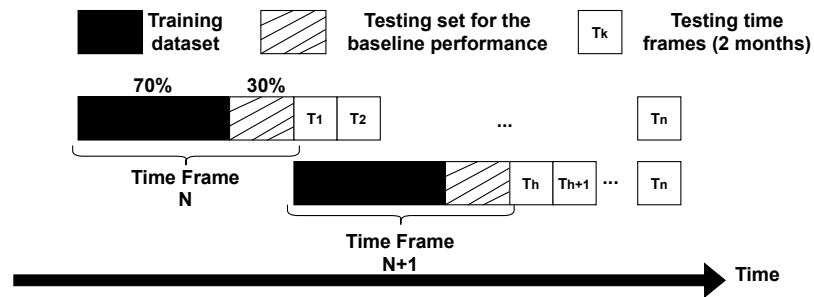


Figure 5.6: Overview of model evaluation under concept drift for a specific time frame size.

and 1 testing time frames to drop statistically significantly (Wilcoxon test; $\alpha = 0.01$) below the baseline performance in terms of AUC for Hadoop, Spring and OpenStack respectively. Meanwhile, the four-month based DL-LLP takes a median¹² of one, one and 1.5 time frames to drop below the baseline performance.

Furthermore, the performance of our Shallow-LLP is statistically significantly (Wilcoxon test; $\alpha = 0.01$) lower than the baseline performance in a median¹² of 56% (observed for 24-month based Shallow-LLPs) to 85% (observed for 10-month based Shallow-LLPs) of the testing time frames for Hadoop. The same median¹² percentage ranges from 60% to 96% and 30% to 81% for Spring and OpenStack respectively. Figures 5.7, 5.8 and 5.9 highlight the concept drift for our evaluated four-month based Shallow-LLPs. We observe similar findings for our DL-LLPs (shown in Figures 5.10, 5.11 and 5.12), as the performance on testing time frames is statistically below the baseline performance in a median¹² of 78% (observed for four month based DL-LLPs) to 100% (observed for eight-month based DL-LLPs) of the testing time frames of Hadoop. Such a median¹² of time frames in which the DL-LLPs perform statistically significantly below the baseline ranges between 79% and 98% and between 89% and 100% for the respective DL-LLPs of Spring and

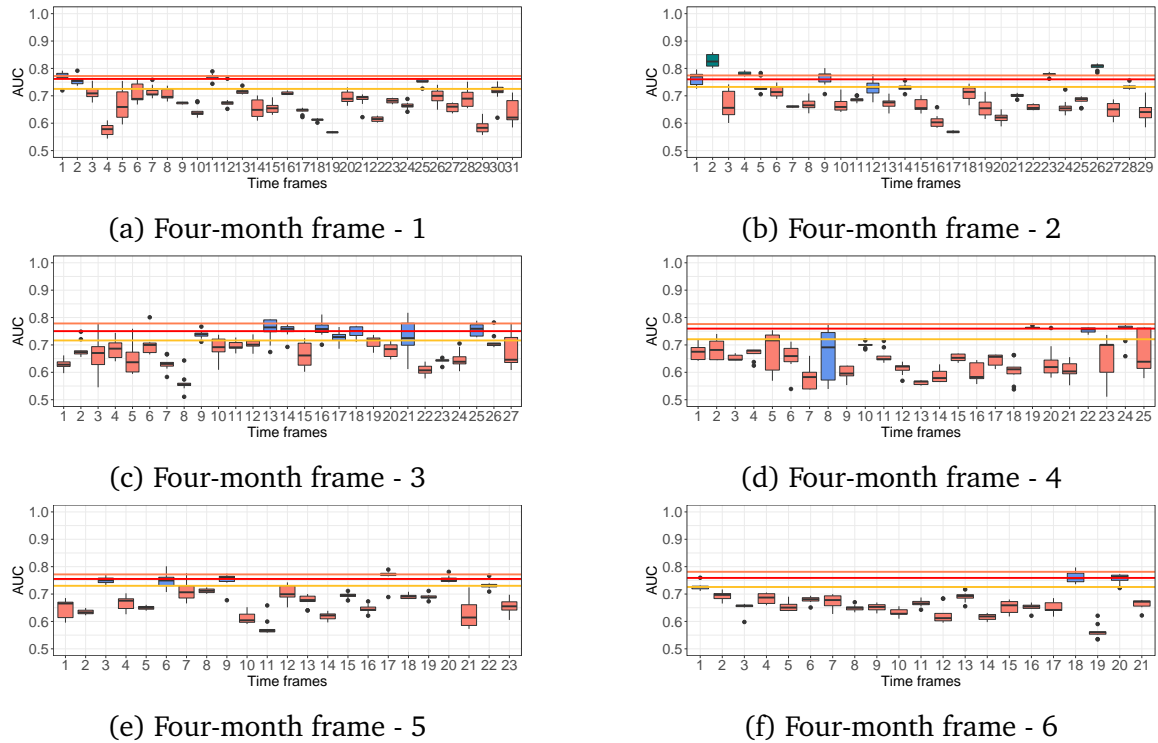


Figure 5.7: AUC performance of Hadoop’s Shallow-LLPs across time. the lines represent the 1st quantile, median and 3rd quantile for the baseline AUC performance. Red boxplots show time frames with a performance statistically below the baseline, blue boxplots show no statistical difference and green boxplots show time frames with a performance statistically better than the baseline.

OpenStack. Finally, we observe that the performance of our models (Shallow and DL) statistically significantly exceeds the baseline performance in a median¹¹ time ranging from 0% to 8%, 0% to 13% and 0% to 40% (Shallow-LLP) and in 0% to 0%, 0% to 50% and 0% to 0% (DL-LLP) of the testing time frames of Hadoop, Spring and OpenStack respectively. This range depends on the size of the training time frame.

The diminishing performance of the LLPs can be explained by the fast increase in the number of statistically different independent features between the baseline

¹¹Median over the 100 bootstrap samples.

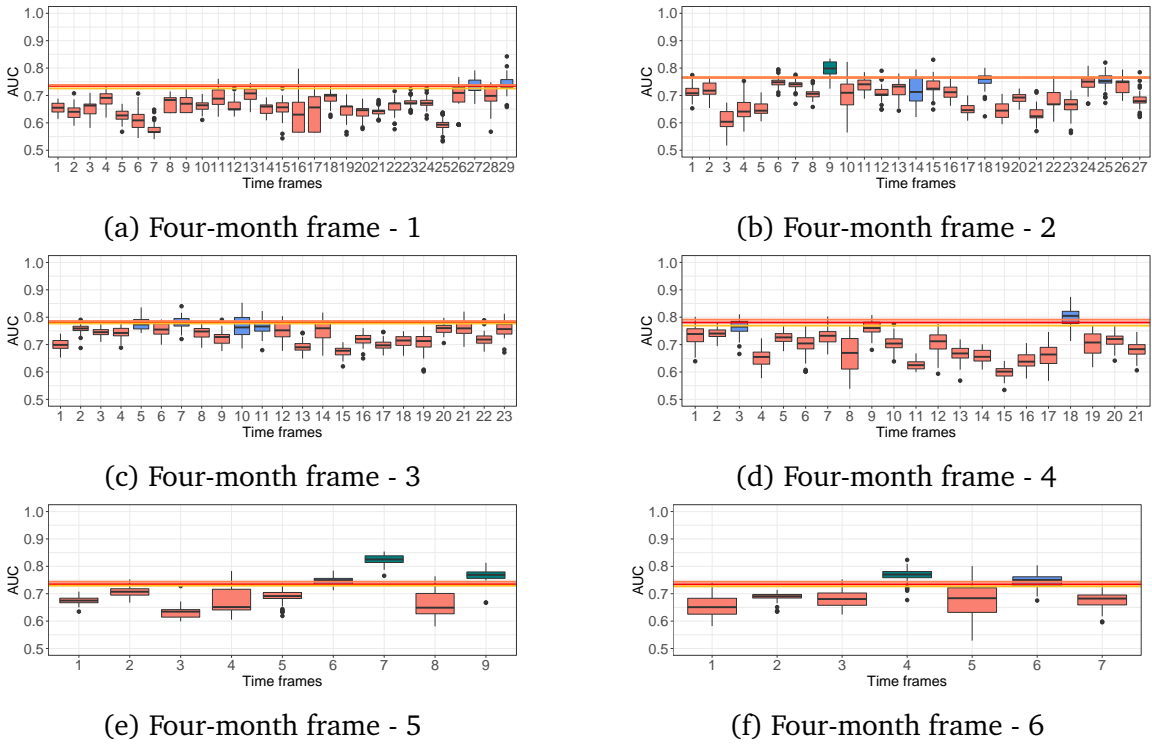


Figure 5.8: AUC performance of Spring’s Shallow-LLPs across time. the lines represent the 1st quantile, median and 3rd quantile for the baseline AUC performance. Red boxplots show time frames with a performance statistically below the baseline, blue boxplots show no statistical difference and green boxplots show time frames with a performance statistically better than the baseline.

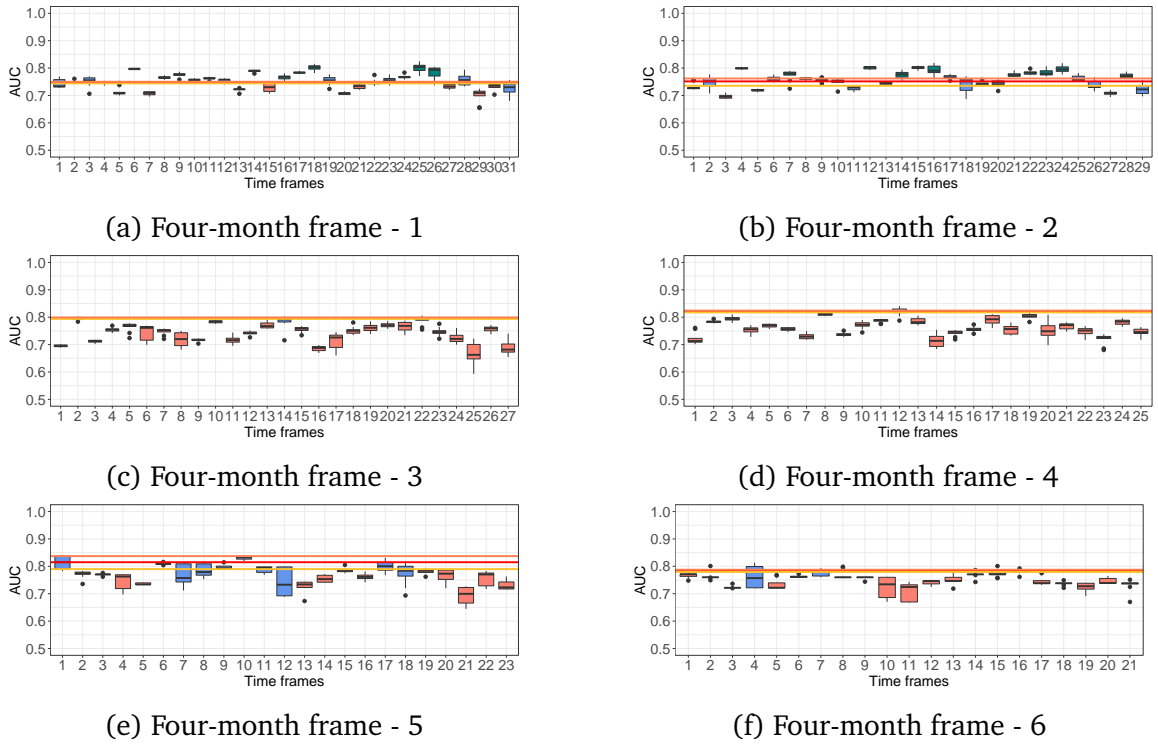


Figure 5.9: AUC performance of OpenStack’s Shallow-LLPs across time. the lines represent the 1st quantile, median and 3rd quantile for the baseline AUC performance. Red boxplots show time frames with a performance statistically below the baseline, blue boxplots show no statistical difference and green boxplots show time frames with a performance statistically better than the baseline.

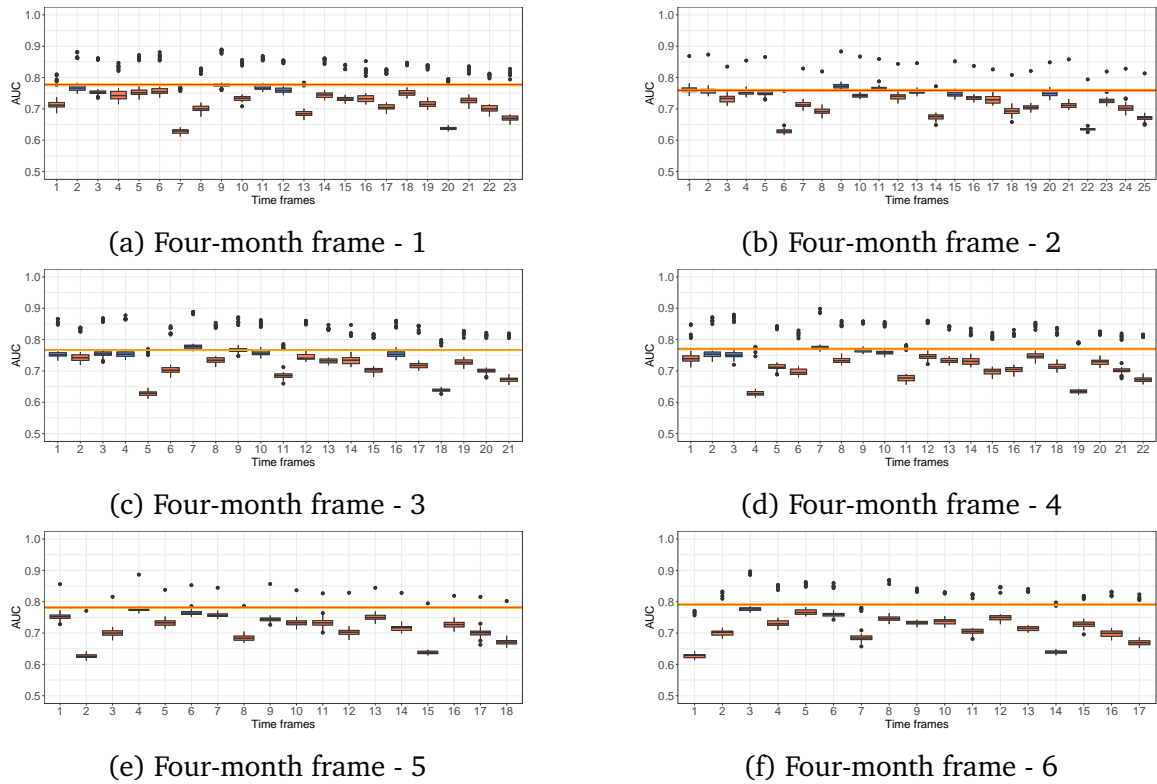


Figure 5.10: AUC performance of Hadoop’s DL-LLPs across time. the lines represent the 1st quantile, median and 3rd quantile for the baseline AUC performance. Red boxplots show time frames with a performance statistically below the baseline, blue boxplots show no statistical difference and green boxplots show time frames with a performance statistically better than the baseline.

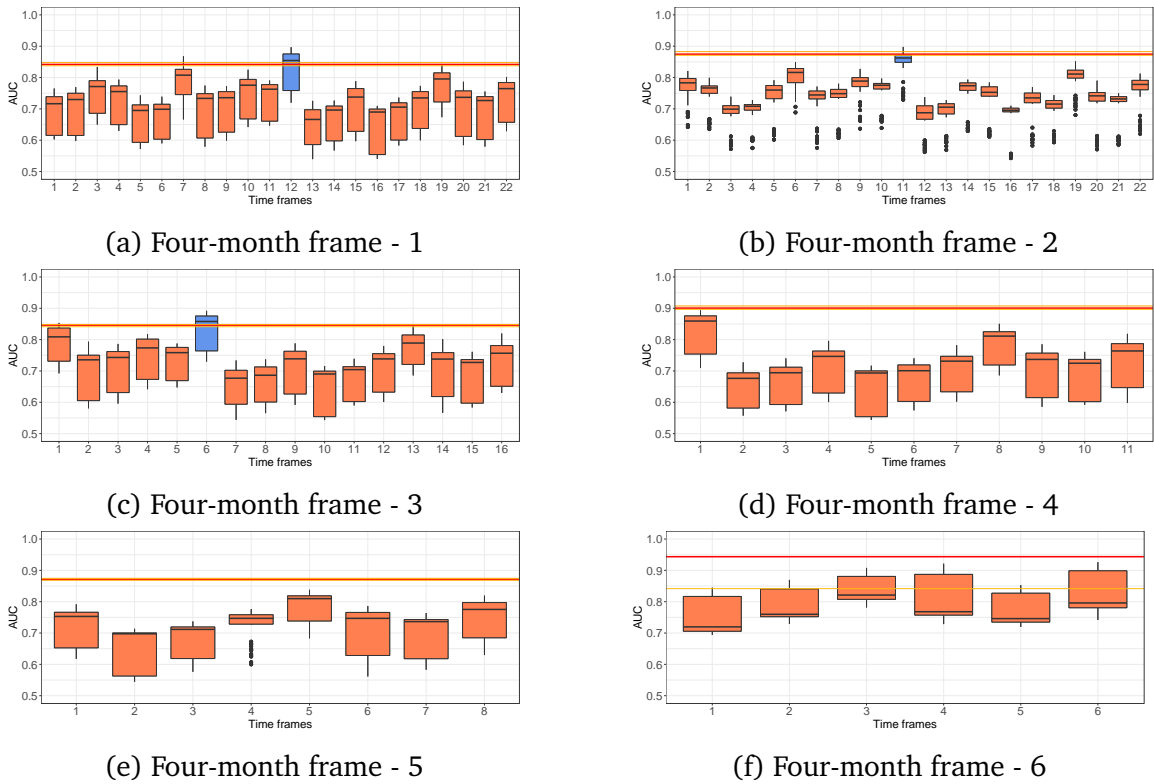


Figure 5.11: AUC performance of Spring’s DL-LLPs across time. the lines represent the 1st quantile, median and 3rd quantile for the baseline AUC performance. Red boxplots show time frames with a performance statistically below the baseline, blue boxplots show no statistical difference and green boxplots show time frames with a performance statistically better than the baseline.

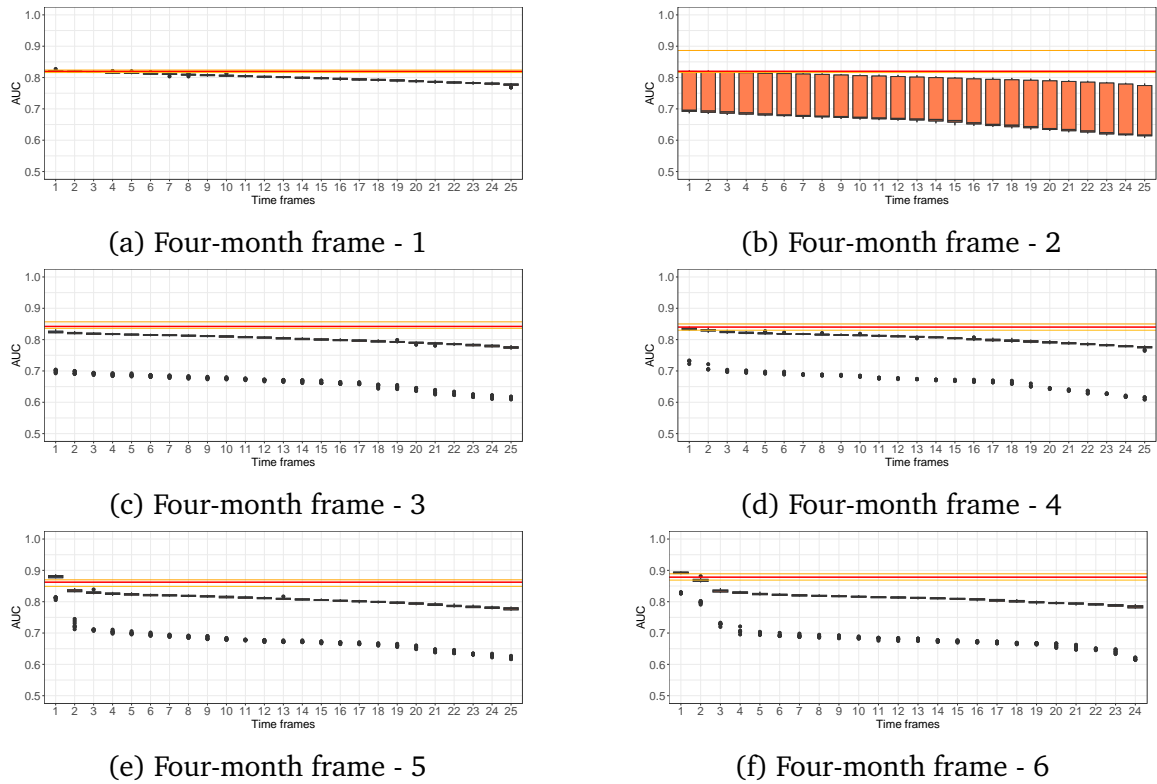


Figure 5.12: AUC performance of OpenStack’s DL-LLPs across time. the lines represent the 1st quantile, median and 3rd quantile for the baseline AUC performance. Red boxplots show time frames with a performance statistically below the baseline, blue boxplots show no statistical difference and green boxplots show time frames with a performance statistically better than the baseline.

model data and the future test sets. In fact, we observe that at least 50% of the features leveraged by the Shallow-LLP model, including those features related to the log message and the context of the logging statement (used to train the DL-LLP) showed statistically significant differences between the baseline data and the data of the testing time frames, after a median¹² time ranging between one and three (Hadoop), 1 and 1.5 (Spring) and 1 and 5.5 (OpenStack) time frames.

Additionally, we observe that 52%, 56% and 34% of the features of adjacent testing time frames of Hadoop, Spring and OpenStack are statistically significantly different (Wilcoxon, $\alpha=0.01$). For example, the churn of logging statements for the OpenStack project in the time frame between August and October 2014 (a median of 36) is statistically significantly different (Wilcoxon test; $\alpha = 0.01$) than the churn for the logging statements of the same project (i.e., OpenStack) in the following time frame (a median of 31). In fact, maintainers of OpenStack conducted a number of logging maintenance¹³ activities (e.g., changing verbosity of logs) in the period between August and October 2014, which resulted in a higher churn for logging statements in that period (median 36) compared to the next time frame (median churn 31) as well as the previous time frame (median churn 21). Only 0%, 1% and 2% of the adjacent testing time frames had the same distribution for Hadoop, Spring and OpenStack, respectively.

Across the different training time frame sizes, the performance of 27% (Hadoop), 63% (Spring) and 72% (OpenStack) of our Shallow-LLPs exceeds the baseline performance after dropping below it. In fact, our Shallow-LLPs statistically significantly (Wilcoxon test; $\alpha = 0.01$) exceed the baseline performance on

¹²Median over time frame sizes from 4 to 24.

¹³<https://github.com/openstack/openstack/commit/56cc320240c983742c467f7afd7cc6b11dde8625>

a median¹⁴ of 0% (observed for the four-month based models) to 4.1% (observed for the 18-month based models) of the testing time frames for Hadoop. Similarly, this occurs on a median¹⁵ of 0% to 13% and 0% to 44% of the testing time frames of Spring and OpenStack respectively. For example, the four-month based model for all our evaluated projects statistically significantly exceeds the baseline performance in a median¹⁵ 0% of the testing time frames, as shown in Figures 5.7, 5.8 and 5.9.

Concept drift might be a more serious problem for DL-LLPs, as their performance typically never becomes as good as the baseline performance after dropping below it. In fact, we observe that the performance of a median of 0% (Hadoop), 50% (Spring) and 0% (OpenStack) of the DL-LLPs (depending on the training frame size) exceeds the baseline performance after dropping below it, which indicates that DL-LLPs are less likely to re-exceed the baseline performance compared to Shallow-LLPs.

¹⁴Median over the 100 bootstrap samples.

¹⁵Median over the 100 bootstrap samples.

Summary of RQ2

Logging level prediction models suffer from concept drift as their AUC performance drops on future testing time frames after a median^a of just 1.5, 1 and 1 testing time frames for Hadoop, Spring and OpenStack respectively. The effect of concept drift is more severe on the DL-LLPs, for which the performance drops significantly below the baseline performance in 22% (Hadoop), 19% (Spring) and 59% (OpenStack) more testing time frames than the Shallow-LLP. **Our results suggest the need for continuous concept drift tracking and frequent updates to the LLPs, especially the DL-LLP, whose performance is less stable throughout time compared to the Shallow-LLP.**

^aMedian over the time frame sizes from 4 to 24 months.

5.3.3 RQ3: What is the best performing time-based modeling strategy for logging level prediction models?

Motivation: Since previous research questions suggest using time-aware approaches, we explore different strategies to train time-aware models as an approach to mitigate the impact of concept drift. Specifically, we evaluate two time-aware modeling strategies in this research question: *contextual model* that leverages recent data and *all-knowing-model* that leverages the whole history of data. In particular, leveraging just the most recent data in a contextual model might not benefit from recurring events that exist throughout the whole history of data. On the other hand, leveraging the whole available historical data in an all-knowing-model might contain noisy

data that are drifting from the current data. In this research question, we quantify such a trade-off by comparing the two approaches for both the Shallow-LLP and the DL-LLP to identify which of the two approaches better fits the logging level prediction models..

Approach: To compare the two time-aware models, we split the whole history of the available data into two-month time frames, which are used as testing time frames. We compare on each of these testing time frames how our two evaluated types of models perform. We train and test our two types of models as discussed below:

- *All-knowing model:* We train a model that leverages all the existing data prior to each of our testing time frames. For example, we train a model on data from the first three years for a testing time frame that covers the 37th and 38th months of a project.
- *Contextual models:* We train contextual models on data that belongs to N months prior to each of our testing time frames. For our experiment, we evaluated different time frame sizes (i.e., N) that range from four to 24 months, with two months increment, to train our contextual models. In other words, we train a four-month, six-month, eight-month, and up to 24-month based models for each of our testing time frames. For the early testing time frames, we train contextual models based on the amount of existing data. For example, we train a four-month to 12-month models for the testing time frame that covers the 13th and 14th months of a project. Note that we consider the 12-month model, in our example, as the all-knowing model since it is trained on the whole available data.

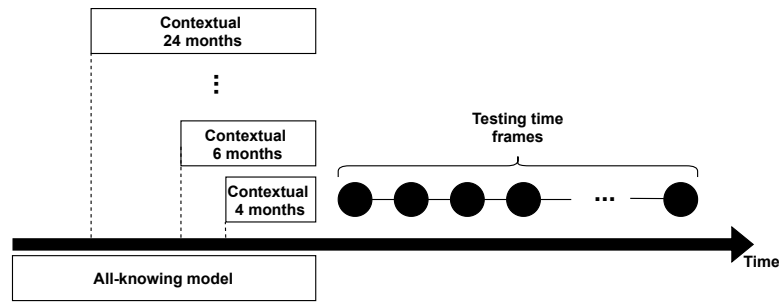


Figure 5.13: Comparison of contextual and all-knowing models. We compare the models shown in the figure just on their first following testing time frame. We train new contextual and all-knowing models for each testing time frame.

For both types of models, we evaluate 100 models, each on a different bootstrap sample such that we obtain for each model 100 performance measurements to use for a statistically robust comparison. Note that the comparison of two models is performed on the same test time frame. In other words, we do not compare two models on different testing time frames, as shown in Figure 5.13. Note that the same approach is followed for both the Shallow-LLP and the DL-LLP.

Results: We observe that at least one contextual Shallow-LLP statistically significantly outperforms the all-knowing Shallow-LLP on 94%, 82% and 86% of testing time frames for Hadoop, Spring and OpenStack respectively. We also observe that the all-knowing Shallow-LLPs statistically outperform all the contextual Shallow-LLPs in only 5%, 8%, and 0% of our Hadoop, Spring and OpenStack testing time frames, respectively.

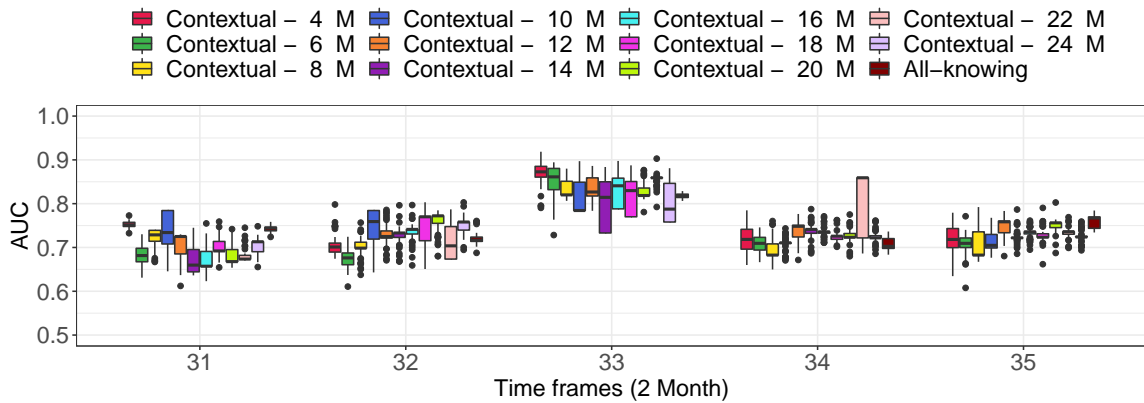
While 51%, 31% and 22% of the cases in which the contextual Shallow-LLP outperforms the all-knowing model have a large effect size (Cohen’s d) for Hadoop,

Spring and OpenStack respectively, we observe that 0% of the cases in which the all-knowing Shallow-LLP outperforms the contextual Shallow-LLPs with a large effect size for both Hadoop and Spring projects.

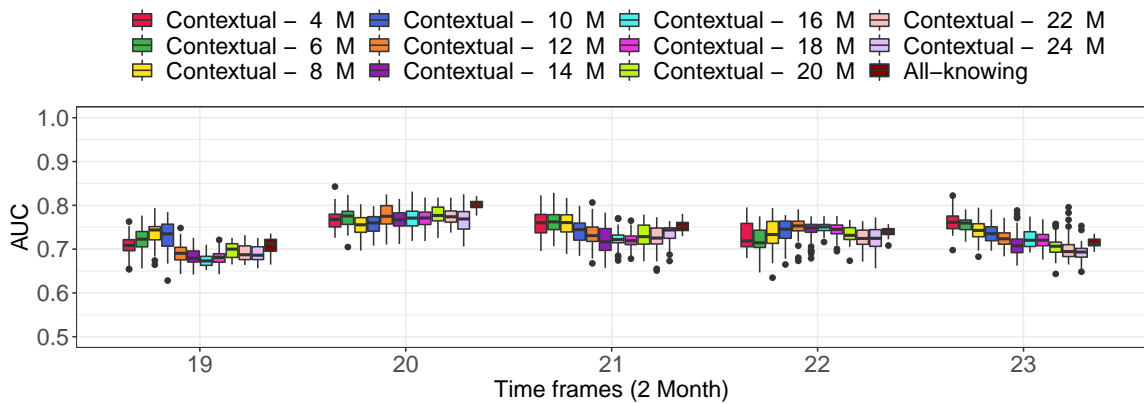
Furthermore, across the different testing time frames, we observe a median of one, three and three contextual Shallow-LLPs that outperform the all-knowing Shallow-LLP for Hadoop, Spring and OpenStack respectively. For example, three Spring contextual models (6-month, 8-month, and 10-month based contextual models) outperform the all-knowing model on the nineteenth testing time frame, while the all-knowing model outperforms the 22-month and 24-month based contextual model on the same testing time frame.

As for the DL-LLPs, we do not observe a significant performance improvement when comparing the all-knowing and the best contextual approach, as shown in Figure 5.15. In fact, the AUC performance difference between the best contextual DL-LLP and the all-knowing DL-LLP is negligible on 83%, 82% and 92% of the testing time frames of Hadoop, Spring and OpenStack respectively. One reason why contextual DL-LLPs are not performing as well as the Shallow-LLPs compared to their respective all-knowing LLPs might have to do with the data quality vs. quantity trade-off (Bertram et al., 2022). In fact, the contextual DL-LLPs are trained on good quality data (i.e., avoiding noise of irrelevant past data) but that data might not be large enough for a data-hungry model such as the DL-LLP.

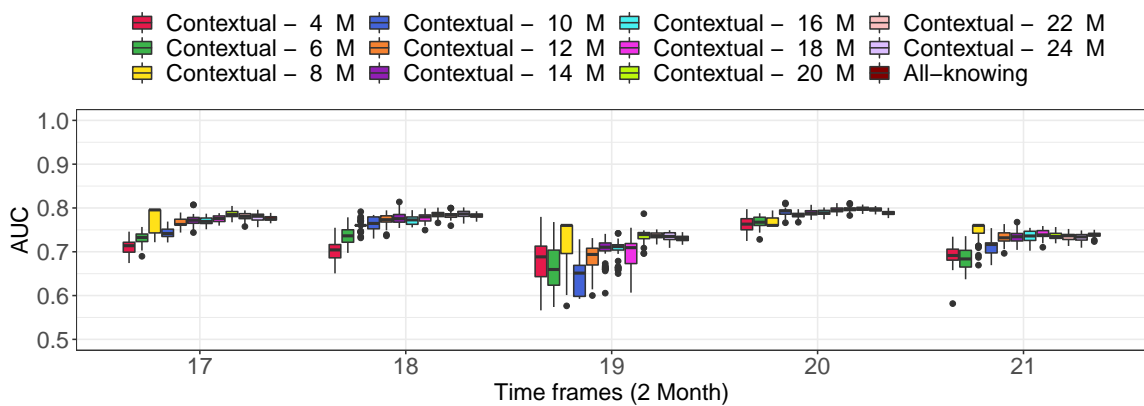
As we do not observe a statistical difference between the performance of all-knowing and contextual DL-LLPs, we focus in the remainder of this RQ on contextual Shallow-LLPs for which we consistently observe at least one contextual model that exceeds the performance of the all-knowing Shallow-LLP.



(a) Hadoop



(b) Spring



(c) OpenStack

Figure 5.14: The AUC Performance of contextual and all-knowing Shallow-LLPs on a selection of testing time frames (two-month long) - The remaining time frames (e.g., 1 to 35 for Hadoop) are available in the appendix.

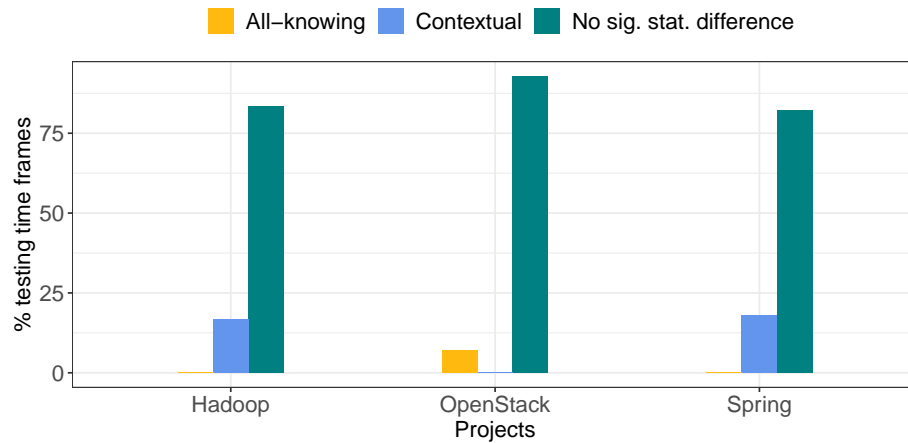


Figure 5.15: Percentage of testing time frames in which one of our time-aware DL-LLPs (i.e., contextual or All-knowing) is the statistically significantly best performing model in terms of AUC.

No contextual Shallow-LLP consistently performs the best on all of our evaluated testing time frames, as shown in Figure 5.16. In fact, we do not observe a clear pattern about contextual models that perform the best, as six, seven and eight different contextual models perform the best on at least one testing time frame of Hadoop, Spring and OpenStack respectively. For example, the best Hadoop contextual model for the 33rd testing time frame is the four-month based-model, while it is the 16-month based model for the 34th testing time frame.

Despite performing well on some testing time frames, a contextual model can poorly perform on other time frames. For example, the four-month based contextual model of Hadoop has a median¹⁶ AUC of 0.86 on the 33rd testing time frame as shown in Figure 5.14, while the same contextual model has a median¹⁶ AUC performance of just 0.72 on the 34th time frame.

¹⁶Median over the 100 bootstrap samples.

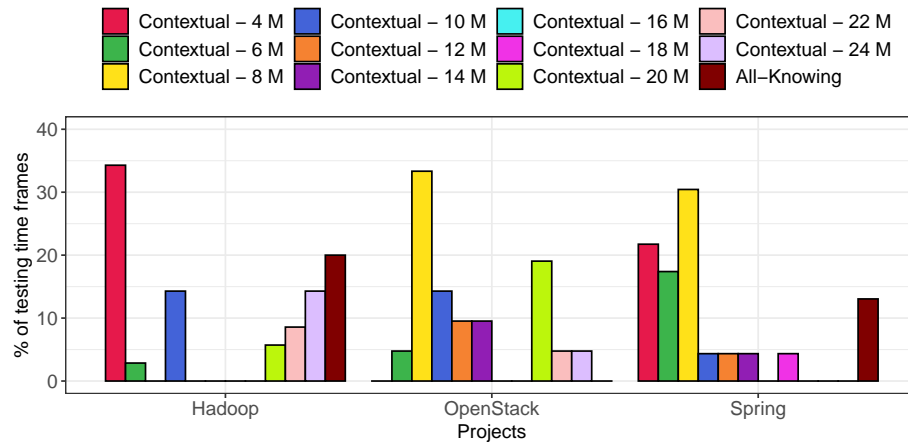


Figure 5.16: Percentage of testing time frames in which the contextual Shallow-LLPs perform the best in terms of AUC.

Additionally, we observe that the size of the contextual model (i.e., number of months used to train it) has a negligible to moderate correlation with the performance of our evaluated contextual models, as Spearman’s correlation between the size and performance of our contextual models is -0.05, -0.3 and 0.47 for Hadoop, Spring and OpenStack respectively.

We also investigate ML modeling techniques aiming to improve the performance our time-aware Shallow-LLPs. Specifically, we implement two ML models that take advantage of the nature of our evaluated time-aware approaches. While the first is an ensemble model (Zhou, 2012) in which our contextual Shallow-LLPs vote for the appropriate logging level across all frame sizes, the varying model leverages a subset of features to train an all-knowing Shallow-LLP. We chose a threshold of six features for our varying models (i.e., a maximum of six most varying features can be removed), in order to guarantee enough features for training representative varying models (i.e., avoid issues merely due to lack of features).

We observe that the ensemble learning approach does not bring any significant improvement to the performance. In fact, the best performing contextual Shallow-LLP outperforms the ensemble learning model in 95%, 95% and 82% of the testing time frames. Additionally, the improvement (if any) brought by the ensemble learning approach to the best performing time-aware Shallow-LLP on a given time frame is negligible for all of our studied projects except for one time frame for Hadoop project.

As for our all-knowing varying Shallow-LLPs (shown in Figure 5.17), we do not observe a significant difference between the performance of the model trained using all features and the performance of models trained using all features except the n features (n from 1 to 6) that vary the most. Specifically, we observe that the models omitting most varying features (one or more) outperform the model that uses all features in only 25% (Hadoop), 43% (Spring) and 55% (OpenStack) of our testing time frames.

Summary of RQ3

While there is no difference in terms of performance between the contextual and all-knowing DL-LLPs, at least one contextual Shallow-LLP statistically outperforms the all-knowing Shallow-LLP on our evaluated testing time frames. **Our results suggest including the size of the contextual model as a hyperparameter to tune when experimenting with contextual LLPs.**

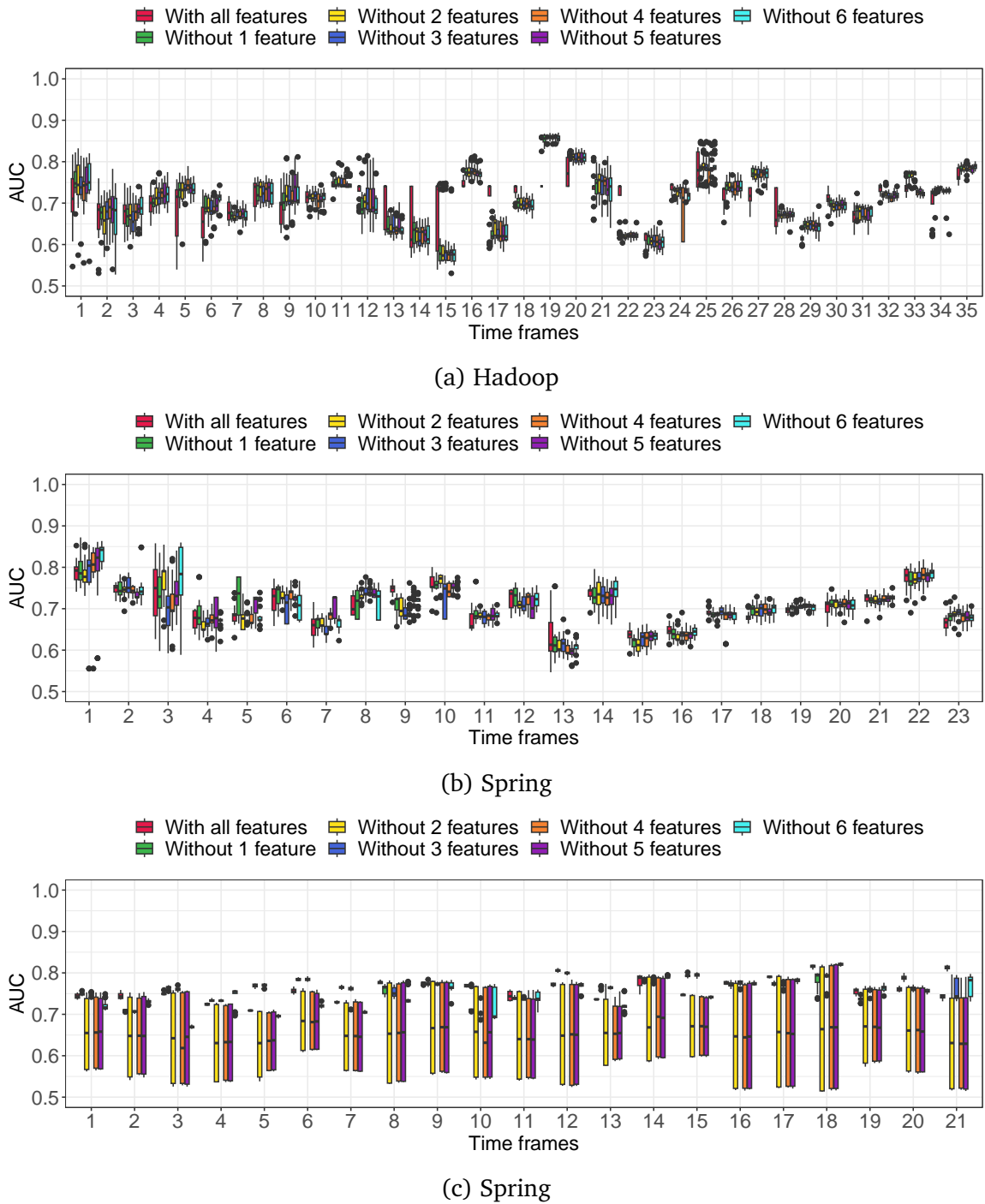


Figure 5.17: The AUC Performance of the varying all-knowing Shallow-LLP on all testing time frames (two-month long)

5.3.4 RQ4: How does the interpretability of the logging level prediction models change over time?

Motivation: The goal of this research question is to quantify how the interpretability changes over time for Shallow-LLPs (which our previous RQ indicates as the more fitting for handling time-related issues), so that practitioners can better understand what impacts logging level decisions in a particular time period. In fact, previous findings also indicate that the performance of Shallow-LLPs trained using a time-aware approach can change significantly from one time frame to the other (regardless of the training time frame size). Such changes can be due to changing logging patterns across time. Therefore, we investigate if the change in performance is accompanied with an interpretability change, so updating models is important for understanding the important factors related to logging level prediction. The interpretability is relevant (especially for models not leveraging deep learning such as the Shallow-LLP) for practitioners in their decision making such that shifting or conflicting feature importance rankings might be confusing for practitioners.

Approach: To investigate logging level prediction models interpretability across time, we train time-aware Shallow-LLPs following the steps below, which are highlighted in Figure 5.18.

We split the history of our studied project into equal time frames (TF) of size S (e.g., four months). For each time frame, we train 100 logging level prediction models using 100 bootstrap samples, similarly to the other research questions.

We then extract a ranking of important features from each of the 100 models, as well as the positive or negative impact each of these features can have on the

logging level prediction, similar to prior work (Shihab et al., 2013; Sayagh et al., 2017). For instance, a feature can have a positive impact on the prediction if increasing the value of that feature increases the prediction probability and vice-versa. To determine if a feature F has a positive or negative effect on a logging level L , we first calculate the probability (P1) of predicting L using features that are set at their median values. Next, we increase the value of the feature F by one standard deviation from its median while keeping the other features at their median values, and re-predict the probability (P2) of the logging level L . The two probabilities P1 and P2 are then compared to determine the type of impact feature F has on logging level L .

Since each model has a different ranking of features, we cluster the 100 rankings using the Scott-Knott clustering algorithm into a single ranking. Similarly, we summarize the impact of a feature as a vote. A feature has a positive impact if that feature has a positive impact on the majority of our 100 trained models. Note that we rarely observe different impacts of the same feature on the 100 models that are trained on the same time frame.

We repeat the same analysis on each of the following time frames to obtain a ranking as well as the impact of each important feature.

We then statistically compare each pair of the obtained rankings, as well as the rankings of the pair of successive time frames using Spearman Correlation. Since we are conducting multiple comparison tests, we additionally leverage Bonferroni's correction (Haynes, 2013). Note that a ranking correlation ranging between 0 and 0.4 is considered weak to very weak, a ranking correlation ranging between 0.4 and 0.6 is considered moderate, and a ranking correlation higher than 0.6 is considered

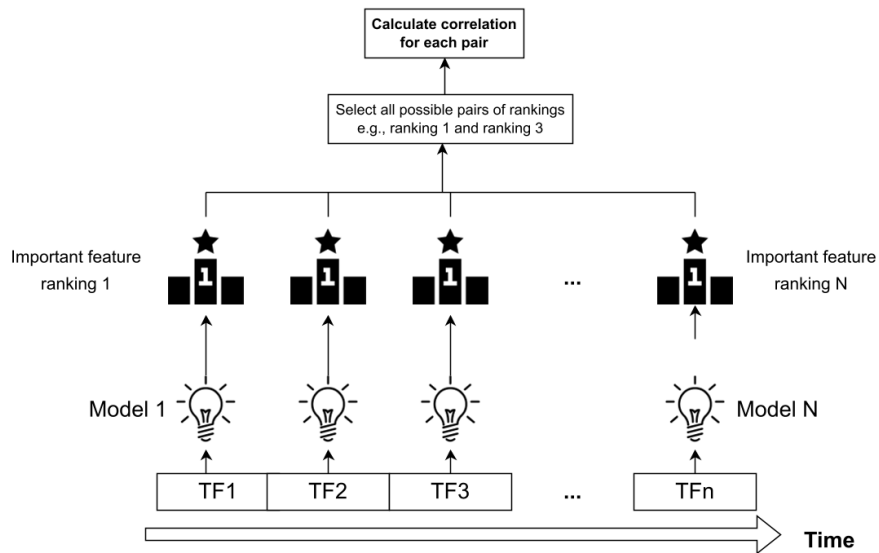


Figure 5.18: Overview for comparison of interpretability across time

strong to very strong. Additionally, we leverage the information about features’ impact in order to detect contradictory features (i.e., features with different impact across two or more time frames). Specifically, we compare whether a feature has a positive impact on the models of a given time frame, while it has a negative impact on the models of another time frame.

We repeat the same analysis with different time frame sizes, starting from four and up to 24 months similarly to the previous research questions. Note that this research question focuses on contextual Shallow-LLPs, as they outperform the all-knowing Shallow-LLP (according to our findings of RQ3).

Results: Across different time frame sizes, a median of 88%, 80% and 79% of the model pairs exhibit a very weak to weak correlation in terms of feature rankings for Hadoop, Spring and OpenStack respectively. For instance, we observe 90%, 87% and 79% of model pairs trained on a six-month time frame exhibit

a very weak to weak important features correlation. Furthermore, we report a median (across different time frame sizes) of 12%, 20% and 27% of model pairs that have a strong to very strong correlation in terms of important features ranking. For example, only 14%, 20% and 20% of the six months-based model pairs have strong to very strong important features correlation for Hadoop, Spring and OpenStack, respectively.

Furthermore, successive pairs of models do not have consistent rankings of the most important features. We observe that the ranking of important features for a median (across different time frame sizes) of 31%, 40% and 0% of the successive training time frames is weak to very weak for Hadoop, Spring and OpenStack respectively. For instance, the percentage of successive frames with weakly correlated ranking of important features when considering six-month based models is 33%, 29% and 50%. Meanwhile, the percentage for successive time frames with strongly correlated features is 13%, 42% and 21% for Hadoop, Spring and OpenStack respectively.

Additionally, we observe that the median (over our evaluated time frame sizes) percentage of important features that are common across all models is 40%, 54% and 44% for Hadoop, Spring, and OpenStack. That percentage increases as the size of the frame gets smaller, as the correlation between the size of the time frame and the number of different features is strongly negatively correlated (Spearman's $\rho = -0.9, -0.93$ and -0.97 for Hadoop, Spring and OpenStack respectively). For example, while the percentage of common features for Hadoop models that are trained on six months time frames is 26%, that percentage is 94% for models trained on frames with a size of 48 months. We think that training models on small time frames

(assuming enough data points for training) allows to unveil specific patterns to those time periods (reflected by the highly different feature importance results), as opposed to models trained on larger time frames for which feature importance becomes similar. For example, OpenStack's three models that are trained on 24 months time frames have 17 out of 18 of their important features in common.

Despite observing models trained on different time frames with common important features, these features can have a contradictory effect. In fact, the median percentage of contradictory features over shared features between a pair of models ranges from 0% to 40% (depending on the evaluated time frame size), 0% to 30% and 0% to 22% for Hadoop, Spring and OpenStack, respectively. For example, we observe that while having more variables in a logging statement increases the probability of that logging statement to have the *Info* logging level according to Hadoop's six-month based-models trained between May and November 2012, the opposite is observed for the model trained on the next six months time frame, as increasing the number of variables within a logging statement reduces the probability of predicting the logging level *Info*.

We observe that the percentage of important features that remain important after retraining a logging level prediction model later in time (aka., feature survival rate) exhibits a different pattern than models studied by a prior study (Olewicki et al., 2022). While brown test (i.e., tests that trigger false positive build failures) prediction models studied by Olewicki Olewicki et al. (2022) show a monotonically decreasing feature survival rate, our logging level prediction models show a fluctuating feature survival rate, as shown in Figure 5.19. Across the different training

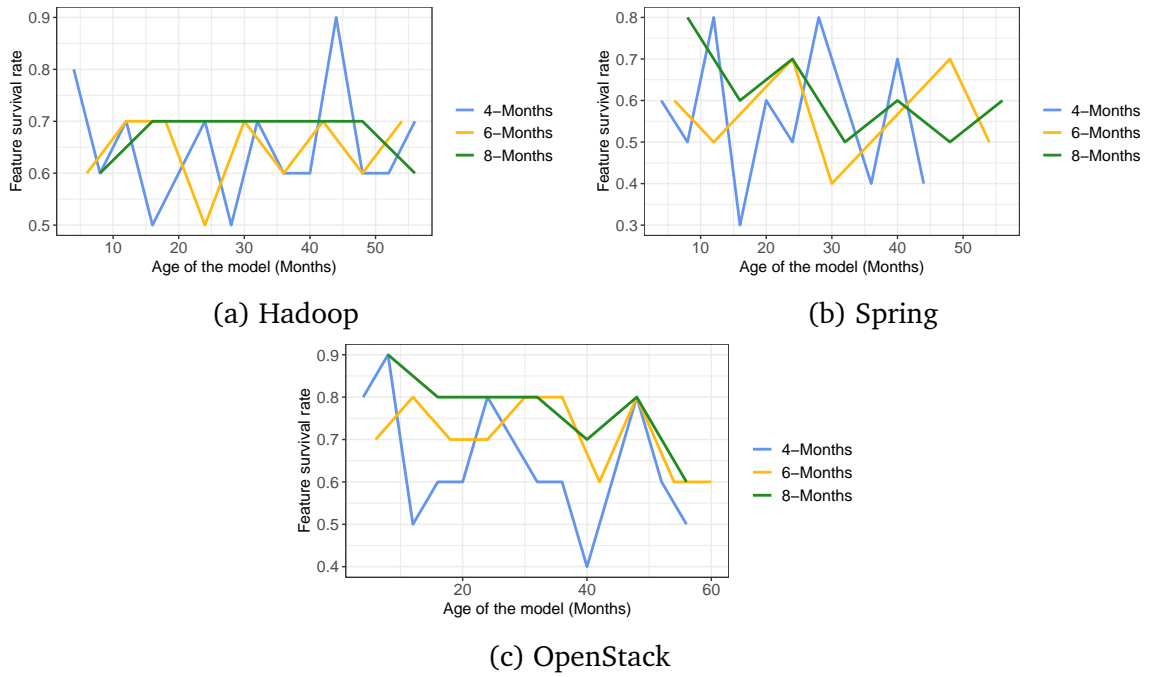


Figure 5.19: Feature survival rate for logging level prediction models. For clarity purpose, we only show the four, six and eight months based models.

sizes, the median feature survival rate ranges from 60% to 70%, 55% to 65% and 60% to 80% for Hadoop, Spring and OpenStack respectively.

Summary of RQ4

The interpretability of logging level prediction models changes across time, as the important features are different across a median of 88%, 80% and 79% of the model pairs for Hadoop, Spring, and OpenStack respectively. The smaller the time frame, the more unique its models' interpretability. **Our results suggest the use of recently trained models for more accurate interpretability.**

5.4 Threats to Validity

5.4.1 External validity

One external threat to the validity of our work concerns the generalization of our results to other software systems as well as models. Even if our study covers three popular, large software projects maintained for a long period, and our experiments targeted different time frames and time frame sizes, we do not generalize our results to other software systems or other machine learning models. We also encourage future studies to replicate our study on other software engineering models as well as software systems.

5.4.2 Internal validity

One internal threat to the validity of our work regards the time frame sizes used to train and test our models. For instance, leveraging a different time-frame size can show a different result. To mitigate this threat, we leverage different time-frame sizes ranging from four to 24 months and we leverage 100 bootstrap samples for each trained model to make our analysis statistically robust. Similarly, our comparison between the all-knowing models and the contextual ones can be different with different contextual models sizes. To mitigate this threat, we also compared the all-knowing models to different contextual models, each of which is trained on a different sample size of four to 24 months. Another internal threat to validity regards the quality of the logging levels selected by developers. In fact, developers can introduce inaccurate logging level choice which can impact the overall quality of our datasets. However, we observe that the logging statements in our datasets

are not changed frequently (3.8% to 7.5% of the logging statements across our evaluated projects).

5.5 Chapter Summary

logging level prediction models proposed by prior work (Li et al., 2017a, 2021b) leverage a set of metrics to predict the appropriate logging level for a logging statement. While their models (Shallow-LLP and DL-LLP) show a good performance, the way these are evaluated can suffer from data leakage and concept drift, i.e., two time-related issues. Due to the change in logging level choice decisions across time as well as the context differences (e.g., domain, data) between the LLPs and previously studied models (i.e., defect prediction and AIOps models), one might be unclear whether LLPs are less impacted by data leakage and concept drift (e.g., stagnant logging practices), or whether they can be severely impacted by time-related issues (e.g., abrupt changes to logging strategy).

Since prior work (Zhang and Tsai, 2002; Bennin et al., 2020) indicates that time-related issues (e.g., concept drift) impact software engineering models differently. Therefore, in this chapter, we quantify the impact of data leakage and concept drift on the performance and interpretability of logging level prediction models.

Our findings indicate that a data leakage risk exists for both the Shallow and DL-LLPs. Furthermore, we found that LLPs (especially DL-LLPs) need to be updated frequently as they can suffer from concept drift as soon as a couple of months post training.

As a means of mitigating time-related issues, we evaluated two types of time-aware models: *a*) contextual models that leverage data from a time frame of the

history, and *b*) the all-knowing model that leverages all the data available at the moment of its creation. Through the comparison of the performance of these time-aware models, we observe that the all-knowing Shallow-LLP –despite using more data– can perform significantly worse than some contextual Shallow-LLPs. Yet, no specific contextual Shallow-LLP consistently outperforms all the other contextual Shallow-LLPs. Therefore, we encourage considering the size of the contextual model as a hyperparameter to tune when training logging level prediction models spanning through time.

Finally, we investigated how the important features for logging level prediction change for a time-aware model. We observe that while some features are shared between all the contextual models, a larger portion (i.e., up to 40%) of the features are exclusive to a set of the contextual models. Furthermore, even when two contextual models share some important features, they may have contradictory effect on the prediction of a specific logging level.

Our results suggest paying attention to time-related issues when leveraging logging level prediction models. To mitigate the effect of these time-related issues, we recommend using contextual models that are time-aware (i.e., not susceptible to data leakage), easier to train than the global model and –given the right window size– perform at least as good as the global models. While finding the best performing shallow model is relatively fast (i.e., real-time retraining), we seek to develop in future work an approach to find the best contextual model for DL-LLPs.

CHAPTER 6

Retrieval Supersedes Scale: Efficient Logging Level Prediction via Multiplex Socio-Technical Context

This chapter is under review at the IEEE Transactions on Software Engineering journal (TSE) ([Ouatiti et al., 2026](#)).

DEVELOPERS insert logging statements in the source code to log its execution. The generated logs are essential for developers in predicting, detecting and troubleshooting system anomalies. A key part of these statements is the logging level (e.g., DEBUG, INFO, WARN, ERROR, FATAL), which determines whether a log message is recorded. Choosing an appropriate logging level is a complex trade-off as levels set too low (TRACE or DEBUG) mask important information in production, while levels set too high (ERROR or FATAL) create noisy

logs and impose extra I/O and storage overhead. A large body of research investigates how to predict the accurate logging level for a logging statement leveraging AI approaches. For instance, recent work leverages Large Language Models (LLMs) via In-Context Learning (ICL) to infer the suitable logging level for a logging statement. However, current approaches typically augment the prediction prompt with randomly sampled examples. We argue that this diversity-based approach overlooks a critical reality pertaining to the socio-technical nature of logging, where conventions may vary significantly by component and development teams. Consequently, off-the-shelf LLMs using random examples report median AUCs of only 0.64 to 0.75, and higher scores (up to 0.80) are typically only unlocked after expensive fine-tuning. In this chapter, we propose a retrieval approach for LLM based logging level prediction that prioritizes context relevance over random selection of ICL examples and basic similarity approaches typically used in ICL retrieval. Our ICL RAG-based method retrieves logging examples relevant along two socio-technical dimensions: (i) functionality (code performing similar operations) and (ii) ownership (code maintained by the same set of developers). Across four large Java projects, and using 4 different LLMs, this retrieval improves AUC over random baselines by 0.11 to 0.16 (median). Combining both signals yields median AUCs of 0.9 to 0.96. Furthermore, our ablation studies reveal that ownership signals are important for correctly predicting levels in files with high semantic noise, where pure code similarity fails. Our AUC results also significantly exceeds state-of-the-art standard supervised predictors (e.g., DeepLV) by a median between 0.14 and

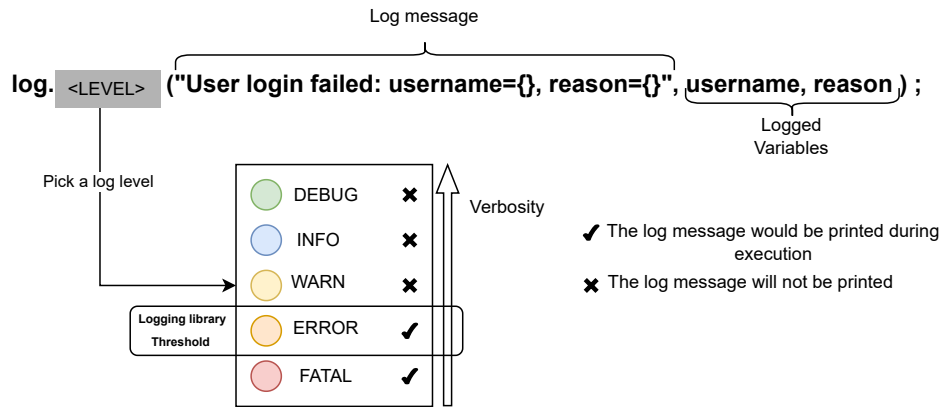


Figure 6.1: Logging statement template

0.19 across our evaluated projects. Our findings highlight the importance of leveraging socio-technical signals such as code functionality and ownership to inform LLM-powered logging level prediction.

6.1 Introduction

Developers insert logging statements into the source code of software systems to capture essential runtime information, facilitating debugging, system observability, and anomaly prediction, detection, and diagnosis (Li et al., 2021a; Shang and Hassan, 2015; Yuan et al., 2010). A typical logging statement includes a logging level (e.g., DEBUG), and a textual message with or without variables, as shown in Figure 6.1. logging levels are typically ordered by verbosity from DEBUG (most verbose) to FATAL (least verbose) in Java logging libraries. Such logging levels control the volume of logged data, as during software operation, only log messages with high verbosity (i.e., higher than the logging framework threshold) are recorded, while log messages with low logging levels are excluded.

However, accurately determining the appropriate logging level for each logging statement remains a challenging activity for developers (Li et al., 2017a; Pecchia et al., 2015), who typically make initial poor logging level choices (Li et al., 2021a; Oliner et al., 2012). Suboptimal logging level decisions have significant practical consequences as a too low logging level drastically reduces software observability, complicates debugging, and prolongs failure diagnosis (Yuan et al., 2010), while an excessive logging level introduces performance overhead, inflates storage, and overwhelms developers with redundant information (Yuan et al., 2014). Furthermore, modern observability stacks and AIOps dashboards rely heavily on high-quality log data. Consequently, inaccurate logging levels can introduce noise that degrades the performance of automated anomaly detection tools or leads to sparse data that make monitoring dashboards ineffective (He et al., 2021).

To assist developers in logging level decision-making process, researchers have proposed various automated logging level Predictors (LLPs) (Anu et al., 2019; Heng et al., 2024; Kim et al., 2020; Li et al., 2017a; Liu et al., 2022; Li et al., 2021b). Early approaches (Anu et al., 2019; Li et al., 2017a; Kim et al., 2020) relied on engineered software features (e.g., cyclomatic complexity) to train shallow classifiers such as ordinal logistic regression. Subsequent research shifted toward deep learning, leveraging learned embeddings of logging statements and their surrounding code context to capture semantic nuances (Liu et al., 2022; Li et al., 2021b). Most recently, the advent of generative AI has spurred the adoption of Large Language Models (LLMs) for logging level prediction, utilizing paradigms ranging from zero-shot inference to few-shot In-Context-Learning (ICL) and fine tuning (e.g., LoRA) (Heng et al., 2024).

Current LLM-powered logging level prediction approaches typically rely on random or stratified sampling for in-context examples, a practice that implicitly favors data diversity over contextual relevance (Heng et al., 2024; Ma et al., 2024; Song et al., 2016). While often chosen for simplicity, this design is supported by the assumed benefits of data diversity in deep learning (Gong et al., 2019), including (1) improved robustness on unseen inputs, (2) mitigation of selection bias, and (3) coverage of all output classes within the limited context window. Consequently, diverse, randomized sampling is widely treated as a reasonable default for conditioning frozen LLMs.

We argue that this approach (i.e., diversity-first) overlooks key aspects that are fundamental to the logging practice (Ouatiti et al., 2023). Specifically, projects are not a uniform entity, with a single, consistent logging strategy. In reality, distinct components or subsystems often exhibit different logging practices influenced by factors such as code functionality (i.e., what the code does) and developer ownership (i.e., who maintains or frequently changes a file). We hypothesize in this chapter that leveraging these internal distinctions can allow existing LLM-powered LLPs to use the most contextually relevant software knowledge, thereby enhancing their predictive accuracy and practical utility.

For example, the SIG-Auth team ¹ in the Kubernetes (k8s) contributing community is responsible for authentication logic that is architecturally dispersed across the API Server (control plane) and the Kubelet (node agent). Despite residing in completely different components with distinct semantic contexts, these files might exhibit highly consistent logging practices as they are written by the same people ().

¹<https://github.com/kubernetes/community/tree/master/sig-auth>

Similarly, the Hadoop common component contains a retry-handler (*RetryInvocationHandler.java*), which wraps RPC calls in a retry loop, logs each failed attempt at DEBUG, and escalates to “ERROR” after the last retry. Meanwhile, the Yarn component contains (*RequestHedgingRMFailoverProxyProvider.java*), a class that performs the same retry duty for Resource-Manager calls, logs attempts at DEBUG, and issues a final “WARN” once fail-over succeeds. Such examples demonstrate that logging conventions can be strongly influenced by shared developer ownership and functional similarities. Omitting these signals may limit the predictive accuracy of LLM-powered LLPs. We use the term socio-technical knowledge in this chapter to mean information that captures both the software system’s technical structure and evolution (e.g., components, code context, change history) and the social/organizational context around the code (e.g., ownership, team conventions, developer preferences), as it relates to logging decisions.

In this chapter, we investigate the impact of two socio-technical signals (i.e., code functionality and ownership) on the performance of in-context-learning LLM-powered logging level prediction. We propose a methodology that extracts and fuses two distinct socio-technical signals (i.e., code functionality and ownership) into the retrieval process. Specifically, we model a studied software project as a two-layer multiplex graph ([Battiston et al., 2014](#); [De Domenico et al., 2013](#)), where nodes represent source files and layers represent the distinct relationship types. While the first layer encodes functionality, where edges connect files with high semantic similarity, the second layer encodes ownership, where edges connect files maintained by the same group of contributors. By applying community detection

on this unified multiplex structure, we partition the project into coherent socio-technical clusters. Finally, for any given target file, we restrict the retrieval of in-context-learning examples exclusively to its assigned cluster, ensuring that the LLM is augmented with examples that are both functionally relevant and aligned with the team’s logging conventions.

It is important to distinguish logging level prediction from the broader task of end-to-end log generation. While generative tools aim to solve the ‘cold start’ problem by synthesizing messages for unlogged code, they introduce significant risk when applied to existing logging statements as regenerating a full statement can alter developer-tuned message content or variable selection, degrading log quality and increasing code review burden (Rawte et al., 2023; Yu et al., 2024). In practice, developers often simply need to ‘up-shift’ or ‘down-shift’ the visibility of an existing log message without rewriting it. Therefore, logging level prediction addresses a distinct calibration problem pertaining to the optimization of the trade-off between observability and overhead for developer-written log messages.

To evaluate our approach, we answer the following RQs:

RQ1: How does the integration of ownership and functionality signals via multiplex clustering impact the predictive performance of LLM-based LLPs?

Our approach, leveraging both functionality and ownership signals simultaneously via multiplex clustering, achieves a predictive performance, reaching a median AUC between 0.9 and 0.96 across our evaluated projects. The approach statistically significantly (Wilcoxon test, $\alpha = 0.01$) outperforms state-of-the-art approaches in LLM-powered logging level prediction (in terms of AUC) by a median of 0.12. Moreover,

this integration of socio-technical signals enables frozen LLMs to exceed the accuracy of prior specialized supervised models (e.g., DeepLV: +0.14, Ordinal Regression: +0.19) while avoiding task-specific model training and ongoing retraining costs.

RQ2: What is the individual contribution of functionality and ownership signals? Leveraging in-context-learning examples retrieved from our file ownership clusters alone statistically significantly enhances the precision of LLM-powered logging level prediction by a median of 2% to 7% (Wilcoxon test, $\alpha = 0.01$), compared to the state-of-the-art approaches in LLM-powered logging level prediction. Furthermore, leveraging in-context-learning examples retrieved from our functionality-only clusters statistically significantly enhances the AUC of LLM-powered logging level prediction by a median of 4% to 8% (Wilcoxon test, $\alpha = 0.01$), compared to the state-of-the-art approaches in LLM-powered logging level prediction.

RQ3: Can Multiplex Retrieval reduce the computational cost of LLM-based logging level prediction? Our findings indicate that precise context retrieval is more critical than raw model scale. A multiplex-enhanced 7B model outperforms state-of-the-art commercial models, achieving an average AUC of 0.927 surpassing GPT-4o by 17.6% and DeepSeek-V3 by 40.7%. We demonstrate that scaling to larger local models (34B) yields diminishing returns, offering only marginal performance gains (+0.9% AUC) at a disproportionate operational cost increase (+322%).

Our findings suggest the adoption of small, self-hosted and context-aware models over general-purpose commercial APIs for LLM-powered logging level prediction. We observe that integrating socio-technical signals (i.e., ownership and functionality) enables smaller models (e.g., 7B) to outperform much larger baselines,

proving that precise retrieval supersedes raw parameter scale. We suggest practitioners prioritize self-hosted, batched inference pipelines to balance accuracy with operational efficiency for logging level prediction tasks.

Chapter structure: This chapter is structured as follows. Section 2 details our methodology. Section 3 presents our empirical results. Section 4 discusses threats to validity, and Section 6 concludes the chapter.

6.2 Methodology

In this section, we introduce an in-context learning retrieval approach designed for logging level prediction. Our approach provides relevant logging examples to LLMs to generate logging levels for newly added or modified logging statements within source code, as shown in Figure 6.2. The input of our approach is the file name and the specific logging message for which a logging level prediction is to be predicted. The output is the predicted logging level. To perform the prediction, we first select in-context logging examples from the cluster that is relevant both in terms of functionality and ownership to the logging statement in hand. We discuss the creation of these clusters in the following sections:

6.2.1 Semantic/Functionality based clustering

We leverage semantic clustering to group files based on their functional similarity, to identify functionally cohesive file clusters that are expected to share similar logging behaviors.

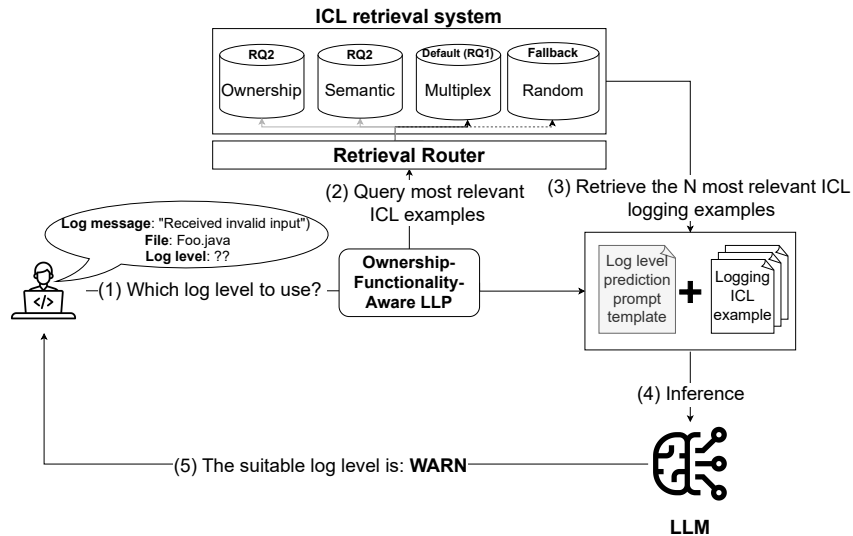


Figure 6.2: Overview of our Ownership-Functionality Aware retrieval approach for logging level prediction.

Raw data construction: To build our semantic-based clusters we follow the approach summarized in Figure 6.3. First, we create a transformer-based embedding for the files of each project. Such embeddings are dense vector representations obtained by encoding textual inputs (e.g., source code) using transformer models. These models are extensively pre-trained on massive amounts of text and source code data, enabling them to capture semantic patterns within their inputs. Encoder-only models are particularly effective for embedding tasks as their training encourages the model to learn representations that take into account both syntax and semantics (Chochlov et al., 2022; Feng et al., 2020; Liu et al., 2024).

For the purpose of our study, we chose the encoder-only transformer CodeX-Embed (Liu et al., 2024) because it provides state-of-the-art performance on code-related embedding tasks, supports input sequences of up to 32K tokens - thus effectively capturing the full context of most source files in our studied projects -

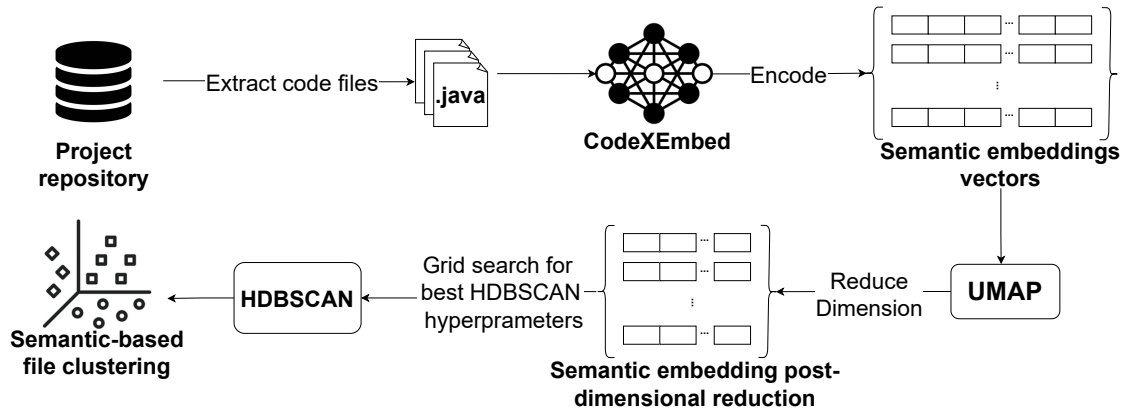


Figure 6.3: Overview of the semantic-based clustering

and achieves these capabilities despite its modest size (2.6B parameters), making it computationally efficient and practical.

After the encoding, each source file is represented as a 2304-dimensional embedding vector (default dimension for CodeXEmbed embeddings) capturing the file’s semantic information. For exceptionally large files (less than 5% across projects) exceeding the 32K token window, we partition them at top-level method boundaries, individually encode each method, and then aggregate the resulting vectors via mean-pooling to produce a single representative embedding for the entire file (Günther et al., 2024).

As direct clustering on high-dimensional embeddings is computationally expensive and sensitive to noise, we first apply UMAP (Uniform Manifold Approximation and Projection) (McInnes et al., 2020) to project the 2,304-dimensional file embeddings into a 50-dimensional space, similarly to prior work (Baligodugula and Amsaad, 2025). UMAP preserves local neighborhood structure while substantially reducing the computational cost of the clustering.

We then apply the density-based clustering algorithm HDBSCAN (Malzer and Baum, 2020) to the reduced embeddings. HDBSCAN is well suited to our setting because it can discover clusters of varying densities (i.e., handling both sparse and compact clusters) and explicitly label low-density points as noise, which is important given the heterogeneous nature of source files in large projects. To obtain robust and high-quality clusters, we perform a grid search over the two main HDBSCAN hyperparameters for each project:

- The minimum cluster size (mcs): For a project with N files, we explore $mcs \in \{N/300, N/150, 25, 40, 50, 75, 100, N/10\}$, which spans very small to relatively large cluster granularities
- The minimum number of samples (ms): For each mcs value, we set $ms \in \{0.5 \times mcs, 0.25 \times mcs\}$.

For every (mcs, ms) configuration we compute Silhouette and Davies–Bouldin indices and select the configuration that maximizes the Silhouette score among those with Davies–Bouldin index below 1.0, thereby favouring compact, well-separated clusters. Finally, we assess the stability of the selected clustering (i.e., the best mcs and ms) through bootstrap resampling (30 iterations). Specifically, we generate 30 different bootstrap samples of the files, re-clustered each sample using the selected best configuration, and compare the resulting partitions to the original clustering. We report the distribution of Adjusted Rand Index (ARI) values across bootstrap samples, to demonstrate that the cluster structures remain consistent even when the data is perturbed.

6.2.2 Ownership clustering

We perform ownership clustering to group files according to their shared developer contributions, based on the intuition that files maintained by the same authors might exhibit similar logging conventions.

Raw data construction: To construct our ownership-based clusters, we mine the Git history of each studied project and build an author–file *ownership matrix* (Figure 6.4). Rows correspond to source files and columns to authors and each cell records how frequently an author has modified a file, with commit counts weighted by an exponential time-decay factor so that recent edits contribute more heavily than older ones. This representation captures both long-term ownership and recent maintenance activity in a single high-dimensional vector for each file.

Build a file-to-file graph: We treat each row of this matrix as a file-level ownership vector and compute pairwise cosine similarities between files. Using these similarities, we construct a weighted file–file graph in which edges connect files that share similar maintainer sets. To avoid an overly dense graph while still preserving strong ownership signals, we connect each file only to its k nearest neighbours by cosine similarity, with $k = 20$. We selected $k = 20$ empirically to balance graph connectivity with local specificity. In our sensitivity analysis (i.e., trying different k values), lower values (e.g., $k = 10$) resulted in excessive fragmentation (isolating too many files), while higher values (e.g., $k = 30$) introduced noise, merging distinct functional groups. $k = 20$ consistently produced the most coherent community structures.

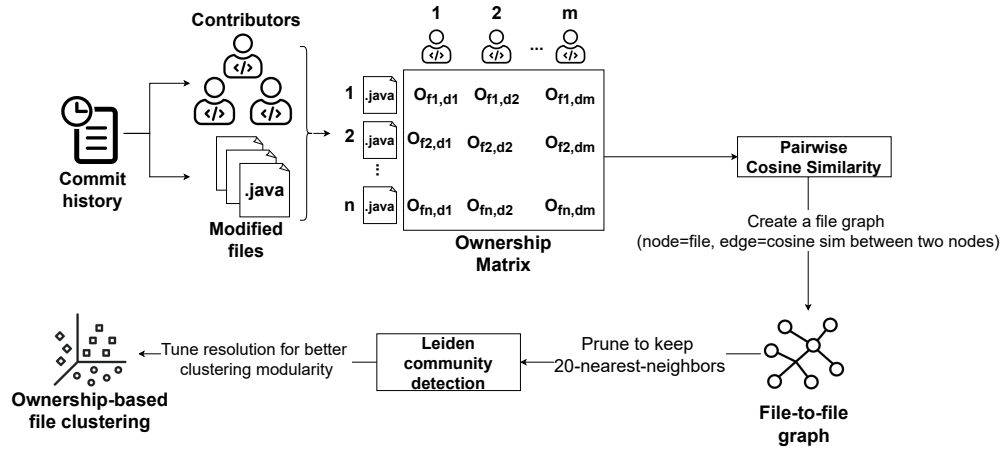


Figure 6.4: Overview of the ownership-based clustering

Build community-based clusters: We then apply the weighted Leiden community detection algorithm (Traag et al., 2019) to this k -NN graph to obtain ownership communities. Leiden optimizes modularity while avoiding the disconnected communities sometimes produced by the earlier Louvain algorithm (Blondel et al., 2008), and scales well to our graph sizes. We tune the resolution parameter and retain settings that yield modularity above 0.7 for all projects, which indicates a pronounced community structure in the ownership graph (Fortunato and Barthelemy, 2007; Traag et al., 2019).

6.2.3 Multiplex clustering

We further investigate the combination of these two socio-technical signals into a unified multiplex graph, as shown in Figure 6.5. A multiplex graph is a multi-layered network structure where the same set of nodes exists (i.e., files) across all

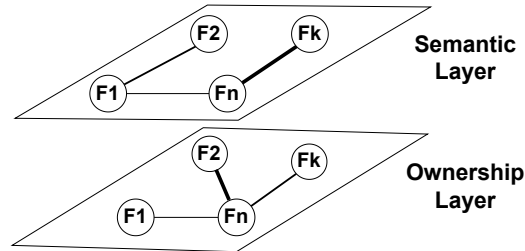


Figure 6.5: Overview of multiplex graphs. Nodes are Java files and edges are labeled with cosine similarity scores between the vectors representing the connected files.

layers, but each layer represents a distinct type of relationship between them (Battiston et al., 2014; De Domenico et al., 2013). Specifically, we construct a two-layer multiplex graph, where the first layer encodes *semantic similarity* between files based on code embeddings (computed using cosine similarity between file embeddings) and the second layer represents *file ownership* file-to-file graph (section 6.2.2) .

Both layers have the same set of nodes (each node corresponding to a Java source file) but maintain distinct edge sets, preserving the unique insights offered by each type of signal. To identify cohesive clusters across these complementary layers, we apply the multiplex variant of the Leiden community detection algorithm (Traag et al., 2019). This multiplex algorithm simultaneously optimizes modularity across both semantic and ownership layers, identifying new communities where files share not only similar functionality but also similar maintaining developers.

At inference time, for a given logging statement, our approach identifies the multiplex cluster associated with the source file containing the logging statement,

then it retrieves the top k candidate examples from within this cluster (with $k = 5$ in our experiments), ranking candidates by a combined similarity score

$$s(c) = \alpha \cdot \text{cos}_{\text{sem}}(f^*, c) + (1 - \alpha) \cdot \text{cos}_{\text{own}}(f^*, c)$$

where:

- $s(c)$: is the combined similarity score for candidate example c .
- f^* : is the file containing the logging level to be predicted.
- $\text{cos}_{\text{sem}}(f^*, c)$: is the semantic cosine similarity between the embeddings of f^* and candidate c .
- $\text{cos}_{\text{own}}(f^*, c)$ is the ownership cosine similarity between the embeddings of f^* and candidate c .
- α : Weighting coefficient that controls the relative influence of semantic similarity versus ownership similarity in the combined retrieval score (tunable with our approach).

We determined the optimal layer weighting α via a grid search on a held-out validation set (a subset of the retrieval data). We evaluated α in increments of 0.1 and selected $\alpha = 0.7$, as it maximized the downstream logging level prediction AUC score. This value was then fixed for all subsequent experiments on the test sets. In this configuration, our approach gives more influence to semantic similarity while still incorporating ownership information. This choice aligns with our design assumption that functional (semantic) similarity should be the primary retrieval signal, with ownership acting as a secondary refinement. That said α remains a

configurable parameter and can be adjusted by practitioners if different trade-offs between semantic and ownership signals are desired.

If we cannot identify a suitable multiplex cluster or if the cluster does not contain enough logging examples, based on the input logging message and file, our approach automatically falls back to in order to:

1. Semantic/functionality cluster retrieval.
2. Ownership community retrieval.
3. Random-retrieval from the entire project.

6.2.4 Prompting LLMs for logging level prediction

To create the logging level prediction prompt we leverage the approach explained in Figure 6.6. Specifically, we first parse the code to isolate the target logging instruction (e.g., `logger.warn(...)`) from its surrounding method. The process involves three key steps:

- **Extraction:** We separate the log message (the text identifying the event) and the ground truth logging level from the source code.
- **Masking:** We extract the surrounding code context to provide the model with syntactic cues. Crucially, we mask the actual logging statement within this context. This prevents data leakage and forces the model to infer the level based on the logic of the code rather than simply reading the existing statement (e.g., `if (logger.isDebugEnabled())`).

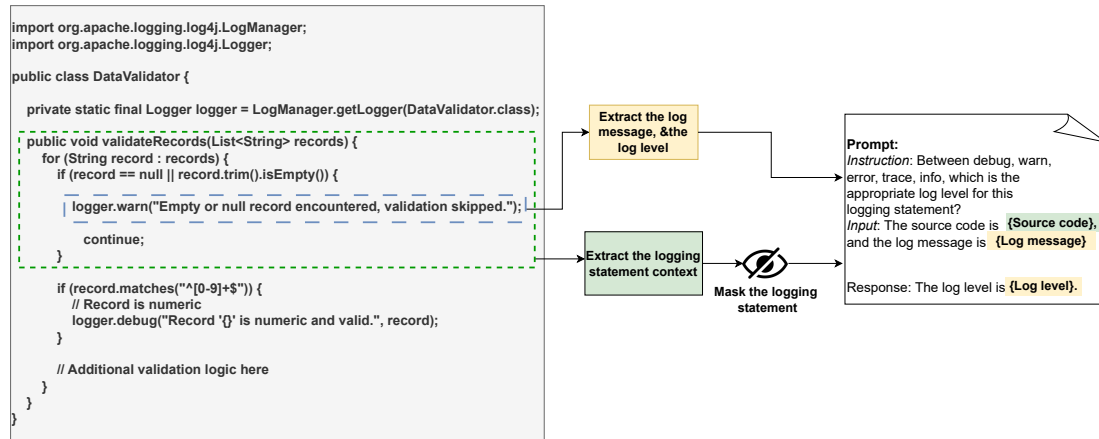


Figure 6.6: Prompt creation process for the LLM-LLP.

- **Prompt Assembly:** The final prompt is constructed by combining a specific task instruction (defining the valid output format), the masked source code context, and the isolated log message.

For our in-context-learning prompts, we also add the k logging examples extracted from the pertinent multiplex clusters to the constructed prompt.

For the purposes of this study, we chose four models CodeLlama-7B-Instruct, CodeLlama-34b-Instruct, Deepseek-V3.2, and GPT-4o, as shown in Table 6.1. We specifically chose the CodeLlama family (7B and 34B) to isolate the impact of model size on performance within a single, open-source architecture. The 7B version serves as a baseline for consumer-grade hardware, while the 34B version represents the upper limit of what can be deployed on a single consumer-grade GPU node (without quantization). To benchmark these local models against industry standards, we selected GPT-4o as a high-performance commercial baseline, as it is one of the few leading frontier models that provides API access to token log probabilities, which are necessary for our evaluation metrics. Finally, we included

Table 6.1: Summary of Large Language Models Selected for Evaluation

Model	Provider	Architecture	Parameters (Total / Active)	Context (Window)	Access
<i>Open-Weight Models</i>					
CodeLlama-7B-Instruct	Meta	Dense Decoder	7B / 7B	16k	Local
CodeLlama-34b-Instruct	Meta	Dense Decoder	34B / 34B	16k	Local
DeepSeek-V3.2	DeepSeek	MoE (Sparse Attn)	685B / 37B	164k	API
<i>Proprietary Models</i>					
GPT-4o	OpenAI	Undisclosed*	Unknown	128k	API

Note: For proprietary models, exact architectural details and parameter counts are not public. *While technically undisclosed, community analysis suggests GPT-4o utilizes a Mixture-of-Experts (MoE) architecture.

DeepSeek-V3.2 to serve as a cost-efficiency benchmark, allowing us to compare our self-hosted approach against a high-performance but low-cost commercial API.

6.2.5 Implementation details

To validate the practical feasibility of our approach, we measured the execution time of each clustering phase using a representative workstation. This evaluation aims to demonstrate that the socio-technical analysis can be integrated into standard development workflows without imposing prohibitive overhead. In fact, all our presented clustering approaches are computationally efficient, ensuring practical applicability for cluster refreshing if/when needed. Specifically, ownership-based clustering completes in a few minutes, even for our largest studied project (Elasticsearch). Semantic-based clustering takes slightly longer due to the initial step of computing embeddings using a transformer-based tokenizer. However, it remains practical, taking approximately 20 minutes for Elasticsearch (the largest project

studied) on standard hardware (16-core CPU, 64 GB RAM, 24 GB GPU memory). Finally, once the artifacts from the prior two clusterings (i.e., ownership matrix and semantic embeddings) are created, multiplex clustering takes few minutes for all of our projects.

6.3 Empirical evaluation setup

We empirically evaluate the effectiveness of our approach (with its different operation modes) across four widely used open-source Java projects: Hadoop, HBase, Elasticsearch, and Cassandra. These projects were selected due to their varied size, distinct logging practices, and differing clarity in documentation-based component definitions. Specifically, Hadoop and HBase feature explicit documentation-defined components, allowing comparison against a documentation-based retrieval baseline. Conversely, Elasticsearch and Cassandra lack clearly defined component boundaries, presenting a scenario ideal for assessing the robustness and generalizability of our approach.

6.3.1 Data collection

We collected the datasets used in this chapter by extracting source code files, logging statements, and associated metadata directly from each project’s GitHub repository. Specifically, we cloned each repository and checked out the latest stable release versions (e.g., Cassandra-4.1.9) to ensure reproducibility. To extract logging statements systematically, we parsed Java source files using the *javalang*² library,

²<https://pypi.org/project/javalang/>

Table 6.2: Summary of the studied projects

Project	Versions	# SLOC	# Contributors	# Logging statements
Hadoop	3.3.6	1.8M	635	21,910
HBase	2.5.9	841K	483	10,449
Elasticsearch	8.9.2	2.5M	2,042	25,187
Cassandra	4.1.9	595K	464	4,808

identifying log invocations and capturing relevant context such as the log message, verbosity level, method boundaries, and surrounding code snippets. Metadata such as file paths, line numbers, and documentation-based components (if any) were also recorded. In cases where a project lacked explicit documentation-based component definitions (e.g., Cassandra), we labeled the component information as *unknown*. Table 6.2 summarizes the characteristics of each studied project.

Table 6.2 summarizes key characteristics of each of our studied projects.

6.3.2 Evaluation baselines

To evaluate the performance of our approach, we consider four baselines: The two retrieval approaches used by prior studies (Heng et al., 2024) and two machine learning specialized logging level prediction models (i.e., the ordinal logistic regression proposed by Li et al. (2017a) and the deep learning model proposed by Li et al. (2021b)):

- **Random retrieval (diversity-first)** (Heng et al., 2024): Logging examples for in-context-learning are randomly selected from the entire dataset of the studied project.

- **Documentation-based retrieval:** For projects explicitly defining components in their documentation (e.g., HDFS vs. Yarn in Hadoop), logging examples for in-context-learning are exclusively retrieved from within the same documented component (e.g., for a file in Yarn, examples are retrieved only from other Yarn files). This approach relies solely on predefined documentation-based boundaries without employing clustering and is only applicable when such explicit documentation is available. We adopt this retrieval strategy due to its algorithmic simplicity and demonstrated efficacy in enhancing supervised logging level prediction, as established in our prior work ([Ouatiti et al., 2023](#)).
- **Ordinal Logistic Regression (OLR)** ([Li et al., 2017a](#)): A machine learning model that treats logging level prediction as an ordinal classification problem. OLR uses hand-crafted features derived from (i) the logging statement itself (e.g., log message length), (ii) change history (e.g., log churn), (iii) file-level characteristics (e.g., average logging level within the file), and (iv) project history (e.g., number of revisions touching the file).
- **DeepLV** ([Li et al., 2021b](#)): A deep-learning approach that formulates logging level prediction as an ordinal classification problem. DeepLV extracts syntactic context features from the AST (from the start of the enclosing method up to the basic block containing the logging statement) and log message tokens, represents them as sequences, and feeds them through an embedding layer and a Bi-LSTM network.

The evaluation of our approach against all baselines begins by ordering each project’s logging statements chronologically and splitting them into a retrieval/-training portion and a test portion (70% retrieval/training, 30% test), similar to prior work (Ouatiti et al., 2024). For retrieval-based approaches (random retrieval, documentation-based retrieval, and our ownership-functionality aware retrieval), the earlier 70% of the data is used as the pool from which in-context examples are drawn, and the later 30% is used exclusively for testing. All retrieval strategies share the same underlying LLM (e.g., CodeLlama-7B) and prompting protocol and only the mechanism for selecting in-context-learning examples differs. The specialized supervised models (i.e., OLR and DeepLV) are trained on the same 70% retrieval/training portion (using only training labels) and evaluated on the same 30% test set as the retrieval-based methods.

To robustly assess predictive performance, we evaluate each method using five bootstrap samples drawn from the test dataset. Because all models produce predictions for the same set of logging statements, we use paired Wilcoxon signed-rank tests to assess whether observed performance differences are statistically significant. We also report Cohen’s d to quantify effect size, interpreting $0.2 \leq d < 0.5$ as a small effect, $0.5 \leq d < 0.8$ as medium, and $d \geq 0.8$ as large (Cohen, 1992).

6.3.3 Evaluation metrics

As our approach integrates two modeling tasks (i.e., clustering and logging level prediction), we evaluate the effectiveness of our method using two distinct sets of robust metrics: clustering quality metrics and logging level prediction performance metrics.

Clustering quality metrics

- Adjusted Rand Index ³ (ARI): quantifies the agreement or similarity between two clustering solutions. ARI ranges between -1 and 1, where values close to 1 indicate strong agreement, values close to 0 indicate random clustering, and negative values indicate worse-than-random clustering (Steinley, 2004).
- Silhouette score ⁴: measures how tightly grouped (i.e., cohesive) points within clusters are compared to points in other clusters. It ranges from -1 to 1, with values above 0.5 indicate strong, cohesive, and well-separated clusters; values between 0.25 and 0.5 reflect moderate clustering structure; and values below 0.25 suggest weak or ambiguous clustering (Rousseeuw, 1987).
- Davies–Bouldin index (Davies and Bouldin, 1979) (DBI): measures how well clusters are separated by comparing each cluster’s internal dispersion with its nearest-neighbour distance. It ranges from 0 to ∞ , where lower values are better. Typically, DBI scores below 1 indicate well-separated and compact clusters (Davies and Bouldin, 1979).
- Density-Based Clustering Validation (DBCVC) (Moulavi et al., 2014): a density-aware validation metric tailored for algorithms such as HDBSCAN. DBCVC combines cluster compactness (how tightly points are packed around local density modes) with cluster isolation (how deep the density valleys are between neighbouring clusters). Values range from -1 to $+1$, with positive scores indicating well-formed, well-separated density regions, values near 0 suggesting

³https://scikit-learn.org/stable/modules/generated/sklearn.metrics.adjusted_rand_score.html

⁴https://scikit-learn.org/stable/modules/generated/sklearn.metrics.silhouette_score.html

structures that are indistinguishable from noise, and negative scores indicating badly fragmented or overlapping clusters.

- Leiden modularity (Traag et al., 2019): assesses how a graph can be partitioned into distinct clusters by providing insight into the overall strength and quality of cluster boundaries. High modularity indicates that nodes within clusters have dense internal connections and relatively sparse connections to nodes in other clusters. Values above 0.3 typically indicate a strong and meaningful community structure (Fortunato and Barthelemy, 2007; Traag et al., 2019).

logging level prediction performance metrics

- **Area Under the ROC Curve (AUC):** quantifies the ability of the model to discriminate between logging level classes at various threshold settings. To adapt this metric for generative LLMs, we do not rely solely on the discrete text output. Instead, we extract the conditional probabilities of the first generated token corresponding to our target labels (i.e., $\mathcal{V}_{target} = \{\text{DEBUG, INFO, ...}\}$). We normalize these token probabilities to sum to 1 over the candidate set of logging levels, effectively treating the LLM as a probabilistic classifier. Following prior work (Heng et al., 2024; Li et al., 2017a, 2021b; Ouatiti et al., 2023), we then compute the multiclass AUC using the One-vs-Rest scheme as implemented in scikit-learn⁵.

⁵https://scikit-learn.org/stable/modules/generated/sklearn.metrics.roc_auc_score.html

- **Precision:** measures the proportion of predicted logging levels that are correct. For the multi-class setting, we compute precision per logging level and report the macro-averaged precision across all levels.
- **Average Ordinal Distance Score (AOD):** captures how close predicted logging levels are to their true ordinal positions (Heng et al., 2024; Li et al., 2021b). Each logging level is mapped to an ordinal value (e.g., DEBUG < INFO < WARN < ERROR < FATAL), and AOD normalizes the distance between the predicted and true levels to the $[0, 1]$ interval, where higher values indicate predictions that are, on average, closer to the correct logging levels. We report AOD as it provides a more nuanced assessment of ordinal prediction quality than precision or AUC alone.

6.4 Results

In this section, we present the results for each of our RQs. For each RQ, we discuss the motivation, the approach we used to address the RQ, and our findings.

RQ1: How does the integration of ownership and functionality signals via multiplex clustering impact the predictive performance of LLM-based LLPs?

Motivation: The goal of this research question is to understand the impact of leveraging both semantic similarity and developer ownership signals simultaneously to

retrieve ICL logging examples for logging level prediction scenarios. While semantic similarity captures the functional context (i.e., what is being logged), developer ownership captures the human aspect (i.e., who is deciding). Prior research has established that logging level selection is influenced by individual developer habits and team norms (Li et al., 2017a; Yuan et al., 2010). Consequently, retrieving examples based solely on code similarity may yield technically relevant but stylistically inconsistent in-context-learning examples. By integrating the ownership signal with the semantic signal, we align the examples not only with the code’s logic but also with the specific logging conventions of the contributor.

Approach: To quantify the impact of multiplex based retrieval (taking into consideration ownership and semantic knowledge simultaneously) on the performance of LLM-powered LLPs, we first perform the multiplex clustering for each studied project, as described in Section 6.2.3. Then, we evaluate the performance of our approach using in-context-learning examples from the relevant multiplex clusters. We compare this approach to three baselines: (a) random retrieval from the entire project, (b) Ordinal logistic regression LLP (Li et al., 2017a) and DeepLV (BiLSTM) (Li et al., 2021b).

Results: Multiplex retrieval significantly boosts the performance of LLM-powered LLPs, achieving both high accuracy and low retrieval latency across all studied projects, as shown in Table 6.3. In fact, leveraging our ownership-functionality aware approach to retrieve in-context-learning logging examples from multiplex clusters results in a statistically significant (Wilcoxon test, $\alpha = 0.01$, Cohen’s d , $d > 0.7$) improvement in AUC ranging from 0.907 (observed for Cassandra) to 0.966 (observed for Elasticsearch), outperforming the global random retrieval

Table 6.3: Summary of the performance of the multiplex retrieval compared to other approaches (metrics are obtained from 5 bootstrap runs and for CodeLlama-7B)

Model	Metric	Hadoop	HBase	Elasticsearch	Cassandra
Multiplex retrieval (5)	AUC	0.921 ± 0.005	0.914 ± 0.004	0.966 ± 0.002	0.907 ± 0.003
	Precision	0.694 ± 0.006	0.663 ± 0.015	0.764 ± 0.005	0.668 ± 0.017
	AOD	0.883 ± 0.004	0.868 ± 0.007	0.912 ± 0.002	0.877 ± 0.002
Few-random shot (5) (Heng et al., 2024)	AUC	0.810 ± 0.002	0.785 ± 0.005	0.801 ± 0.003	0.786 ± 0.017
	Precision	0.523 ± 0.003	0.435 ± 0.005	0.513 ± 0.016	0.443 ± 0.014
	AOD	0.818 ± 0.003	0.782 ± 0.003	0.829 ± 0.009	0.790 ± 0.008
Ordinal Log. Regression (Li et al., 2017a)	AUC	0.731 ± 0.011	0.729 ± 0.004	0.766 ± 0.004	0.710 ± 0.019
	Precision	0.359 ± 0.010	0.367 ± 0.017	0.329 ± 0.012	0.371 ± 0.018
	AOD	0.637 ± 0.007	0.621 ± 0.007	0.766 ± 0.004	0.619 ± 0.012
DeepLV (Li et al., 2021b)	AUC	0.815 ± 0.003	0.778 ± 0.012	0.868 ± 0.005	0.762 ± 0.007
	Precision	0.508 ± 0.008	0.508 ± 0.008	0.553 ± 0.006	0.482 ± 0.011
	AOD	0.790 ± 0.004	0.752 ± 0.011	0.807 ± 0.009	0.757 ± 0.007

baseline by 0.11 to 0.16. Similarly, precision and AOD exhibit similar statistically significant (Wilcoxon test, $\alpha = 0.01$, Cohen’s d , $d > 0.7$) improvements as the median Precision and AOD improves by 0.17 to 0.23 and 0.06 to 0.08 respectively. Furthermore, our retrieval step leverages FAISS ⁶ making it highly efficient, with a median latency of 8.9 ms per retrieval, effectively operating at real-time speed (i.e., suitable for IDEs). This ensures that the logging level prediction bottleneck (if any) remains with the LLM’s generation step rather than with the ICL example retrieval. Furthermore, our approach statistically significantly (Wilcoxon test, $\alpha = 0.01$, Cohen’s d , $d > 0.7$) outperforms the established baselines for logging level prediction (Li et al., 2017a, 2021b). Finally, our results are consistent for the models we evaluated.

Across our evaluated projects, most mispredictions made by our retrieval consistently involve confusion between adjacent logging levels and concentrate in a small subset of files, as shown in Table 6.4. Specifically, we observe that

⁶<https://github.com/facebookresearch/faiss>

Table 6.4: Overview of the mispredictions using OFA-LLP (multiplex mode)

Project	Adjacent Level Confusion	Noise Cluster Membership	Files Covering 50% of Mispredictions
Hadoop	72.8%	41.2%	14.7%
Cassandra	64.1%	56.4%	13.3%
HBase	67.0%	54.5%	13.9%
Elasticsearch	64.7%	39.4%	16.5%

adjacent-level mispredictions (e.g., confusing DEBUG with INFO, or WARN with ERROR) account for 64% (Cassandra), to 72% (Hadoop) of total mispredictions. Furthermore, a significant portion of mispredictions occurs in files that our multiplex clustering marked as noise (i.e., unclustered files), representing between 39% (Elasticsearch) and 56% (Cassandra) of total errors. Finally, mispredictions are heavily concentrated at the file level, with fewer than 17% of files accounting for at least half of all mispredictions in each project. This pattern suggests that predictive difficulties faced by our approach are highly localized, providing clear opportunities for targeted improvements.

Summary of RQ1

Multiplex clustering, which simultaneously integrates semantic and ownership signals, provides a retrieval strategy that significantly boosts the AUC of LLM-powered LLPs (up to 0.23 improvement over random retrieval). We recommend using our ownership-functionality aware approach for real-time retrieval of high-quality ICL examples supporting logging level prediction.

RQ2: What is the individual contribution of functionality and ownership signals?

Motivation: The goal of this research question is to disentangle and quantify the individual contributions of semantic similarity and developer ownership signals within our retrieval approach. While both signals offer theoretical utility (semantic similarity capturing the functional context of the code, and ownership capturing the human aspect of the decision making), their comparative effectiveness remains unclear. By isolating these components, we aim to empirically determine whether these signals serve as complementary information channels or if one dimension dominates the predictive capability in logging level prediction scenarios.

Approach: To disentangle and quantify the individual contributions of semantic similarity and developer ownership signals, we conduct an ablation study by isolating the retrieval mechanisms. Specifically, we instantiate two distinct baselines: (a) Semantic-only retrieval, where examples are selected solely based on vector embedding similarity (as detailed in Section 6.2.1), and (b) Ownership-only retrieval, where examples are selected exclusively from the developer’s collaboration network (as detailed in Section 6.2.2). We then evaluate the performance of the LLMs when prompted with examples from these isolated layers and compare them against our integrated ownership-functionality aware approach presented in RQ1. We primarily report the findings for the CodeLlama-7B model in the main text and refer to the Table 6.9 for the results obtained using CodeLlama-34B.

Results: Our ownership-based clustering approach produces cohesive and stable clusters across all the studied projects, as shown in Table 6.5. In fact, our ownership-based clusters achieve consistently high Leiden modularity scores

Table 6.5: Ownership-based clustering quality metrics

Project	Modularity (Q_c)	Median ARI	Inter/Intra Cosine Similarity	Coverage
Hadoop	0.798	0.524	0.851 / 0.123	99.0%
HBase	0.805	0.286	0.618 / 0.018	67.0%
Elasticsearch	0.850	0.556	0.720 / 0.268	99.5%
Cassandra	0.919	0.588	0.651 / 0.093	96.0%

($Q > 0.79$), with Elasticsearch reaching 0.92. Additionally, intra-cluster cosine similarities (ranging from 0.62 to 0.85) significantly exceed inter-cluster similarities (0.02–0.27), reinforcing the internal cohesion of the identified clusters. Furthermore, the stability analysis of our clustering via the Adjusted Rand Index (ARI) across 15 two-month windows indicates a moderate to high cluster consistency (median ARI between 0.29 and 0.59). Although HBase exhibits relatively low stability (ARI of 0.29), its high modularity (0.81) suggests that while ownership may shift frequently, recalculating clusters periodically (e.g., monthly) would effectively sustain cohesive and useful clusters.

Leveraging ownership-based retrieval significantly enhances the predictive performance across all projects, as shown in Table 6.6. Specifically, providing the LLM used for logging level prediction with five ownership-based ICL examples statistically significantly (Wilcoxon test, $\alpha = 0.01$, Cohen’s d , $d > 0.8$) improves the median AUC of that LLM from a median 0.81 (observed when using random retrieval) to between 0.83 (Hadoop) and 0.90 (Elasticsearch), marking a median improvement of 2.1%. Moreover, ownership-based retrieval outperforms (Wilcoxon test, $\alpha = 0.01$, Cohen’s d , $d > 0.7$) retrieval based on documentation-defined component in one of the two projects for which explicit components are defined in the project’s documentation. The improvements brought by ownership-based retrieval extend

Table 6.6: Summary of the performance of the ownership-only retrieval compared to other approaches (metrics are obtained from 5 bootstrap runs and for CodeLlama-7B)

Model	Metric	Hadoop	HBase	Elasticsearch	Cassandra
Ownership-based (5)	AUC	0.832 ± 0.002	0.808 ± 0.002	0.904 ± 0.001	0.837 ± 0.007
	Precision	0.532 ± 0.002	0.510 ± 0.008	0.578 ± 0.006	0.501 ± 0.019
	AOD	0.819 ± 0.001	0.808 ± 0.002	0.845 ± 0.001	0.824 ± 0.008
Zero-shot	AUC	0.694 ± 0.002	0.677 ± 0.002	0.732 ± 0.003	0.675 ± 0.010
	Precision	0.342 ± 0.001	0.332 ± 0.023	0.400 ± 0.002	0.339 ± 0.010
	AOD	0.729 ± 0.002	0.732 ± 0.011	0.751 ± 0.001	0.723 ± 0.006
Few-shot (5)	AUC	0.802 ± 0.004	0.805 ± 0.000	0.844 ± 0.010	0.825 ± 0.007
	Precision	0.500 ± 0.008	0.502 ± 0.011	0.496 ± 0.003	0.481 ± 0.017
	AOD	0.806 ± 0.002	0.805 ± 0.003	0.814 ± 0.001	0.810 ± 0.008
Documentation Few-shot (5)	AUC	0.808 ± 0.004	0.813 ± 0.005	-	-
	Precision	0.504 ± 0.004	0.483 ± 0.007	-	-
	AOD	0.806 ± 0.003	0.800 ± 0.001	-	-

to precision and AOD scores, for which we report improvements from 1% to 8% (Precision) and from 0.3% to 3% (AOD). Our reported scores account for fallback instances where logging level predictions are made for files not assigned to any ownership clusters. However, the fallback rate remains minimal (below 5%) across all studied projects except for HBase, which experiences a higher fallback rate of 33%. This higher fallback rate indicates that many logging statements fall into files that either belong to small clusters (discarded as noise due to our minimum cluster size constraint of 10 files) or clusters lacking sufficient examples for complete ownership-only retrieval. Consequently, we observe the lowest performance improvement (0.3%) among our projects for HBase when using the ownership-only retrieval mode.

Table 6.7: Semantic-based clustering quality metrics

Project	Files	Best Param. (mcs / ms)	# Clusters	Silh.	DBCV	Noise	Bootstr. ARI
Hadoop	7,395	50 / 25	31	0.533	0.230	49%	0.685 ± 0.027
HBase	2,691	25 / 12	27	0.566	0.325	41%	0.690 ± 0.040
Elasticsearch	14,976	100 / 25	31	0.553	0.219	45%	0.670 ± 0.050
Cassandra	3,158	25 / 12	30	0.606	0.347	37%	0.700 ± 0.030

Leveraging file ownership information enhances retrieval quality for LLM-powered LLPs, as ownership-only retrieval achieves a statistically significant improvement in AUC performance, ranging between 0.3% and 6%, compared to random retrieval.

Our semantic-only based clustering yields compact, cohesive, and stable clusters, as shown in Table 6.7. In fact, our clustering approach consistently identifies between 27 and 31 semantically aligned clusters, across our studied projects. These clusters achieve a strong internal cohesion as the median Silhouette score ranges from 0.53 to 0.6 (more than 0.5 is cohesive) and the DBCV between 0.23 and 0.35 (less than 1 is compact). Furthermore, stability of our clusters is high, as demonstrated by the bootstrapped ARI scores ranging from 0.67 (Elasticsearch) to 0.7 (Cassandra). That said, we observe that HDBSCAN marks 37% (Cassandra) to 49% (Hadoop) of the files as noise, potentially leading to fallback scenarios during retrieval (i.e., if the logging level of the logging statement to be predicted is in a file labeled as noise).

Despite the relatively high ratio of files labeled as noise, leveraging semantic-based clusters significantly enhances LLM-powered LLPs compared to our baselines, as shown in Table 6.8. Indeed, using semantic-only based retrieval to provide

Table 6.8: Summary of the performance of the semantic-only retrieval compared to other approaches (metrics are obtained from 5 bootstrap runs and for CodeLlama-7B)

Model	Metric	Hadoop	HBase	Elasticsearch	Cassandra
Semantic-based (5)	AUC	0.865 ± 0.002	0.863 ± 0.003	0.924 ± 0.002	0.863 ± 0.006
	Precision	0.587 ± 0.002	0.568 ± 0.007	0.638 ± 0.008	0.652 ± 0.020
	AOD	0.837 ± 0.003	0.838 ± 0.004	0.863 ± 0.001	0.868 ± 0.005
Zero-shot	AUC	0.694 ± 0.002	0.677 ± 0.002	0.732 ± 0.003	0.675 ± 0.010
	Precision	0.342 ± 0.001	0.332 ± 0.023	0.400 ± 0.002	0.339 ± 0.010
	AOD	0.729 ± 0.002	0.732 ± 0.011	0.751 ± 0.001	0.723 ± 0.006
Few-shot (5)	AUC	0.802 ± 0.004	0.805 ± 0.000	0.844 ± 0.010	0.825 ± 0.007
	Precision	0.500 ± 0.008	0.502 ± 0.011	0.496 ± 0.003	0.481 ± 0.017
	AOD	0.806 ± 0.002	0.805 ± 0.003	0.814 ± 0.001	0.810 ± 0.008
Documentation-based Few-shot (5)	AUC	0.808 ± 0.004	0.813 ± 0.005	-	-
	Precision	0.504 ± 0.004	0.483 ± 0.007	-	-
	AOD	0.806 ± 0.003	0.800 ± 0.001	-	-

in-context-learning logging examples to our LLM-powered LLPs statistically significantly improves the median AUC scores by 4% to 6% compared to the documentation based in-context-learning retrieval (if any), and by 2% (Elasticsearch) to 6% (Hbase) compared to ownership-based retrieval. The performance improvement brought by semantic-based retrieval extends to both the precision and AOD scores, across all our studied projects.

Despite labeling a large proportion of files as noise (up to 49%), semantic-based retrieval statistically significantly enhances the predictive performance of LLM-powered LLPs, improving AUC by up to 6.5% compared to random and documentation-based retrieval. This indicates that code semantic clusters –even with partial coverage– offer a strong and practical value as a retrieval signal for LLM-powered logging level prediction.

Table 6.9: Summary of the performance of the semantic-only retrieval compared to other approaches (metrics are obtained from 5 bootstrap runs and for CodeLlama-34BB)

Retrieval	Metric	Hadoop	HBase	Elastic.	Cassandra
Semantic	AUC	0.875 ± 0.001	0.871 ± 0.005	0.889 ± 0.037	0.859 ± 0.009
	Precision	0.616 ± 0.008	0.580 ± 0.012	0.644 ± 0.004	0.569 ± 0.026
	AOD	0.850 ± 0.004	0.838 ± 0.007	0.865 ± 0.001	0.844 ± 0.011
Ownership	AUC	0.824 ± 0.002	0.812 ± 0.006	0.905 ± 0.001	0.838 ± 0.012
	Precision	0.544 ± 0.004	0.508 ± 0.018	0.582 ± 0.005	0.504 ± 0.020
	AOD	0.822 ± 0.001	0.809 ± 0.007	0.843 ± 0.003	0.821 ± 0.010
Multiplex	AUC	0.932 ± 0.006	0.926 ± 0.003	0.969 ± 0.001	0.914 ± 0.002
	Precision	0.727 ± 0.008	0.694 ± 0.008	0.775 ± 0.005	0.693 ± 0.003
	AOD	0.894 ± 0.005	0.880 ± 0.003	0.913 ± 0.001	0.884 ± 0.003

RQ3: Can Multiplex Retrieval reduce the computational cost of LLM-based logging level prediction?

Motivation: The goal of this research question is to investigate the trade-off between predictive performance and operational efficiency in LLM-powered logging level predictors. While commercial, large-scale Large Language Models (e.g., GPT-4o, DeepSeek-V3.2) have established new state-of-the-art benchmarks in code intelligence tasks, their deployment in high-volume software engineering workflows remains constrained by prohibitive inference costs and latency. In fact, logging level prediction can be frequently invoked within Continuous Integration (CI) pipelines, where analyzing a large amount of logging statements (along with their context) per commit using a commercial model would incur a large economic overhead. We hypothesize that by enhancing the retrieval mechanism, specifically through our multiplex strategy, we can empower smaller, open-weight models (e.g., 7B parameters) to achieve performance parity with substantially larger commercial models.

Approach: To validate the cost-effectiveness of our approach, we conduct a comparative cost-benefit analysis between our multiplex-enhanced CodeLlama-7B model and two leading commercial "black-box" models: GPT-4o (OpenAI) ⁷ and DeepSeek-V3.2-chat (DeepSeek) ⁸. We evaluate these models on the same test sets used in RQ1 and RQ2, recording three key dimensions for each experimental run:

- **Predictive Performance:** Measured via AUC, Precision and AOD (Similar to RQ1 and RQ2).
- **Economic Cost:** For commercial models, we calculate the estimated monetary cost (in USD) for reproducing the results, based on the official API pricing schemas of the respective providers (e.g., OpenAI) at the time of the experiment. As for our open-weights models we estimate the cost (in USD) based on the hourly rate of the GPU infrastructure (e.g., NVIDIA H100) required to host the model during the inference window.

By positioning the performance gains of the large commercial LLMs against the efficient, localized inference of the 7B model equipped with Multiplex retrieval, we aim to determine if smarter context retrieval can effectively substitute for raw model scale in the context of LLM-powered logging level prediction.

Results: Multiplex-enhanced CodeLlama-7B consistently outperforms significantly larger commercial models across all studied projects, as shown in Table 6.10. In fact, we observe that our localized 7B model achieves an average AUC of 0.927, surpassing GPT-4o (0.788) by 17.6% and DeepSeek-V3.2 (0.659) by 40.6%. This performance gap is even more pronounced in the largest datasets, such

⁷<https://openai.com/index/hello-gpt-4o/>

⁸<https://api-docs.deepseek.com/news/news251201>

Table 6.10: Predictive Performance (AUC): Local Models vs. Commercial Baselines

Model	Cassandra		HBase		Hadoop		Elasticsearch		Average	
	AUC	AOD	AUC	AOD	AUC	AOD	AUC	AOD	AUC	AOD
CodeLlama-34B (Multiplex)	0.914	0.884	0.926	0.880	0.932	0.894	0.969	0.913	0.935	0.893
CodeLlama-7B (Multiplex)	0.907	0.877	0.914	0.868	0.921	0.883	0.966	0.912	0.927	0.885
GPT-4o (Zero-shot)	0.806	0.798	0.776	0.807	0.809	0.814	0.760	0.780	0.788	0.800
DeepSeek-V3.2 (Zero-shot)	0.669	0.779	0.693	0.802	0.679	0.794	0.595	0.775	0.659	0.788

as Elasticsearch, where the 7B model maintains a high AUC of 0.966 compared to 0.760 for GPT-4o. These results suggest that in the domain of logging level prediction, precise context retrieval is a more determinant factor for success than raw parameter scale. By feeding the model project-specific signals via Multiplex retrieval, a 7B model can effectively "punch above its weight" and defeat general-purpose models that are estimated to be 20 to 100× larger.

The CodeLlama-7B model occupies the optimal trade-off region between predictive performance, data privacy, and operational cost, as shown in Figure 6.7. While DeepSeek-V3.2 offers the lowest absolute price for sequential processing ($\approx \$0.07$ per 1k logs), this saving imposes a large penalty of 29% in predictive AUC and requires the transmission of proprietary code to external servers. In contrast, our self-hosted 7B model maintains superior accuracy (0.927 AUC) and strict data sovereignty. Furthermore, by leveraging concurrent batching on our experimental hardware, we achieve a unit cost of \$0.18 per 1k logs, as shown in Table 6.11. While slightly higher than the cheapest API, this cost remains negligible for high-value CI pipelines while mitigating the risks of data leakage and external service outages.

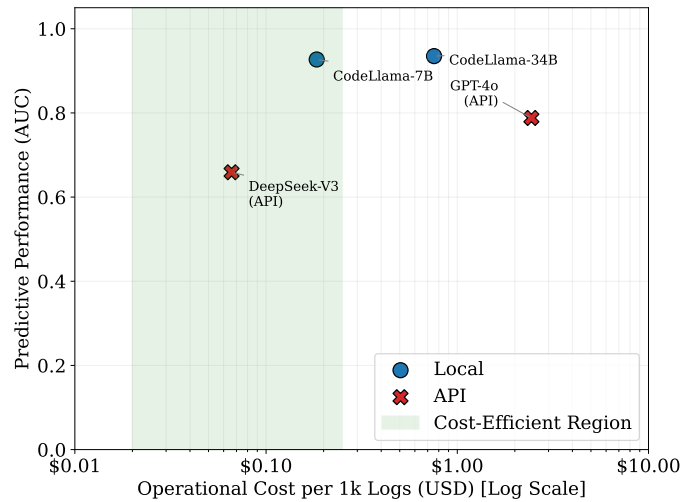


Figure 6.7: Predictive Accuracy vs. Operational Cost Analysis

Table 6.11: Operational Cost Comparison: Self-Hosted vs. Commercial APIs (per 1k Logs)

Model Strategy	Inference Method	Cost	vs. DeepSeek
CodeLlama-7B	Local (Batched)	\$0.18	Competitive
CodeLlama-34B	Local (Batched)	\$0.76	+1,051%
DeepSeek-V3.2	API (Token Rate)	\$0.07	Reference
GPT-4o	API (Token Rate)	\$2.44	+3,598%

Scaling up the local model from 7B to 34B parameters yields diminishing returns, as shown in Table 6.12. In our standardized experimental setup, we observe that while the CodeLlama-34B model achieves the highest absolute AUC of 0.935, it offers only a marginal improvement of 0.8% over the 7B model. However, this increment comes at a steep price, as the 34B model is significantly more computationally intensive, slowing down batch throughput by 4.1×. This throughput degradation directly translates to a 322% increase in operational costs (rising from \$0.18 to \$0.76 per 1k logs). Consequently, for CI/CD integrated logging level

prediction (where throughput is critical), the 7B model represents the superior efficiency choice.

Table 6.12: The Scaling Analysis: Performance Gains vs. Operational Cost Increases (4x H100)

Model Size	AUC	Cost/1k	Perf. Gain	Cost Inc.
7B (Base)	0.927	\$0.18	–	–
34B (Large)	0.935	\$0.76	+0.8%	+322%

Summary of RQ3

Our results show that a multiplex-retrieval enhanced 7B model outperforms commercial frontier models (GPT-4o, DeepSeek-V3.2) by up to 0.13 in terms of AUC for logging level prediction, proving that our specialized retrieval strategy is more critical than raw parameter scale. While the budget API (i.e., Deepseek) appears cheaper for low volumes, batched local inference achieves competitive unit costs (\approx \$0.18/1k logs) without data transmission risks. We recommend practitioners prioritize localized, retrieval-augmented models to ensure data sovereignty and reliable logging level recommendations.

6.5 Threats to validity

6.5.1 Internal Validity

An internal validity threat concerns our reliance on embedding models to represent files for clustering and retrieval. While we employed a state-of-the-art embedding model (CodeXEmbed) due to its proven effectiveness in capturing code semantics, any inherent biases or limitations in this model could influence the quality of our

clustering outcomes, consequently affecting the performance of LLM-powered LLPs using those clusters. To mitigate this threat, we validated cluster quality using established metrics (Silhouette, DBCV, and modularity) and assessed their stability through bootstrapping and temporal re-clustering analyses. Future work could benefit from exploring different embedding approaches to further strengthen the generalizability and robustness of the results.

6.5.2 Construct Validity

A construct validity threat involves the selection of clustering hyperparameters, such as the resolution parameter (γ) and cluster size thresholds (e.g., minimum cluster size). Arbitrary or suboptimal parameter choices could impact clustering results. To address this concern, we conducted specialized grid searches for each of our projects, identifying optimal parameter values based on robust clustering quality evaluation metrics, therefore ensuring our results' consistency and reducing sensitivity to hyperparameter choices.

6.5.3 External Validity

Regarding external validity, our empirical evaluation focused on four large-scale, widely-used open-source Java projects (Hadoop, HBase, Elasticsearch, and Cassandra). Although these projects represent significant open-source software systems, our findings might not fully generalize to smaller, proprietary, or non-Java software projects. Future work could replicate our analysis across diverse software ecosystems and languages. Another external validity threat relates to the LLMs used in our study. While our evaluation includes both open-weight (CodeLlama-7B, 34B) and

commercial state-of-the-art models (GPT-4o, DeepSeek-V3), our investigation into local deployment focused primarily on the CodeLlama family. Although CodeLlama is a widely adopted standard for code tasks, different open-weight architectures may exhibit distinct behaviors when coupled with our multiplex retrieval strategy. We mitigate this by leveraging multiple robust evaluation metrics (AUC, precision, AOD) and conducting statistical analyses of performance improvements. Future studies could explore different LLM architectures and their impact on logging level prediction.

One final external validity threat relates to our approach’s reliance on two specific clustering signals (semantic similarity and developer ownership). While these signals have demonstrated significant predictive performance improvements, other signals (e.g., defect density) might also influence developers’ logging level decisions and could lead to further performance gains. Nevertheless, our results show that our selected signals represent two core dimensions reflecting real-world logging practices (i.e., shared functionality and authorship), thus supporting the validity and utility of our hypothesis. Future research could explore incorporating additional signals to further enhance our approach’s predictive capabilities.

6.6 Chapter Summary

Selecting appropriate logging levels for new logging statements is an important, yet challenging software engineering task. Prior studies have leveraged machine learning models to automate logging level prediction. However, these approaches have either focused on local, code-driven metrics (like AST context) or used crude, file-wide engineered features , overlooking the socio-technical knowledge of a project.

In this chapter, we conducted a comparative study to investigate the impact of these signals: code functionality and developer ownership. We hypothesized that a modern LLM, guided by a novel retrieval strategy, could outperform classic models and commercial models with 100's of billions of tokens. To test our hypothesis, we introduced ownership-functionality aware retrieval, an In-Context Learning (ICL) retrieval approach that integrates semantic and ownership information simultaneously through multiplex clustering.

Our evaluation across four popular open-source projects demonstrates the superiority of this new paradigm. We show that by leveraging these combined signals for in-context-learning retrieval, establishes a new state-of-the-art for LLM-powered logging level prediction, achieving a median AUC between 0.90 and 0.96. This result significantly outperforms the naive LLM baseline and prior specialized supervised learning paradigms, including deep learning (DeepLV: Avg. AUC 0.8) and feature-engineered Ordinal Regression (Avg. AUC 0.734).

Our findings provide evidence that leveraging our evaluated socio-technical signals (i.e., code functionality and ownership) can have a high impact on the accuracy of LLM-powered LLPs. We envision this retrieval-augmented approach combining both signals simultaneously being extended to other logging tasks, such as log message generation and placement recommendations, further assisting developers in producing high quality logs.

CHAPTER 7

Conclusion

WE conclude this thesis by summarizing the research along with the key findings, and discussing their broader implications. We revisit the initial objectives and research questions, outlining the principal contributions to the relevant research area. Finally, we acknowledge the limitations of the study, and propose avenues for future research.

7.1 Closing the Loop: Beyond the Code for Logging Automation

This thesis started from a recurring tension that arises in practice: while logging is implemented in code, logging decisions are rarely determined by code alone. They

are instead shaped by the role of a component within a system, by the conventions of the developers who maintain it, and by how the system evolves over time. Motivated by this observation, the thesis advanced and investigated the hypothesis that automated approaches for supporting logging activities (particularly logging level prediction) are often constrained when they rely exclusively on source-code signals, and that their performance can be improved by incorporating a broader set of socio-technical signals. Namely, the multi-component structure of modern software systems, the evolution patterns of these systems, and the ownership dynamics of the files within these systems.

Across three studies, this thesis shows that these socio-technical signals are not peripheral context. Instead, they affect (i) how well LLP models generalize, (ii) how reliably they can be leveraged, and (iii) how effectively they can be deployed in software development and maintenance settings. In this way, this thesis positions logging not as a technical practice, but as a socio-technical one where automation must be aware of functionality constraints, evolution dynamics, and developer conventions to be reliable and accurate.

7.2 Summary of Key Findings and Contributions

7.2.1 Multi-component structure: the limits of “one global model”

The first study examined LLPs in multi-component systems, asking whether a single global model can accurately serve all components, despite the fact that components

differ in responsibility, maturity, and local conventions. The results provide consistent evidence that global models are frequently misaligned with local logging practices, as local, component-specific models outperform the global model on 60% to 100% of components. This finding matters because global evaluation can mask local underperformances, creating a risk that practitioners deploy a model that looks strong overall but behaves poorly in the individual components. To address the practical constraint that not all components have sufficient history for robust local training, the study proposed Peer-Local modeling, where models trained on peer components are reused for data-lacking components. The results show that peer-local models can substantially mitigate data scarcity: for data-lacking components. In fact, at least one peer-local model outperforms the global model on 88% (Spring) and 100% (OpenStack) of cases. At the same time, the study reports an important nuance: no single peer-local model is consistently superior across all data-lacking components, highlighting that peer selection itself is a non-trivial problem. Finally, this study shows that component differences are not only an LLP performance concern but also an LLP interpretability concern. Specifically, feature rankings derived from global models correlate only weakly with local feature rankings, implying that a “global explanation” can be misleading if it is used to inform component-specific logging guidance.

7.2.2 Evolution and time: risk of performance decay

The second study investigated the role of time in logging level prediction and demonstrated that temporal effects create a practical risk for deployment. Specifically, we quantified concept drift and showed that LLPs’ performance can degrade

shortly after training. Performance drops in AUC occur as early as a median of three months in Hadoop and two months in Spring and OpenStack. Such drops are statistically significant and occur for both shallow and deep LLPs. These findings provide direct evidence that LLP models are not static assets and their utility is time-dependent, and maintenance strategies must account for drift rather than assuming stable performance over long horizons. In response, this study evaluated contextual models and showed that temporal model updates can be beneficial. Notably, all-knowing models (using all available historical data) can underperform contextual models. That said, no single contextual window size dominates across settings, which supports our recommendation to treat the window size as a tunable hyperparameter rather than a fixed design choice. The study also shows that feature effects can change across time frames, further motivating time-aware interpretability rather than treating feature importance as a timeless characterization of logging behavior.

7.2.3 Ownership-Functionality Aware LLM retrieval: socio-technical context for in-context-learning

The third study focused on LLM-powered logging level prediction and investigated how to make in-context-learning (ICL) effective and reliable in the face of socio-technical variation observed in prior studies. The study argued that common ICL strategies (particularly random example selection) can be misaligned with how logging conventions are formed and maintained, since conventions can be localized to teams (e.g., seniority, preferences) and functionality (i.e., what the code does).

To address this, the study proposed a multiplex retrieval approach that fuses functionality and ownership signals to retrieve ICL examples that are socio-technically relevant. The reported results show substantial gains, as multiplex retrieval achieves statistically significant improvement over a random global retrieval baseline (SOTA in LLM-powered logging level prediction), reaching AUC values from 0.907 to 0.966 and improving AUC by 0.11 to 0.16. Importantly, the approach is positioned as operationally feasible, since our retrieval pipeline is implemented using FAISS and achieves a reported median latency of 8.9 ms, supporting the plausibility of interactive workflows (e.g., IDE suggestions). In addition to performance, this study emphasizes deployment-oriented trade-offs. Specifically, we observe that a retrieval-enhanced 7B model can outperform frontier commercial models (e.g., gpt-4o) by up to 0.13 AUC, and that scaling from 7B to 34B yields comparatively small gains relative to large cost increases (e.g., +0.8% AUC vs +322% cost per 1k logs in the reported comparison). Collectively, these findings support the thesis’ broader position that, for LLP, context selection is a first-class lever that may be more impactful than model size scaling alone.

7.3 Implications for Practice: What “Socio-technical” Means Operationally

The combined evidence across studies points to an actionable conclusion: reliable logging automation requires alignment with the socio-technical structure of software projects.

First, architectural boundaries matter. Component-specific models, peer-local modeling, and the observed divergence between global and local explanations together suggest that the granularity at which practitioners reason about logging (often component, or subsystem-level) should be reflected in the granularity at which LLP models are trained, evaluated, and interpreted.

Second, evolution and time matter. Concept drift can erode performance shortly after training. In practice, this implies that LLPs' deployments should incorporate time-aware evaluation and should include a maintenance plan (e.g., retraining schedule, informed window selection and performance monitoring) rather than treating a trained LLP as a set-and-forget artifact.

Third, ownership matters. The retrieval results indicate that ownership information of files within a codebase can be encoded into the selection of ICL examples, enabling relatively small models to achieve strong performance with low retrieval latency.

Our findings have a practical significance for organizations that care about cost, latency, and deployability, and for whom bigger models are not always an option (e.g., academic research labs).

7.4 Limitations and Threats to Validity

While each chapter of this thesis (Chapters 4, 5, and 6) includes a detailed discussion of chapter-specific threats to validity, we reflect in this section on broader limitations that apply across the thesis as a whole.

7.4.1 Scope of the studied artifact: logging statements vs. runtime log entries

Across all chapters, our empirical unit of analysis is the logging statement (a static code artifact), not the produced log entries at runtime. This design choice is intentional (it matches how developers instrument code), but it limits the extent to which our findings can speak to production-time phenomena such as log volume, log quality, sampling, downstream parsing, and operator workflows that depend on emitted log streams.

7.4.2 External validity: representativeness beyond the studied systems and settings

The thesis draws conclusions from large, modern software ecosystems, but they remain a subset of possible software contexts (e.g., proprietary systems, highly regulated domains, safety-critical environments, or organizations with strict logging standards). Similarly, language/library ecosystems and project governance models can shape logging practices in ways not fully captured in this thesis. Consequently, generalization should be made cautiously, particularly when transferring observations to settings with materially different constraints.

7.4.3 Construct validity: Implementing socio-technical signals

A central premise of the thesis is that logging decisions are shaped by socio-technical knowledge “beyond the code,” especially functionality/components, ownership, and temporal evolution.

However, each of these signals must be extracted from repository artifacts (e.g., directory structure for components, contribution history for ownership, and release/time windows for evolution). The captured version of this knowledge can be an imperfect proxy for the “ground truth” organizational realities. As a result, any measured effect of “components” or “ownership” should be interpreted as the effect of our designed proxy for that signal, and not of the effect of a definitive measurement of the underlying socio-technical aspect.

7.4.4 Reproducibility and portability of results across toolchains

The thesis relies on mining, preprocessing, and modeling pipelines whose outputs can be sensitive to (i) repository mining edge cases, (ii) tool versions, and (iii) modeling randomness (e.g., stochastic training, retrieval variation). We address this with repeated runs and standardized evaluation where feasible, yet exact numeric replication may still vary across environments.

7.4.5 Practical validity: deployment constraints and cost/benefit trade-offs

Finally, thesis-wide conclusions should be interpreted in light of operational constraints. Some approaches may require maintaining metadata (e.g., ownership matrices and code embeddings), periodic refreshes of models to handle concept drift, or additional compute for retrieval (e.g., more ICL examples) or model hosting. The practical value of an improvement depends on whether the incremental accuracy

translates to meaningful reductions in incident time-to-diagnosis, logging maintenance effort, or operational cost in a target organization outcomes not directly measured by offline predictive metrics.

In aggregate, these limitations do not negate the thesis’s core message, that socio-technical context (component purpose, ownership conventions, and evolution) materially shapes logging decisions, and can be leveraged to improve the automation of logging activities.

7.5 Future Work

7.5.1 Toward principled peer selection in multi-component systems

Peer-local modeling demonstrates that component-level transfer can solve the problem of relying on poor global models in the case of data-lacking components. However, peer-local modeling also reveals that peer selection is not straightforward. A promising direction is to develop methods that can reliably identify an appropriate peer-local model for a given data-lacking component, potentially integrating architectural similarity, dependency structure, and historical logging overlap to improve selection consistency.

7.5.2 Drift-aware maintenance strategies for LLPs

Our concept drift results motivate work on maintenance strategies that are practical for teams and systems where retraining is expensive or infrequent. Beyond, generic

hyperparameters tuning approaches (e.g., using Bayesian optimization) one direction for automated selection of contextual window size for deep LLPs, could involve investigating whether window size can be treated as a drift-sensitive decision variable. Specifically studying when shorter vs. longer historical contexts are preferable under different concept drift regimes, and developing an empirical criteria for selecting a context window size that balances recency with the data requirements of DL-LLPs.

7.5.3 Generalizing ownership-functionality retrieval beyond the studied setting

For ownership–functionality aware retrieval, future work could investigate how sensitive this retrieval is to the choice of representation and ecosystem, that is, whether the same retrieval gains persist when embeddings, and project characteristics change, and which conditions (e.g., project size, team structure, architectural modularity, or language/tooling) determine when our retrieval is most beneficial. In addition, future work can examine how these retrieval strategies translate from offline accuracy gains to deployment value, by studying what latency, cost, and privacy constraints are acceptable in practice and how they shape the feasibility of integrating retrieval-augmented LLP into developer workflows (e.g., IDE and CI settings).

7.6 Closing Remarks

The research presented in this thesis supports a central conclusion: logging automation cannot be treated as a code-centric prediction task without risking brittle generalization, invalid usage, and impractical deployment assumptions. Instead, logging must be understood as a socio-technical activity that is shaped by architectural boundaries, by the evolution of systems over time, and by the conventions of the developers who maintain those systems.

By empirically demonstrating the limitations of global LLP models in multi-component systems, quantifying the performance decay of frozen LLPs across time, and designing ownership–functionality aware retrieval for LLM-powered LLP, this thesis contributes a coherent evidence base for designing logging automation that is more accurate, and more aware of the socio-technical variation in logging conventions. Ultimately, the thesis argues that “beyond the code” is not a slogan but a methodological requirement for building automated logging support that can be trusted in the complex, evolving socio-technical environments where software is developed and operated.

Bibliography

- Agrawal, A. and Menzies, T. (2019). Is AI different for SE? *NC State Univ.*
- Anu, H., Chen, J., Shi, W., Hou, J., Liang, B., and Qin, B. (2019). An approach to recommendation of verbosity log levels based on logging intention. In *IEEE International Conference on Software Maintenance and Evolution*, pages 125–134.
- Baligodugula, V. V. and Amsaad, F. (2025). Unsupervised learning: Comparative analysis of clustering techniques on high-dimensional data. *ArXiv*. <https://arxiv.org/abs/2503.23215>.
- Battiston, F., Nicosia, V., and Latora, V. (2014). Structural measures for multiplex networks. *Physical Review E*, 89:032804.
- Bennin, K. E., Ali, N. b., Börstler, J., and Yu, X. (2020). Revisiting the impact of concept drift on just-in-time quality assurance. In *IEEE International Conference on Software Quality, Reliability and Security*, pages 53–59.

- Bertram, T., Fürnkranz, J., and Müller, M. (2022). Quantity vs quality: Investigating the trade-off between sample size and label reliability. *ArXiv*. <https://arxiv.org/abs/2204.09462>.
- Blondel, V. D., Guillaume, J.-L., Lambiotte, R., and Lefebvre, E. (2008). Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):P10008.
- Cândido, J., Haesen, J., Aniche, M. F., and van Deursen, A. (2021). An exploratory study of log placement recommendation in an enterprise system. In *IEEE/ACM International Conference on Mining Software Repositories*, pages 143–154.
- Chahar, R. and Kaur, D. (2020). A systematic review of the machine learning algorithms for the computational analysis in different domains. *International Journal of Advanced Technology and Engineering Exploration*, 7(71):147.
- Chen, A., Yao, K., Zhang, H., Tang, Y., and Shang, W. (2025). An empirical study of logging practice in cuda-based deep learning systems. In *International Conference on Software Quality, Reliability and Security*, pages 164–175.
- Chen, B. and Jiang, Z. M. (2017). Characterizing and detecting anti-patterns in the logging code. In *IEEE/ACM International Conference on Software Engineering*, pages 71–81.
- Chen, B. and Jiang, Z. M. (2019). Extracting and studying the logging-code-issue-introducing changes in java-based large-scale open source software systems. *Empirical Software Engineering*, 24(4):2285–2322.

- Chen, B. and Jiang, Z. M. J. (2020). Studying the use of java logging utilities in the wild. In *IEEE/ACM International Conference on Software Engineering*, page 397–408.
- Chen, Y., Xu, Y., Li, H., Kang, Y., Yang, X., Lin, Q., Zhang, H., Gao, F., Xu, Z., Dang, Y., Zhang, D., and Dong, H. (2019). Outage prediction and diagnosis for cloud service systems. In *WWW '19: The World Wide Web Conference*, pages 2659–2665.
- Chochlov, M., Aftab Ahmed, G., Vincent Patten, J., Lu, G., Hou, W., Gregg, D., and Buckley, J. (2022). Using a nearest-neighbour, bert-based approach for scalable clone detection. In *IEEE International Conference on Software Maintenance and Evolution*, page 582–591.
- Cohen, J. (1992). A power primer. *Psychological bulletin*, 112:155.
- Davies, D. L. and Bouldin, D. W. (1979). A cluster separation measure. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, (2):224–227.
- De Domenico, M., Solè-Ribalta, A., Cozzo, E., Kivelä, M., Moreno, Y., Porter, M., Gomez, S., and Arenas, A. (2013). Mathematical formulation of multi-layer networks. *Physical Review X*, 3.
- Ding, Z., Li, H., and Shang, W. (2022). Logentext: Automatically generating logging texts using neural machine translation. In *IEEE International Conference on Software Analysis, Evolution and Reengineering*, pages 349–360.
- Ding, Z., Tang, Y., Cheng, X., Li, H., and Shang, W. (2023). Logentext-plus: Improving neural machine translation based logging texts generation with syntactic templates. *ACM Transactions on Software Engineering and Methodology*, 33(2).

- D'Ambros, M., Lanza, M., and Robbes, R. (2012). Evaluating defect prediction approaches: A benchmark and an extensive comparison. *Empirical Software Engineering*, pages 1–47.
- Ekanayake, J. B., Tappolet, J., Gall, H. C., and Bernstein, A. (2009). Tracking concept drift of software projects using defect prediction quality. *IEEE International Working Conference on Mining Software Repositories*, pages 51–60.
- El-Sayed, N., Zhu, H., and Schroeder, B. (2017). from failure across multiple clusters: A trace-driven approach to understanding, predicting, and mitigating job terminations. In *IEEE International Conference on Distributed Computing Systems*, pages 1333–1344.
- Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., and Zhou, M. (2020). Codebert: A pre-trained model for programming and natural languages. <https://arxiv.org/abs/2002.08155>.
- Foalem, P. L., Khomh, F., and Li, H. (2024). Studying logging practice in machine learning-based applications. *Information and Software Technology*, 170(C).
- Foalem, P. L., Silva, L. M. P. D., Khomh, F., Li, H., and Merlo, E. (2025). Logging requirement for continuous auditing of responsible machine learning-based applications. *Empirical Software Engineering*, 30(3).
- Fortunato, S. and Barthelemy, M. (2007). Resolution limit in community detection. *Proceedings of the National Academy of Sciences of the United States of America*, 104:36–41.

- Fu, Q., Zhu, J., Hu, W., Lou, J.-G., Ding, R., Lin, Q., Zhang, D., and Xie, T. (2014). Where do developers log? an empirical study on logging practices in industry. In *IEEE International Conference on Software Engineering*, page 24–33.
- Fu, W. and Menzies, T. (2017). Easy over hard: A case study on deep learning. In *ACM Joint Meeting on Foundations of Software Engineering*, page 49–60.
- Gama, J., Medas, P., Castillo, G., and Rodrigues, P. (2004). Learning with drift detection. volume 8, pages 286–295.
- Gama, J. a., Žliobaitundefined, I., Bifet, A., Pechenizkiy, M., and Bouchachia, A. (2014). A survey on concept drift adaptation. *ACM Computing Surveys*, 46(4):1–37.
- G.Ditzler, M.Roveri, C.Alippi, and R.Polikar (2015). Learning in nonstationary environments: A survey. *IEEE Computational Intelligence Magazine*, 10(4):12–25.
- Gong, Z., Zhong, P., and Hu, W. (2019). Diversity in machine learning. *IEEE Access*, PP:1–1.
- Graves, T., Karr, A., Marron, J., and Siy, H. (2000). Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, pages 653–661.
- Günther, M., Mohr, I., Wang, B., and Xiao, H. (2024). Late chunking: Contextual chunk embeddings using long-context embedding models. <https://arxiv.org/abs/2409.04701>.
- Hand, D. J. and Till, R. J. (2001). A simple generalisation of the area under the roc curve for multiple class classification problems. *Mach. Learn.*, 45(2):171–186.

- Hanley, J. A. and McNeil, B. (1982). The meaning and use of the area under a receiver operating characteristic (roc) curve. *Radiology*, 143:29–36.
- Harrell, F. E. (2001). *Regression Modeling Strategies*. Springer Int. Publishing.
- Hassan, A. E. (2009). Predicting faults using the complexity of code changes. In *IEEE/ACM International Conference on Software Engineering*, pages 78–88.
- Haynes, W. (2013). *Bonferroni Correction*, pages 154–154. Springer New York.
- He, S., He, P., Chen, Z., Yang, T., Su, Y., and Lyu, M. R. (2021). A survey on automated log analysis for reliability engineering. *ACM Computer Survey*, 54(6).
- Heng, Y. W., Ma, Z., Li, Z., Kim, D. J., and Chen, T.-H. (2024). Studying and benchmarking large language models for log level suggestion. *ArXiv*. <https://arxiv.org/abs/2410.08499>.
- Herbsleb, J. and Mockus, A. (2003). An empirical study of speed and communication in globally distributed software development. *IEEE Transactions on Software Engineering*, 29(6):481–494.
- Hurvich, C. and Tsai, C.-L. (2008). A corrected akaike information criterion for vector autoregressive model selection. *Journal of Time Series Analysis*, 14:271 – 279.
- Hurvich, C. M. and Tsai, C.-L. (1989). Regression and time series model selection in small samples. *Biometrika*, pages 297–307.

- Jiarpakdee, J., Tantithamthavorn, C., and Hassan, A. E. (2019). The impact of correlated metrics on the interpretation of defect models. *IEEE Transactions on Software Engineering*, pages 320–331.
- Kabinna, S., Bezemer, C.-P., Shang, W., and Hassan, A. E. (2016). Logging library migrations: a case study for the apache software foundation projects. In *International Conference on Mining Software Repositories*, page 154–164.
- Kabinna, S., Shang, W., Bezemer, C., and Hassan, A. E. (2016). Examining the stability of logging statements. In *2016 IEEE International Conference on Software Analysis, Evolution, and Reengineering*, volume 1, pages 326–337.
- Kalman, R. (1960). On the general theory of control systems. *IFAC Proceedings Volumes*, 1(1):491–502. 1st International IFAC Congress on Automatic and Remote Control, Moscow, USSR, 1960.
- Kaufman, S., Rosset, S., and Perlich, C. (2011). Leakage in data mining: Formulation, detection, and avoidance. volume 6, pages 556–563.
- Kim, T., Kim, S., Park, S., and Park, Y. (2020). Automatic recommendation to appropriate log levels. *Software: Practice and Experience*, 50(3):189–209.
- Lee, D., Rajbahadur, G. K., Lin, D., Sayagh, M., Bezemer, C.-P., and Hassan, A. E. (2020). An empirical study of the characteristics of popular minecraft mods. *Empirical Software Engineering*, 25(5):3396–3429.
- Li, H., Shang, W., Adams, B., Sayagh, M., and Hassan, A. E. (2021a). A Qualitative Study of the Benefits and Costs of Logging From Developers’ Perspectives . *IEEE Transactions on Software Engineering*, 47(12):2858–2873.

- Li, H., Shang, W., and Hassan, A. E. (2017a). Which log level should developers choose for a new logging statement? *Empirical Software Engineering*, 22(4):1684–1716.
- Li, H., Shang, W., Zou, Y., and Hassan, A. E. (2017b). Towards just-in-time suggestions for log changes. *Empirical Software Engineering*, 22(4):1831–1865.
- Li, Y., Huo, Y., Jiang, Z., Zhong, R., He, P., Su, Y., Briand, L. C., and Lyu, M. R. (2024). Exploring the effectiveness of llms in automated logging statement generation: An empirical study. *IEEE Transactions on Software Engineering*, 50(12):3188–3207.
- Li, Z., Chen, T.-H., and Shang, W. (2020). Where shall we log? studying and suggesting logging locations in code blocks. In *IEEE/ACM International Conference on Automated Software Engineering*, pages 361–372.
- Li, Z., Chen, T.-H., Yang, J., and Shang, W. (2022). Studying duplicate logging statements and their relationships with code clones. *IEEE Transactions on Software Engineering*, 48(7):2476–2494.
- Li, Z., Li, H., Chen, T.-H., and Shang, W. (2021b). Deeplv: Suggesting log levels using ordinal based neural networks. In *IEEE/ACM International Conference on Software Engineering*, pages 1461–1472.
- Lin, Q., Hsieh, K., Dang, Y., Zhang, H., Sui, K., Xu, Y., Lou, J., Li, C., Wu, Y., Yao, R., Chintalapati, M., and Zhang, D. (2018). Predicting node failure in cloud service systems. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, page 480–490.

- Liu, J., Zeng, J., Wang, X., Ji, K., and Liang, Z. (2022). Tell: log level suggestions via modeling multi-level code block information. In *ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2022*, page 27–38, New York, NY, USA. Association for Computing Machinery.
- Liu, X., Jia, T., Li, Y., Yu, H., Yue, Y., and Hou, C. (2020). Automatically generating descriptive texts in logging statements: How far are we? In *Programming Languages and Systems: 18th Asian Symposium, APLAS 2020*, page 251–269.
- Liu, Y., Meng, R., Joty, S., Savarese, S., Xiong, C., Zhou, Y., and Yavuz, S. (2024). Codexembed: A generalist embedding model family for multilingual and multi-task code retrieval. *ArXiv*. <https://arxiv.org/abs/2411.12644>.
- Liu, Z., Xia, X., Hassan, A. E., Lo, D., Xing, Z., and Wang, X. (2018). Neural-machine-translation-based commit message generation: How far are we? In *IEEE/ACM International Conference on Automated Software Engineering*, pages 373–384.
- Lyu, Y., Li, H., Sayagh, M., Jiang, Z. M. J., and Hassan, A. E. (2021a). An empirical study of the impact of data splitting decisions on the performance of aiops solutions. *ACM Transactions on Software Engineering and Methodology*, 30(4).
- Lyu, Y., Rajbahadur, G. K., Lin, D., Chen, B., and Jiang, Z. M. J. (2021b). Towards a consistent interpretation of aiops models. *ACM Transactions on Software Engineering and Methodology*, 31(1).

- Ma, Z., Chen, A. R., Kim, D. J., Chen, T.-H., and Wang, S. (2024). Llmparser: An exploratory study on using large language models for log parsing. *IEEE/ACM International Conference on Software Engineering*, pages 1209–1221.
- Malzer, C. and Baum, M. (2020). A hybrid approach to hierarchical density-based cluster selection. In *IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems*, pages 223–228.
- Mastroiolo, A., Ferrari, V., Pascarella, L., and Bavota, G. (2024). Log statements generation via deep learning: Widening the support provided to developers. *Journal of Systems and Software*, 210:111947.
- McInnes, L., Healy, J., and Melville, J. (2020). Umap: Uniform manifold approximation and projection for dimension reduction. *ArXiv*. <https://arxiv.org/abs/1802.03426>.
- McIntosh, S., Kamei, Y., Adams, B., and Hassan, A. E. (2014). The impact of code review coverage and code review participation on software quality. In *The Working Conference on Mining Software Repositories*, page 292–201.
- Moulavi, D., Andretta Jaskowiak, P., Campello, R., Zimek, A., and Sander, J. (2014). Density-based clustering validation. In *SIAM International Conference on Data Mining*.
- Olewicki, D., Nayrolles, M., and Adams, B. (2022). Towards language-independent brown build detection. In *International Conference on Software Engineering*, page 2177–2188.

- Oliner, A., Ganapathi, A., and Xu, W. (2012). Advances and challenges in log analysis. *Communications of the ACM*, 55(2):55–61.
- Ouatiti, Y. E. (2024). The impact of concept drift and data leakage on log level prediction models - appendix. <https://zenodo.org/records/10898284>.
- Ouatiti, Y. E., Sayagh, M., Adams, B., and Hassan, A. E. (2026). Retrieval supersedes scale: Efficient log level prediction via multiplex socio-technical context. Under review at *IEEE Transactions on Software Engineering*.
- Ouatiti, Y. E., Sayagh, M., Kerzazi, N., Adams, B., and Hassan, A. E. (2024). The impact of concept drift and data leakage on log level prediction models. *Empirical Software Engineering*, 29(5).
- Ouatiti, Y. E., Sayagh, M., Kerzazi, N., and Hassan, A. E. (2023). An Empirical Study on Log Level Prediction for Multi-Component Systems. *IEEE Transactions on Software Engineering*, 49(02):473–484.
- Patel, K., Faccin, J. G., Hamou-Lhadj, A., and Nunes, I. (2022). The sense of logging in the linux kernel. *Empirical Software Engineering*, 27:153–196.
- Pecchia, A., Cinque, M., Carrozza, G., and Cotroneo, D. (2015). Industry practices and event logging: Assessment of a critical software development process. In *International Conference on Software Engineering*, pages 169–178.
- Pecchia, A., Cinque, M., and Cotroneo, D. (2013). Event Logs for the Analysis of Software Failures: A Rule-Based Approach. *IEEE Transactions on Software Engineering*, 39(06):806–821.

- Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. (2020). Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(1).
- Rajbahadur, G. K., Oliva, G., Hassan, A. E., and Dingel, J. (2019). Pitfalls analyzer: Quality control for model-driven data science pipelines. In *International Conference on Model Driven Engineering Languages and Systems*, pages 12–22.
- Rajbahadur, G. K., Wang, S., Ansaldi, G., Kamei, Y., and Hassan, A. E. (2021). The impact of feature importance methods on the interpretation of defect classifiers. *IEEE Transactions on Software Engineering*, (7):2245–2261.
- Rawte, V., Sheth, A., and Das, A. (2023). A survey of hallucination in "large" foundation models. <https://arxiv.org/abs/2309.05922>.
- Rong, G., Gu, S., Shen, H., Zhang, H., and Kuang, H. (2023). How do developers' profiles and experiences influence their logging practices? an empirical study of industrial practitioners. In *International Conference on Software Engineering*, pages 855–867.
- Rousseeuw, P. J. (1987). Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics*, 20:53–65.
- Rudin, C. (2019). Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead. *Nature Machine Intelligence*, 1:206–215.

- Sarro, F., Moussa, R., Petrozziello, A., and Harman, M. (2022). Learning from mistakes: Machine learning enhanced human expert effort estimates. *IEEE Transactions on Software Engineering*, 48(6):1868–1882.
- Sayagh, M., Dong, Z., Andrzejak, A., and Adams, B. (2017). Does the choice of configuration framework matter for developers? Empirical study on 11 java configuration frameworks. In *International Working Conference on Source Code Analysis and Manipulation*, pages 41–50.
- Shang, Weiyi, M. N. and Hassan, A. E. (2015). Studying the relationship between logging characteristics and the code quality of platform software. *Empirical Software Engineering*, page 1–27.
- Shihab, E., Jiang, Z. M., Ibrahim, W. M., Adams, B., and Hassan, A. E. (2010). Understanding the impact of code and process metrics on post-release defects: a case study on the eclipse project. In *IEEE/ACM International Symposium on Empirical Software Engineering and Measurement*, New York, NY, USA. Association for Computing Machinery.
- Shihab, E., Kamei, Y., Adams, B., and Hassan, A. E. (2013). Is lines of code a good measure of effort in effort-aware models? *Information and Software Technology*, pages 1981–1993.
- Song, L., Wang, Y., Han, Y., Zhao, X., Liu, B., and Li, X. (2016). C-brain: a deep learning accelerator that tames the diversity of cnns through adaptive data-level parallelization. In *Annual Design Automation Conference, DAC '16*, New York, NY, USA. Association for Computing Machinery.

- Sridharan, C. (2018). *Distributed Systems Observability: A Guide to Building Robust Systems*. O'Reilly Media.
- Steinley, D. (2004). Properties of the hubert-arabie adjusted rand index. *Psychological methods*, 9:386–96.
- Subramanyam, R. and Krishnan, M. (2003). Empirical analysis of ck metrics for object-oriented design complexity: implications for software defects. *IEEE Transactions on Software Engineering*, 29(4):297–310.
- Tan, B., Xu, J., Zhu, Z., and He, P. (2025). Al-bench: A benchmark for automatic logging. *ArXiv*, abs/2502.03160. <https://api.semanticscholar.org/CorpusID:276116403>.
- Tantithamthavorn, C. (2018). *ScottKnottESD: The Scott-Knott Effect Size Difference (ESD) Test*. <https://cran.r-project.org/package=ScottKnottESD>.
- Tantithamthavorn, C. and Hassan, A. E. (2018). An experience report on defect modelling in practice: pitfalls and challenges. In *IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE-SEIP '18*, page 286–295, New York, NY, USA. Association for Computing Machinery.
- Tantithamthavorn, C., Hassan, A. E., and Matsumoto, K. (2020). The impact of class rebalancing techniques on the performance and interpretation of defect prediction models. *IEEE Transactions on Software Engineering*, 46(11):1200–1219.
- Tantithamthavorn, C., Jiarpakdee, J., and Grundy, J. (2021). Actionable analytics: Stop telling me what it is; please tell me what to do. *IEEE Software*, 38(4):115–120.

- Tantithamthavorn, C., McIntosh, S., Hassan, A. E., and Matsumoto, K. (2019a). The impact of automated parameter optimization on defect prediction models. *IEEE Transactions on Software Engineering*, 45(7):683–711.
- Tantithamthavorn, C., McIntosh, S., Hassan, A. E., and Matsumoto, K. (2019b). The impact of automated parameter optimization on defect prediction models. *IEEE Transactions on Software Engineering*, 45(07):683–711.
- Thongtanunam, P. and Hassan, A. E. (2018). Review dynamics and its impact on software quality. *IEEE Transactions on Software Engineering*, pages 1–13.
- Traag, V. A., Waltman, L., and van Eck, N. J. (2019). From louvain to leiden: guaranteeing well-connected communities. *Scientific Reports*, 9(1).
- Wilks, D. S. (2011). *Statistical Methods in the Atmospheric Sciences*. Academic Press, Amsterdam, 3rd edition.
- Xu, J., Cui, Z., Zhao, Y., Zhang, X., He, S., He, P., Li, L., Kang, Y., Lin, Q., Dang, Y., Rajmohan, S., and Zhang, D. (2024). Unilog: Automatic logging via llm and in-context learning. In *International Conference on Software Engineering*.
- Xu, Y., Sui, K., Yao, R., Zhang, H., Lin, Q., Dang, Y., Li, P., Jiang, K., Zhang, W., Lou, J., Chintalapati, M., and Zhang, D. (2018). Improving service availability of cloud systems by predicting disk error. In *Proc. of the 2018 USENIX Conf. on Usenix Annual Technical Conf.*, page 481–493.
- Yu, X., Liu, L., Hu, X., Keung, J. W., Liu, J., and Xia, X. (2024). Fight fire with fire: How much can we trust chatgpt on source code-related tasks? *IEEE Transactions on Software Engineering*, 50(12):3435–3453.

- Yuan, D., D., Luo, Y., Zhuang, X., Rodrigues, G. R., Zhao, X., Zhang, Y., Jain, P. U., and Stumm, M. (2014). Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *Proceedings of the 11th Conference on Operating Systems Design and Implementation, Systems Design and Implementation*, pages 249–265.
- Yuan, D., Mai, H., Xiong, W., Tan, L., Zhou, Y., and Pasupathy, S. (2010). Sherlog: Error diagnosis by connecting clues from run-time logs. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 143–154.
- Yuan, D., Park, S., and Zhou, Y. (2012). Characterizing logging practices in open-source software. In *International Conference on Software Engineering*, pages 102–112.
- Yuan, D., Zheng, J., Park, S., Zhou, Y., and Savage, S. (2011). Improving software diagnosability via log enhancement. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, page 3–14.
- Zhang, D. and Tsai, J. (2002). Machine learning and software engineering.
- Zhang, F., Zheng, Q., Zou, Y., and Hassan, A. E. (2016). Cross-project defect prediction using a connectivity-based unsupervised classifier. In *IEEE/ACM International Conference on Software Engineering*, pages 309–320.
- Zhou, Z.-H. (2012). *Ensemble Methods: Foundations and Algorithms*. Chapman amp; Hall/CRC.

Zhu, J., He, P., Fu, Q., Zhang, H., Lyu, M. R., and Zhang, D. (2015). Learning to log: Helping developers make informed logging decisions. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, pages 415–425.

A Appendix - Logging level prediction for multi-component software systems

A.1 Performance of the global model on components

Figures [A.1](#) and [A.2](#) show the AUC and Brier Score performance of the global model trained using data from the entire projects on the components of these projects.

A.2 Local models Vs. Global models

Figures [A.3](#), [A.4](#) and [A.5](#) show the comparison between the performances and quality of fit of the global, local and mixed effect models.

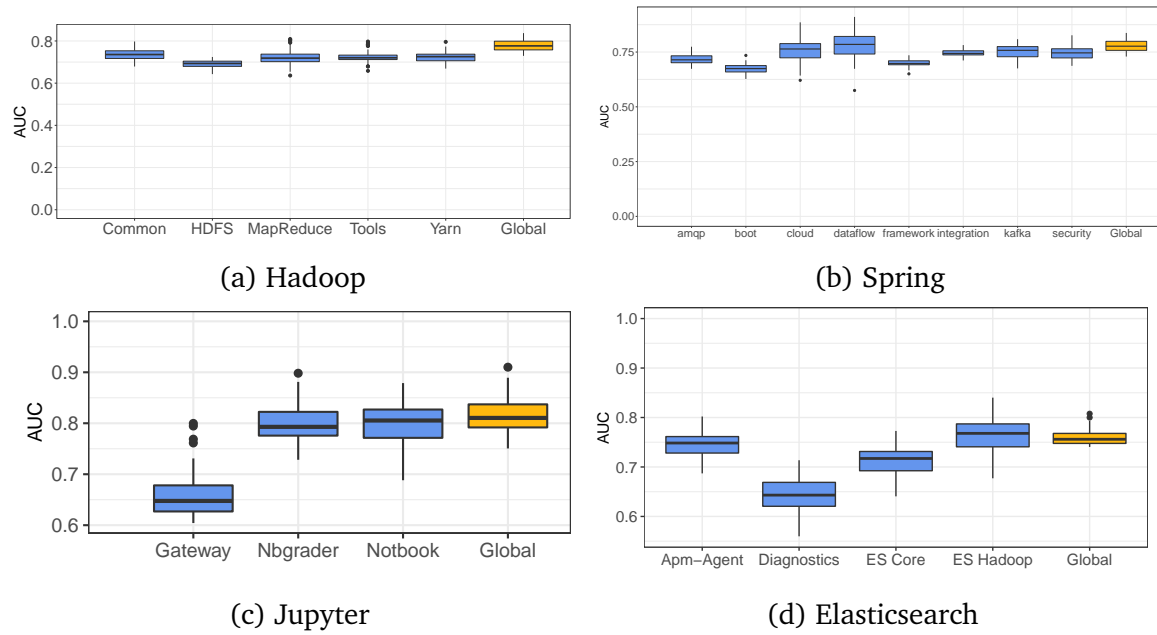


Figure A.1: The AUC performance of the global model on components and on the entire project

A.3 Peer-local models Vs. Global model

The following Figures show the comparison between the AUCs of the peer-local models and the global model on data-lacking components for OpenStack and Spring projects.

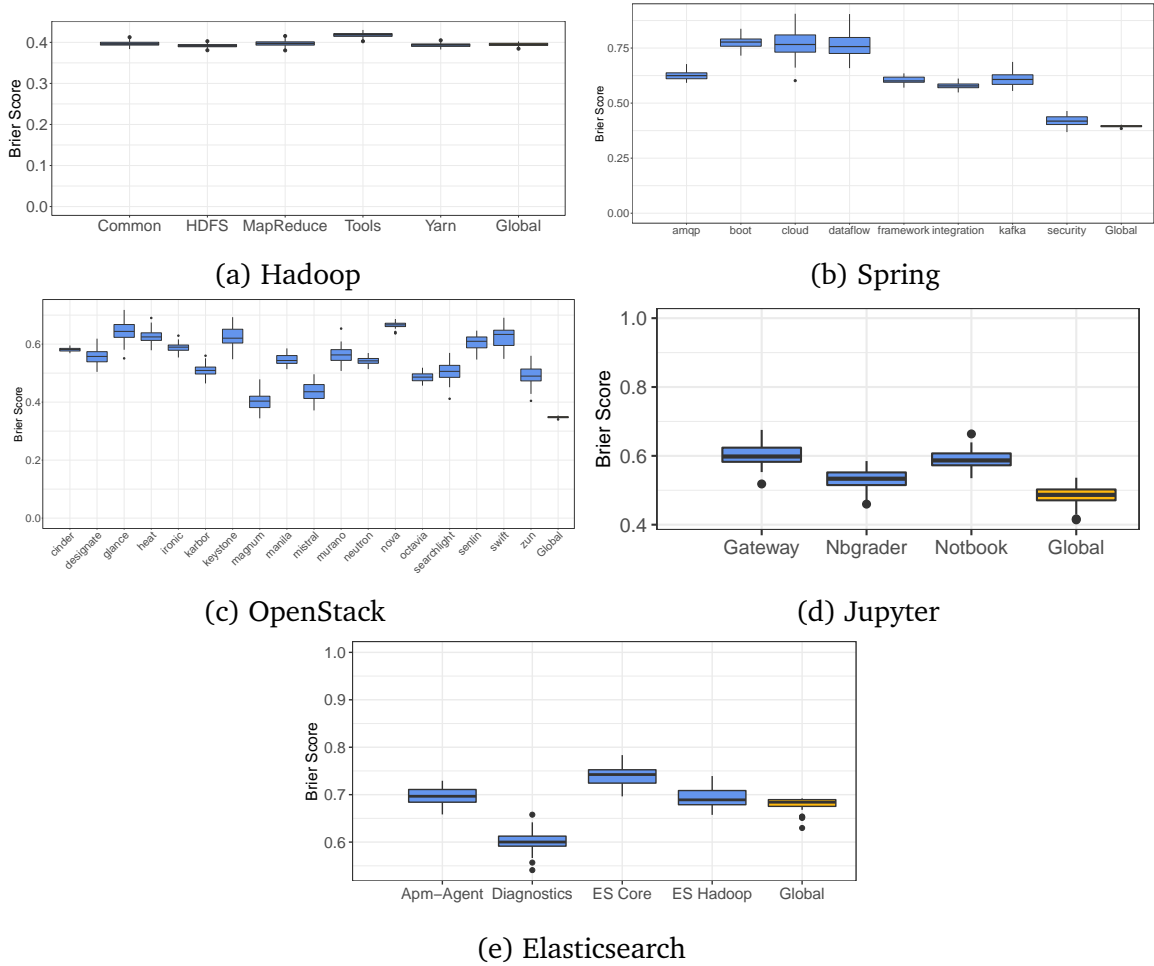


Figure A.2: The Brier Score performance of the global model on different components and on the entire project

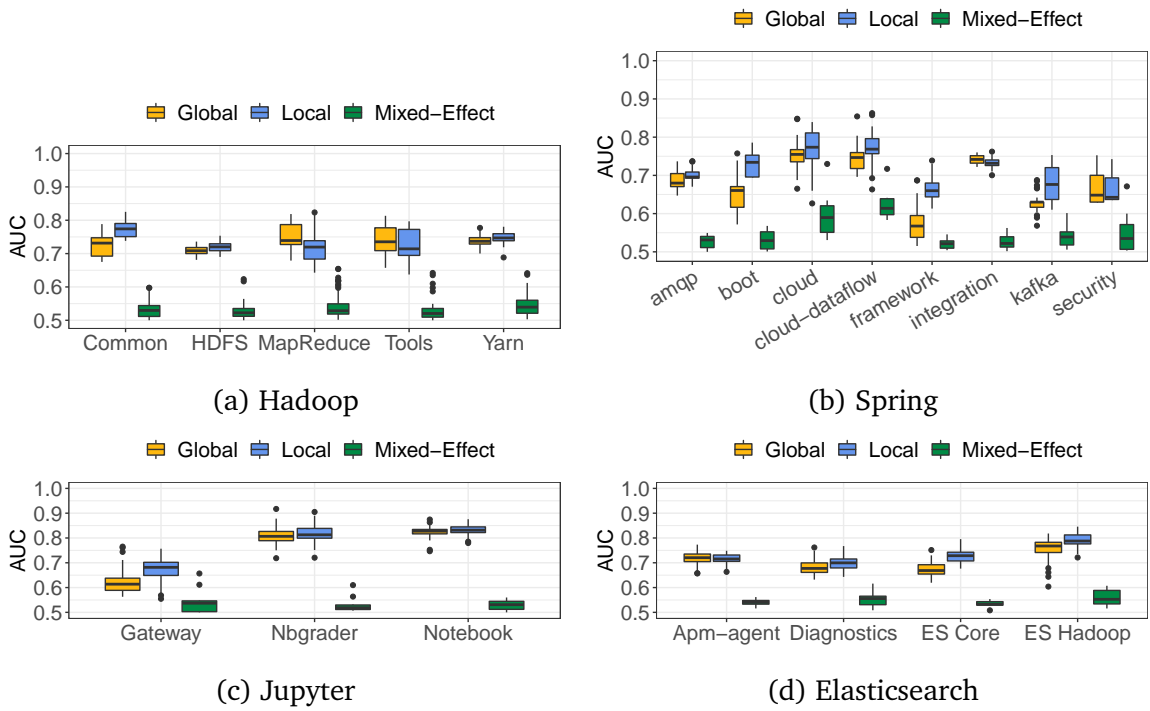
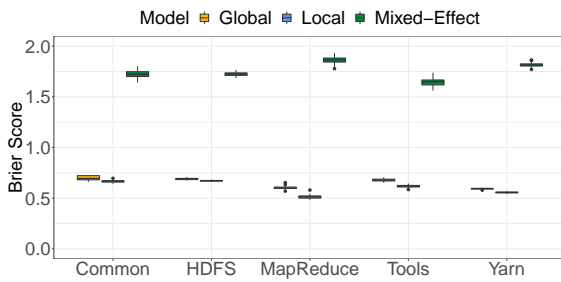
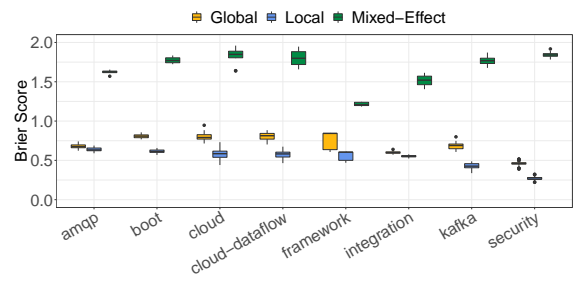


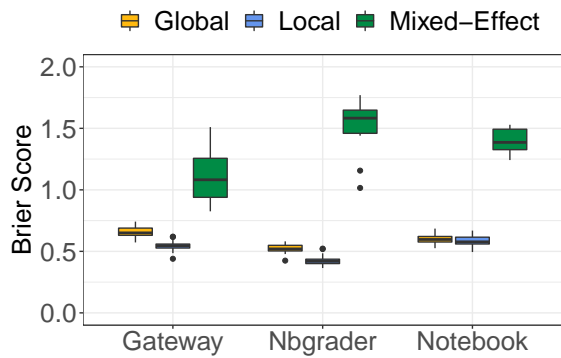
Figure A.3: AUC scores of the global, local and mixed-effect models



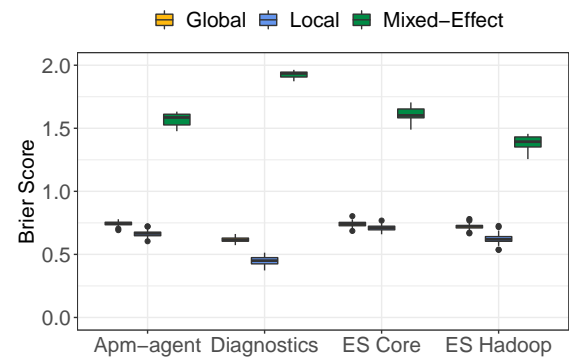
(a) Hadoop



(b) Spring



(c) Jupyter



(d) Elasticsearch

Figure A.4: Brier scores of the global, local and mixed-effect models

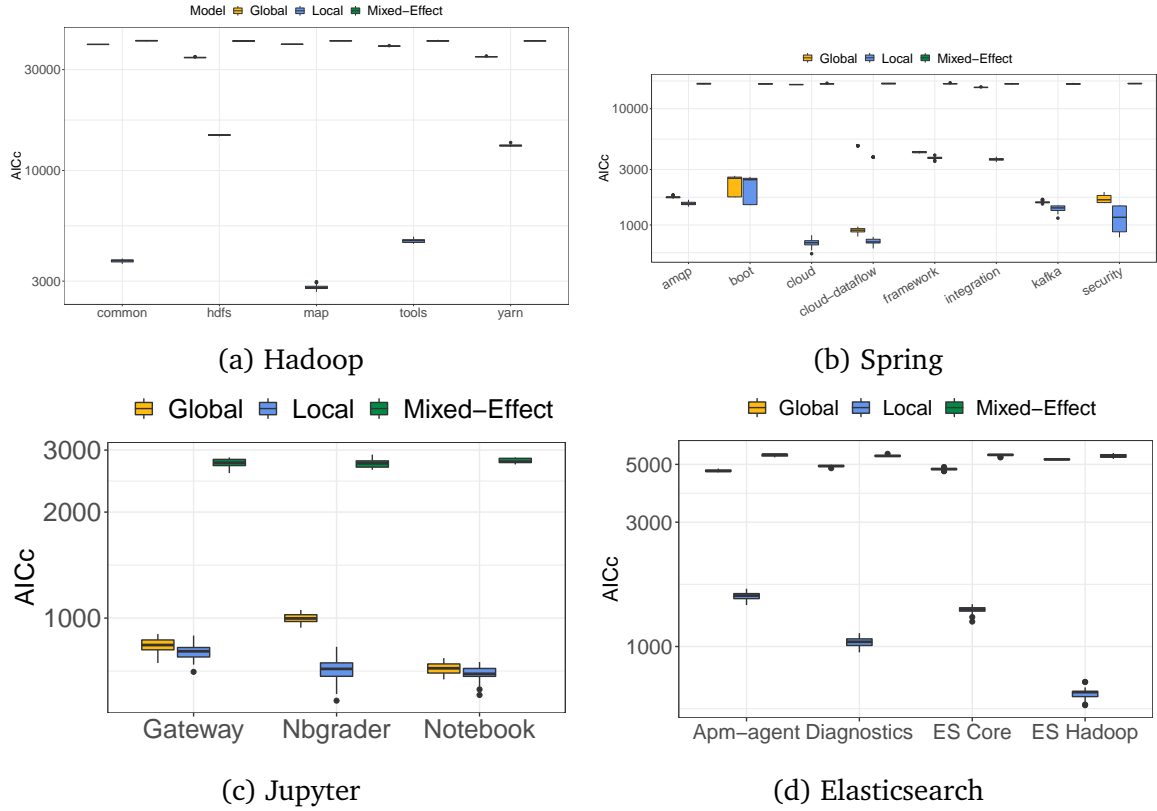


Figure A.5: AICc scores of the global, local and mixed-effect models (log-scale)

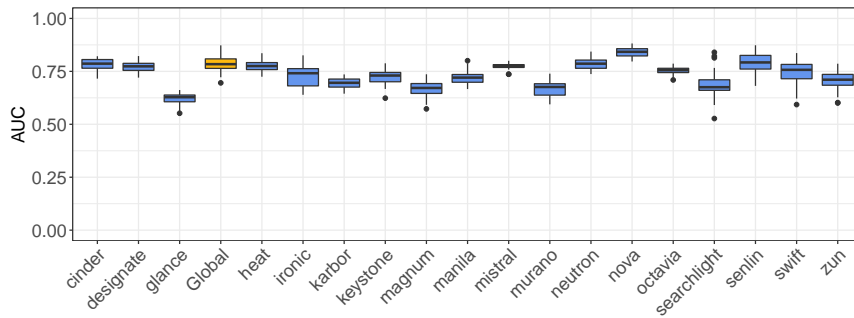


Figure A.6: OpenStack - Blazar

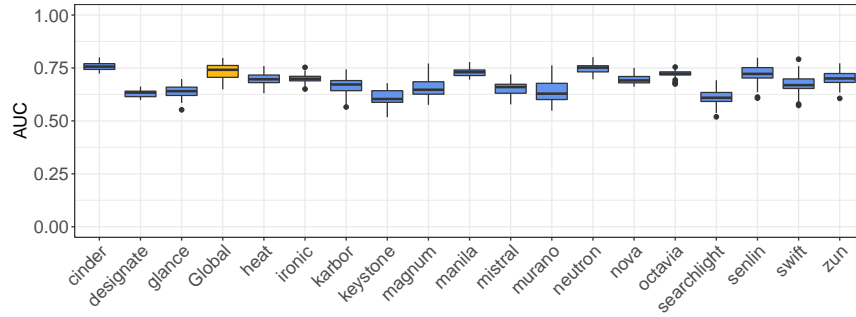


Figure A.7: OpenStack - Cyborg

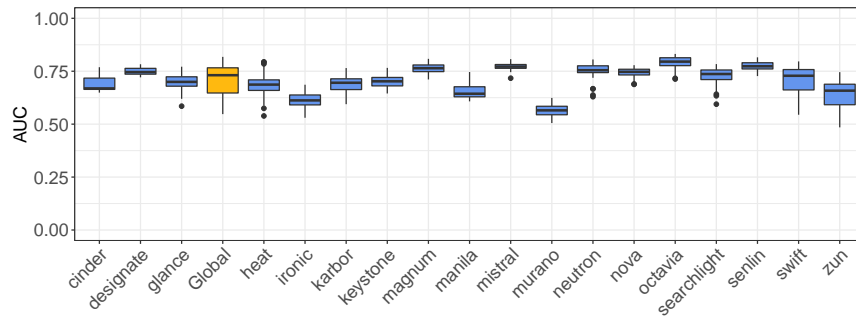


Figure A.8: OpenStack - ec2-api

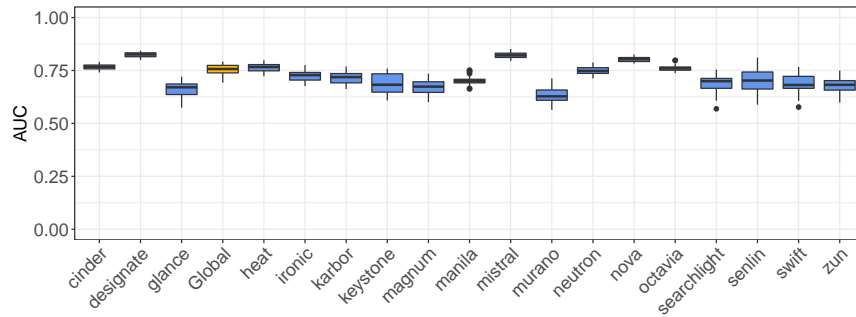


Figure A.9: OpenStack - Horizon

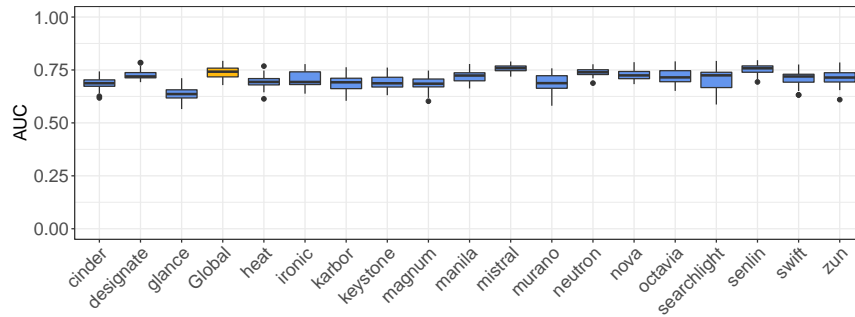


Figure A.10: OpenStack - Masakari

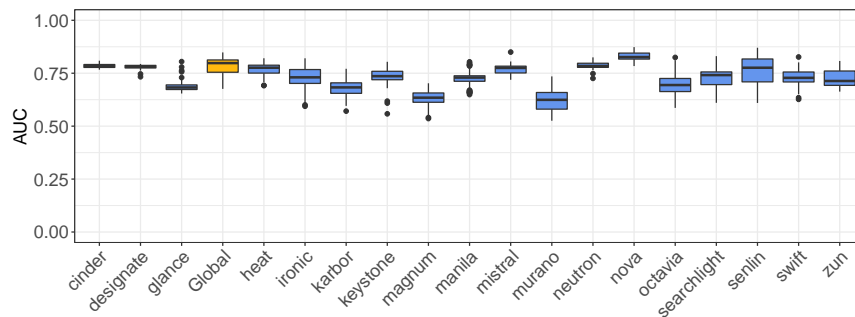


Figure A.11: OpenStack - Placement

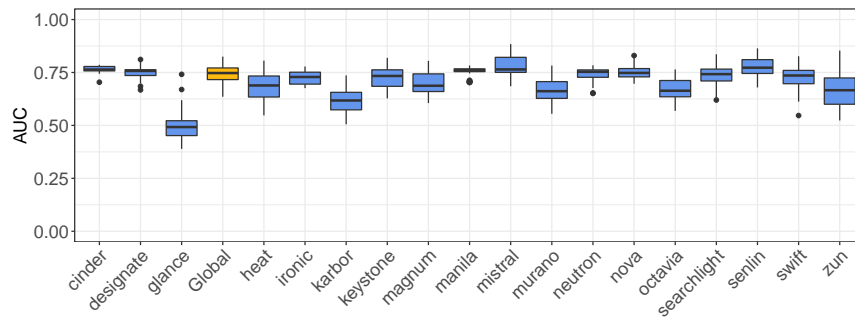


Figure A.12: OpenStack - Qinling

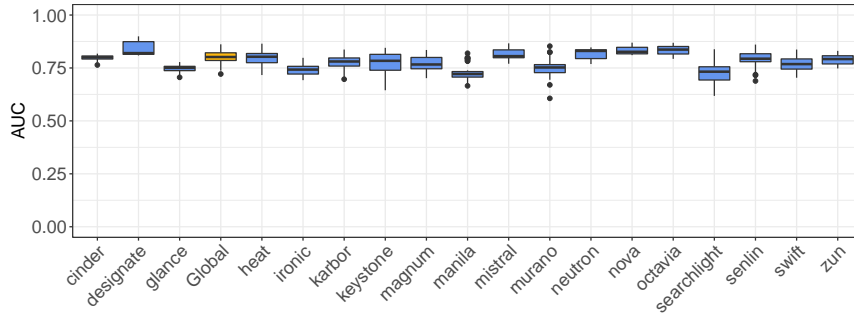


Figure A.13: OpenStack - Solum

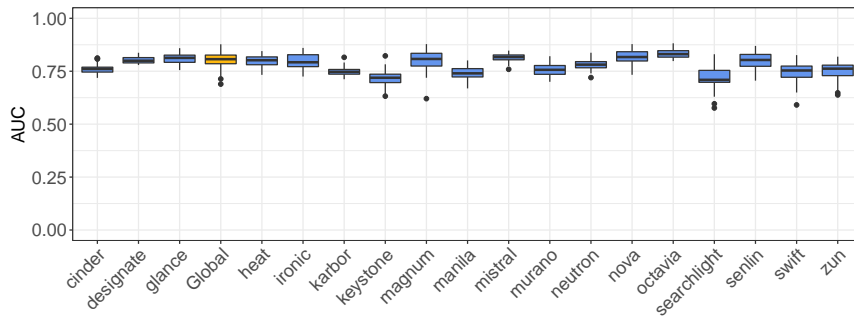


Figure A.14: OpenStack - Zaqr

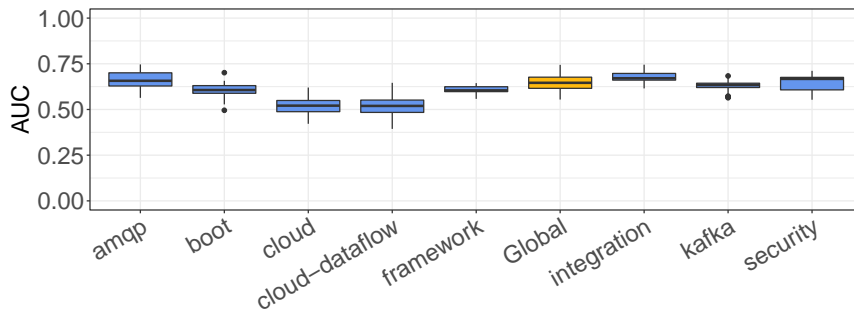


Figure A.15: Spring - Batch

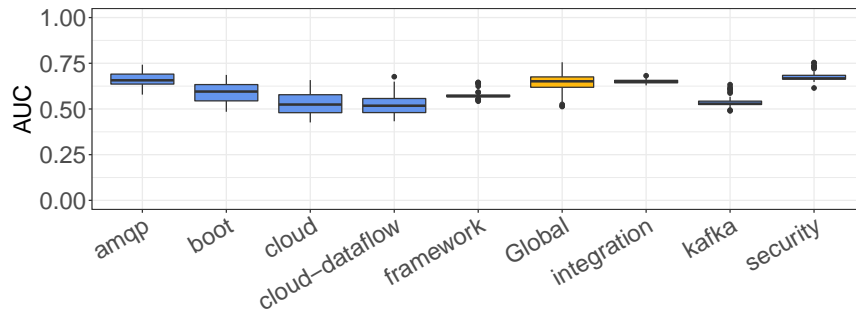


Figure A.16: Spring - Cloud-Commons

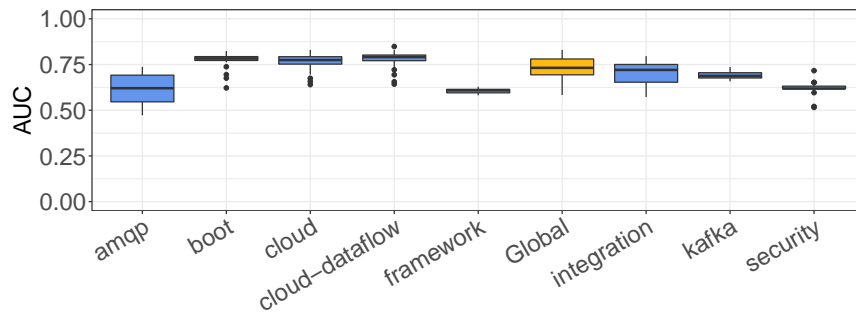


Figure A.17: Spring - Data-Commons

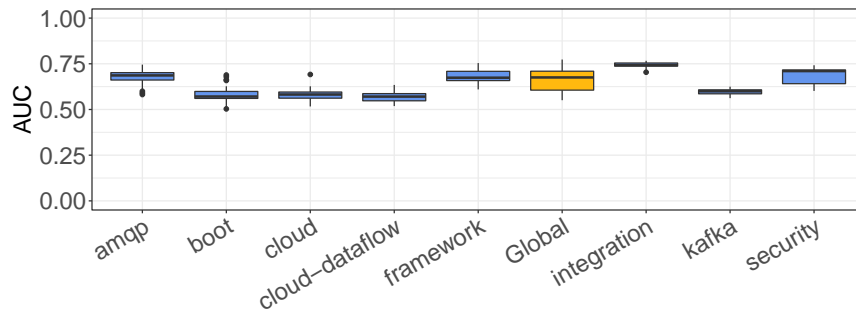


Figure A.18: Spring - Ldap

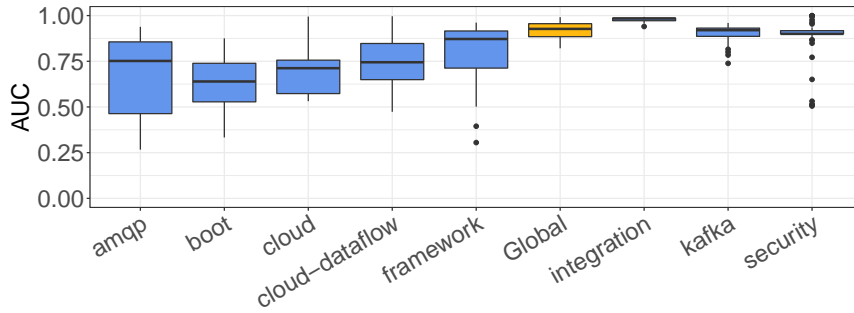


Figure A.19: Spring - Roo

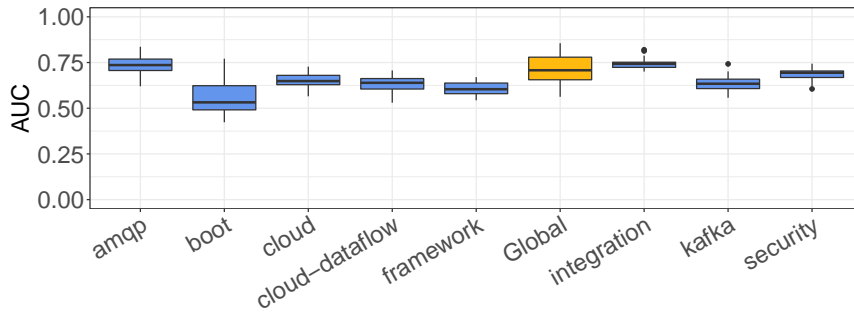


Figure A.20: Spring - Session

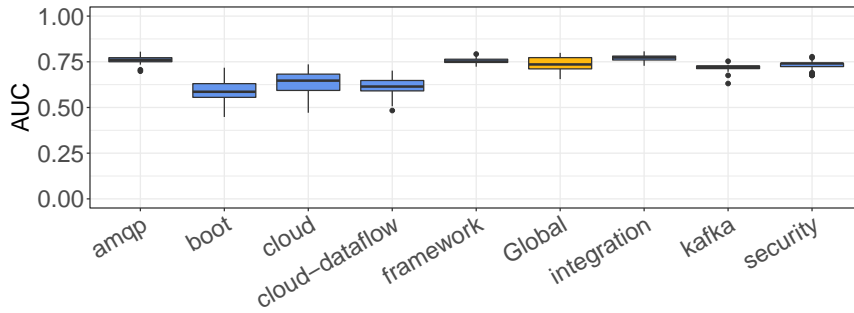


Figure A.21: Spring - Statemachine

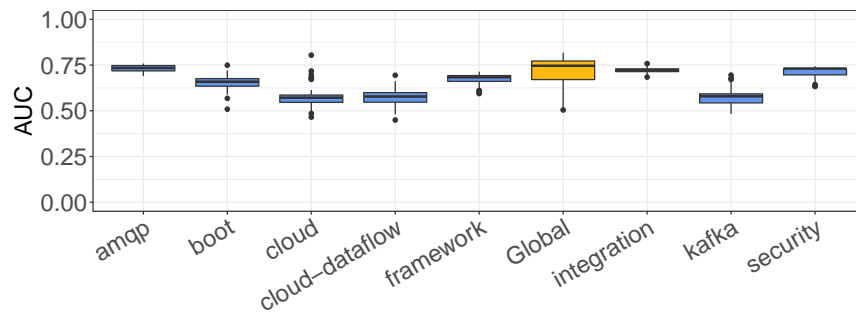


Figure A.22: Spring - Vault

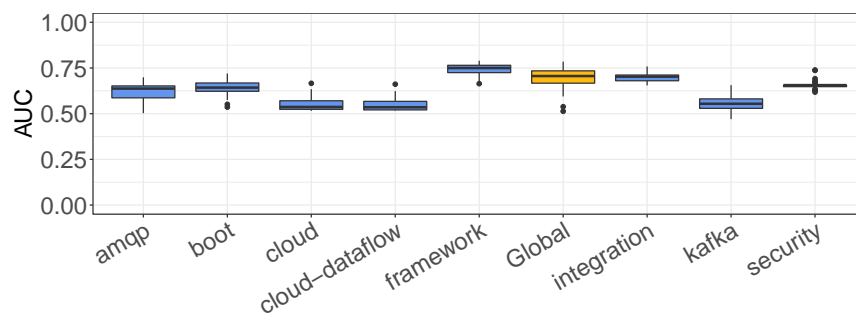


Figure A.23: Spring - Ws

Similarly, the following figures show the comparison between the Brier Scores of the peer-local models and the global model on data-lacking components for OpenStack and Spring projects.

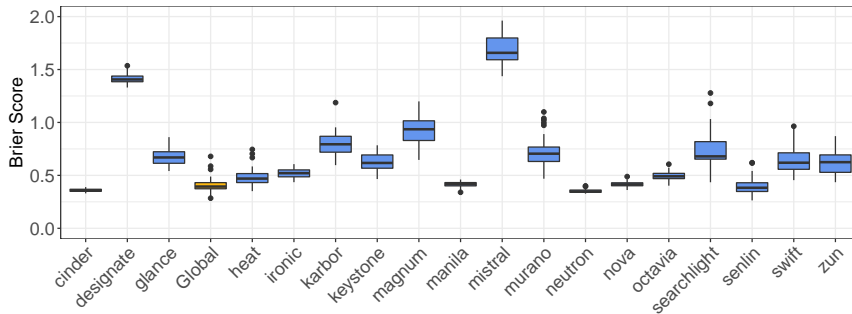


Figure A.24: OpenStack - Blazar

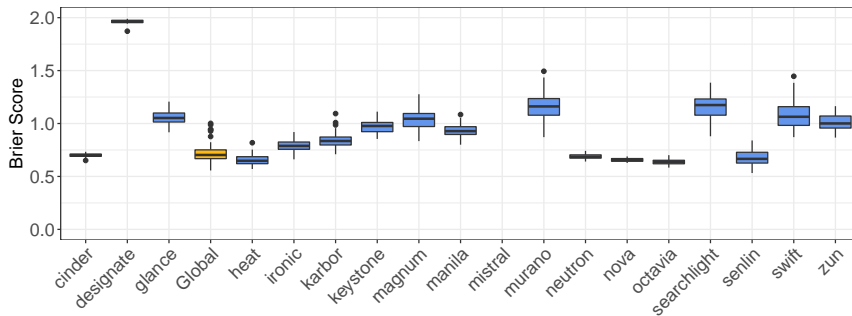


Figure A.25: OpenStack - Cyborg

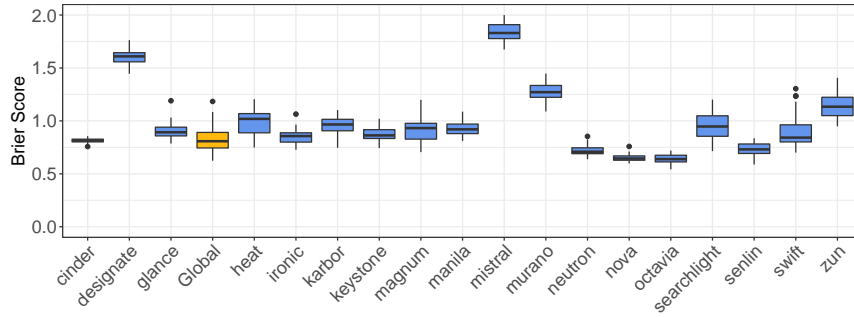


Figure A.26: OpenStack - ec2-api

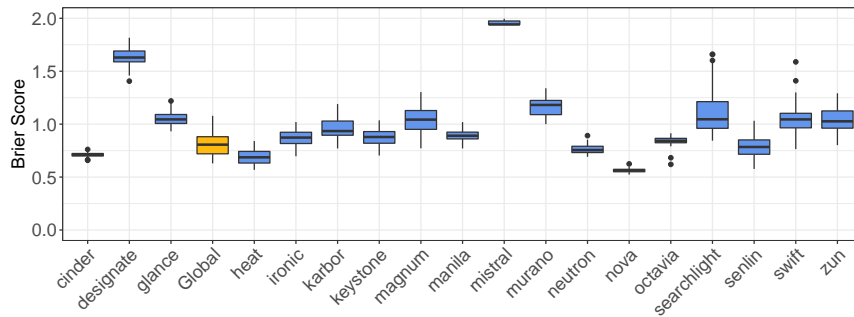


Figure A.27: OpenStack - Horizon

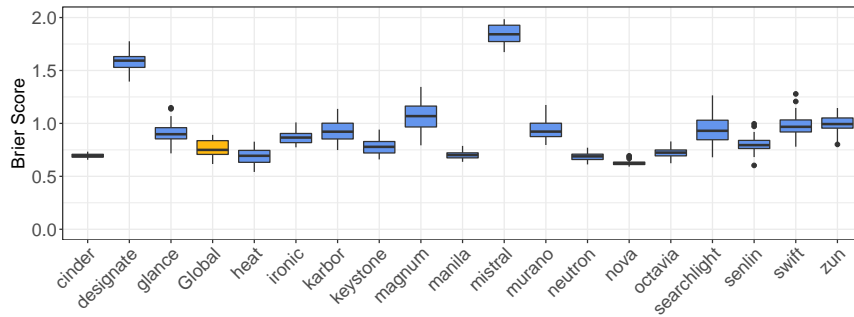


Figure A.28: OpenStack - Masakari

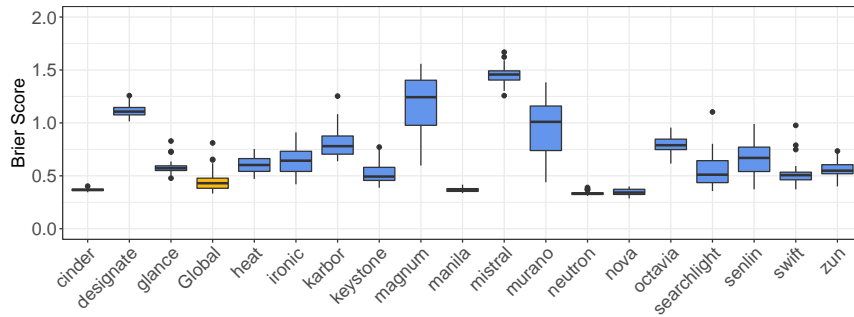


Figure A.29: OpenStack - Placement

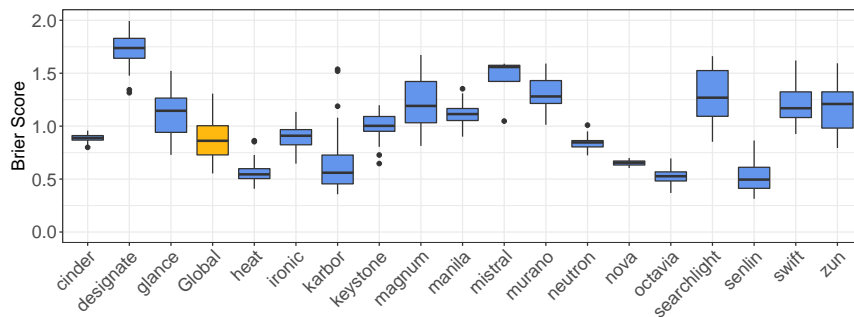


Figure A.30: OpenStack - Qinqing

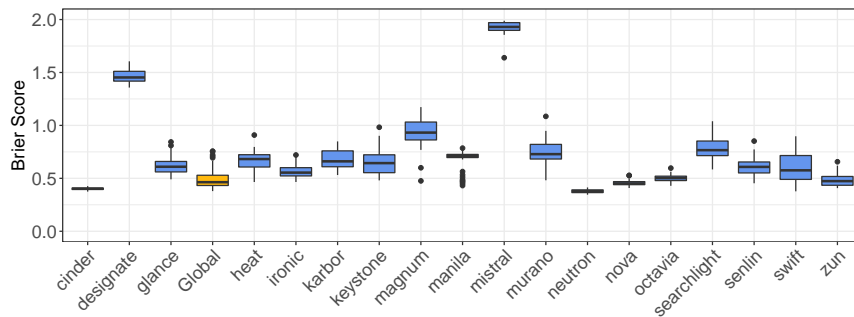


Figure A.31: OpenStack - Solum

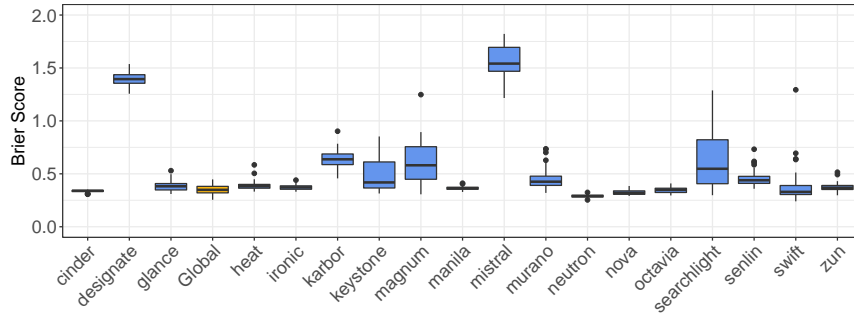


Figure A.32: OpenStack - Zaqr

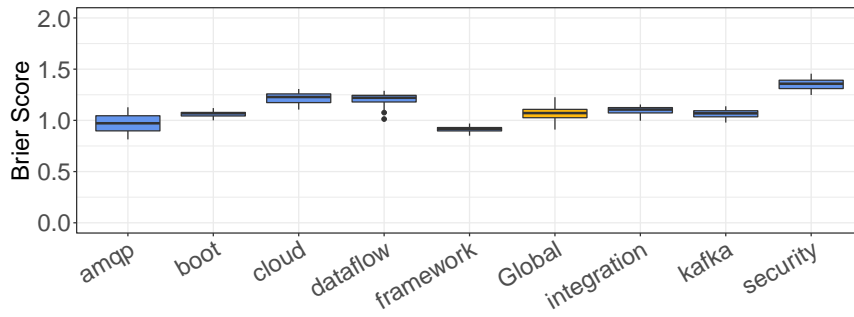


Figure A.33: Spring - Batch

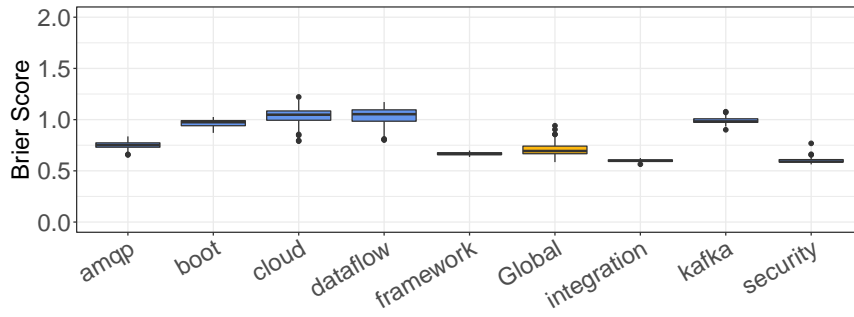


Figure A.34: Spring - Cloud-Commons

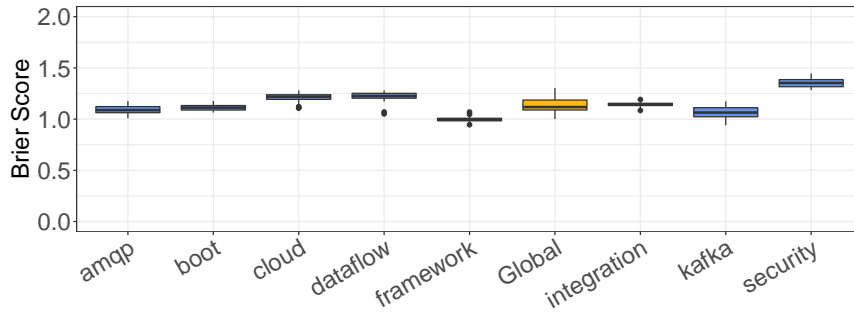


Figure A.35: Spring - Data-Commons

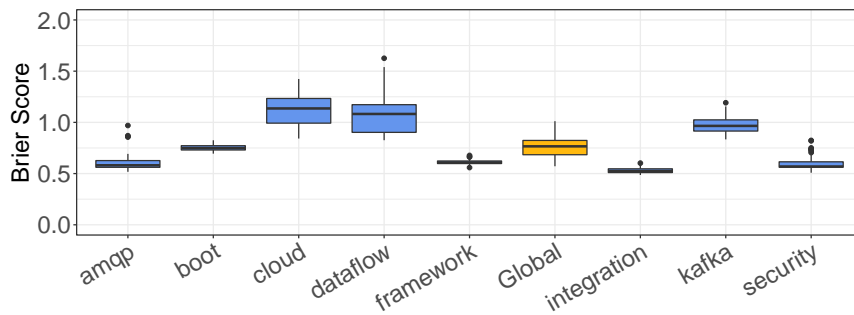


Figure A.36: Spring - Ldap

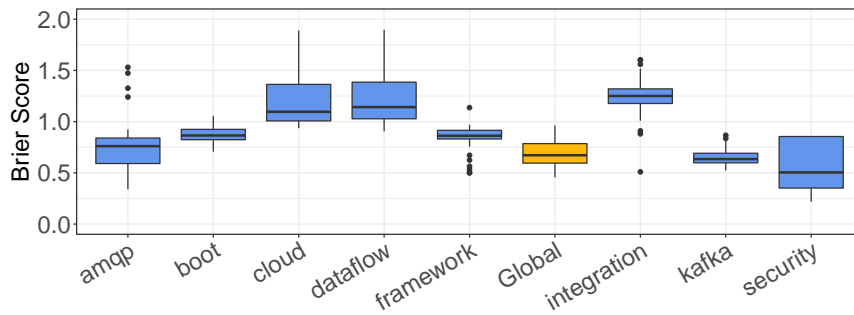


Figure A.37: Spring - Roo

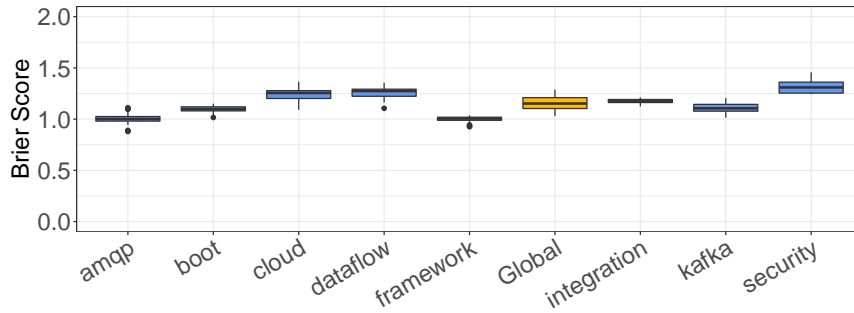


Figure A.38: Spring - Session

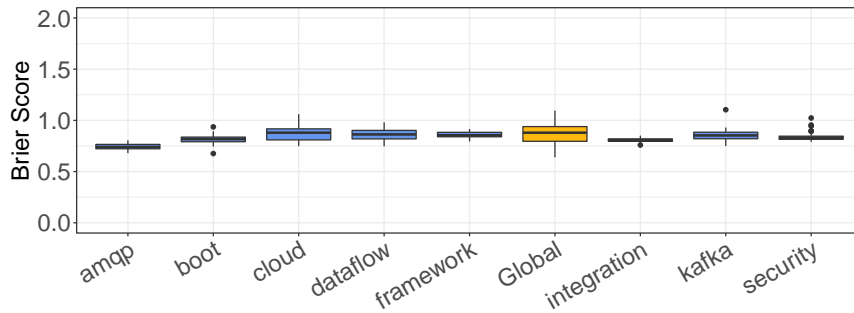


Figure A.39: Spring - Statemachine

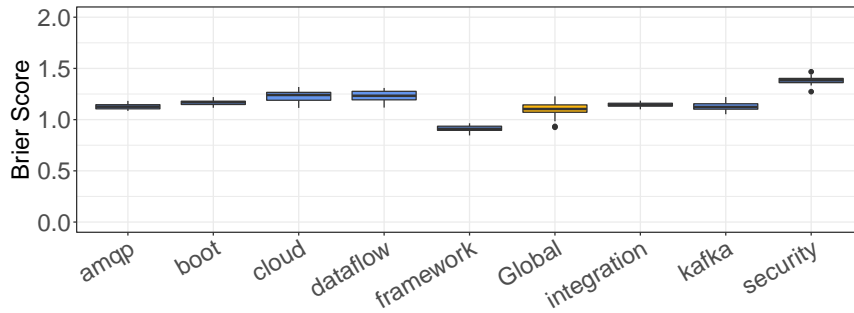


Figure A.40: Spring - Vault

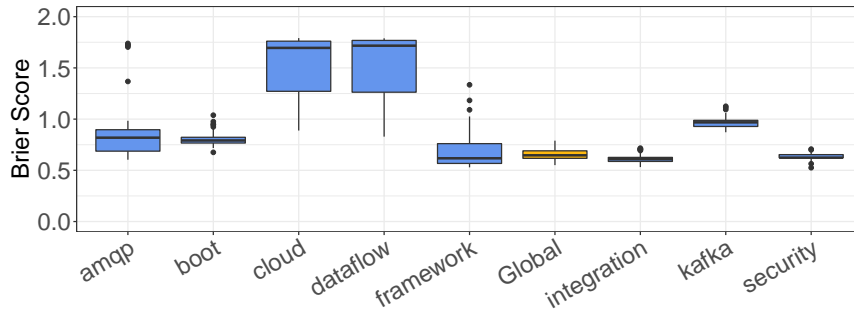


Figure A.41: Spring - Ws

The following figure shows the correlation between the rankings of peer-local models for Spring project:

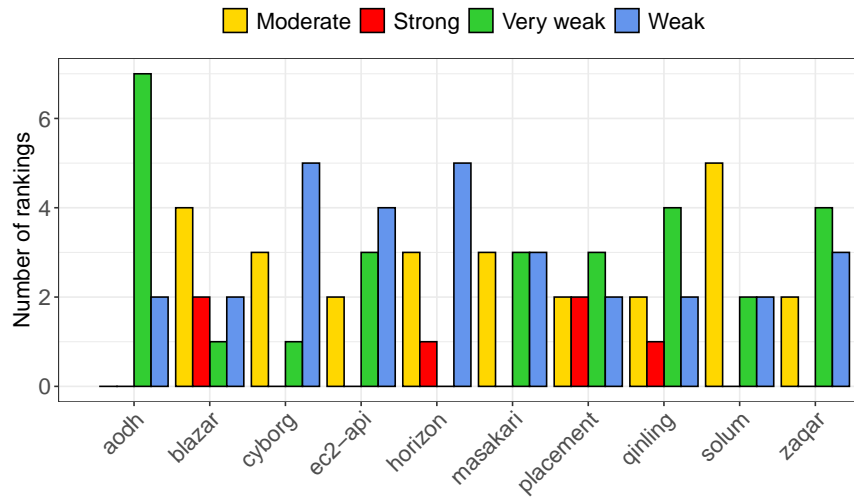


Figure A.42: Level of Correlation between the rankings of best performing peer-local models across the data lacking components of OpenStack.

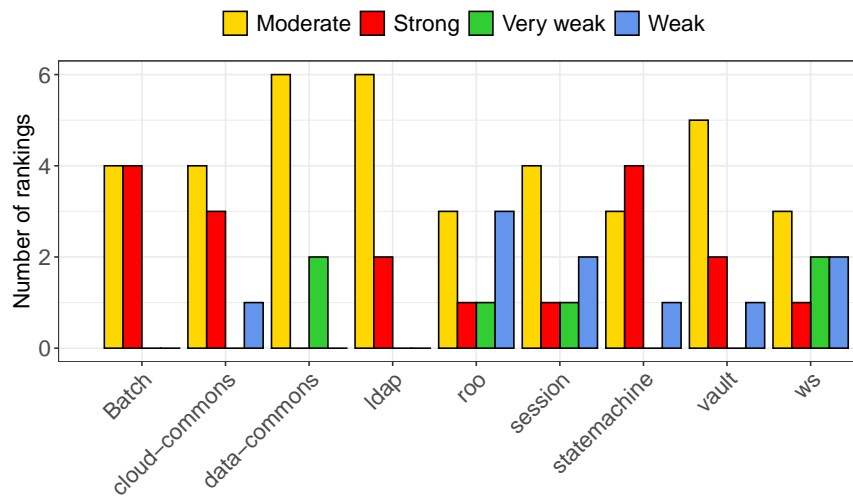


Figure A.43: Level of Correlation between the rankings of best performing peer-local models across the data lacking components of Spring.

A.4 Feature importance across components

Tables [A.1](#), [A.2](#), [A.5](#), [A.3](#) and [A.4](#) show the ranking of the most important features for deciding the logging level on the components level.

Features	Global	HDFS	MapReduce	Yarn	Common	Tools
Static length	8	5	9	10	9	7
Var number	10	7	8	11	-	-
Bloc SLOC	12	9	-	-	-	9
Try bloc	-	-	-	-	-	-
Catch bloc	1	1	1	1	2	2
If bloc	3	4	3	5	4	5
Switch bloc	-	-	-	-	-	-
For bloc	12	15	-	13	-	10
While bloc	-	-	-	-	-	-
Log density	5	6	5	9	5	4
nb log stmt	-	13	-	-	7	-
Avg static length	9	16	10	7	6	-
Avg var number	7	10	6	6	8	6
Avg logging level	2	3	4	4	3	3
file SLOC	-	-	-	-	-	-
McCabe	-	-	-	-	-	-
Fan In	11	11	11	-	-	8
Code churn	-	12	-	-	-	-
log change	6	8	7	3	-	-
log churn ratio	11	14	-	8	-	-
nb of revisions in hist	-	-	-	-	-	-
code churn in hist	-	17	-	-	-	-
log churn in hist	13	-	-	12	-	-
log churn ratio in hist	-	-	-	-	-	-
log changing revs in hist	-	-	-	-	-	-
Tokens	4	2	2	2	1	1

Table A.1: Important features rankings for Hadoop's local and global models

Features	Global	amqp	boot	cloud	cloud-dataflow	framework	integration	kafka	security
Static length	3	4	2	-	-	4	5	5	4
Var number	12	7	5	5	5	7	10	4	-
Bloc SLOC	6	-	9	-	-	5	12	-	-
Try bloc	-	-	-	-	-	-	-	-	-
Catch bloc	2	1	-	-	5	3	2	-	3
If bloc	3	2	7	-	-	4	4	-	5
Switch bloc	9	-	-	-	-	-	-	-	-
For bloc	15	7	9	-	-	-	-	-	-
While bloc	-	-	-	-	-	-	-	-	-
Log density	4	8	5	2	2	2	7	2	2
nb log stmt	10	-	6	-	-	6	10	-	-
Avg static length	6	7	6	7	6	5	3	2	-
Avg var number	13	-	3	3	4	4	8	4	3
Avg logging level	1	2	1	1	1	1	1	1	1
file SLOC	7	-	6	-	-	5	10	-	5
McCabe	-	-	-	-	-	-	-	-	-
Fan In	10	7	6	6	7	7	12	-	-
Code churn	16	-	-	5	5	7	-	-	-
log change	13	8	6	7	-	-	11	5	-
log churn ratio	5	6	10	7	-	-	8	4	-
nb of revisions in hist	13	-	8	-	7	-	10	-	-
code churn in hist	-	-	-	-	7	7	-	-	-
log churn in hist	9	8	9	7	7	7	12	5	5
log churn ratio in hist	9	-	10	-	-	-	12	6	-
log changing revs in hist	10	8	7	6	6	7	6	-	-
Tokens	6	3	4	4	3	6	9	3	-

Table A.2: Important features rankings for Spring's local and global models

Feature	Nbgrader	Gateway	Notebook	Global
Static length	-	2	3	4
Var number	-	-	7	8
Bloc SLOC	-	-	4	7
Try bloc	-	-	-	-
Catch bloc	2	4	-	3
If bloc	-	-	-	-
Switch bloc	-	-	-	-
For bloc	5	-	-	-
While bloc	-	-	-	-
Log density	-	-	-	9
nb.log.stmnt	-	6	-	-
Avg static length	6	5	5	6
Avg var number	-	-	-	-
Avg logging level	1	1	1	1
File SLOC	-	-	-	5
McCabe	-	-	-	-
Fan In	-	-	-	-
Code churn	3	-	6	-
log change	-	7	-	-
log churn ratio	-	-	-	-
nb of revisions in hist	-	-	-	-
code churn in hist	-	-	-	-
log churn in hist	-	-	-	-
log churn ratio in hist	-	-	-	-
log changing revs in hist	-	-	-	-
Tokens	4	3	2	2

Table A.3: Important features rankings for Jupyter's local and global models

Feature	Apm agnet	Diagnostics	Elasticsearch-Core	Elasticsearch-Hadoop	Global
Static length	6	-	8	4	-
Var number	-	10	5	7	6
Bloc SLOC	9	3	6	5	8
Try bloc	-	-	-	-	-
Catch bloc	2	1	1	3	1
If bloc	4	-	-	9	4
Switch bloc	-	-	-	-	-
For bloc	8	-	-	-	-
While bloc	-	5	-	-	-
Log density	10	7	10	-	9
nb_log_stmt	-	6	9	-	-
Avg static length	-	-	11	6	5
Avg var number	-	-	7	-	7
Avg logging level	1	4	2	2	2
File SLOC	-	8	-	-	-
McCabe	-	-	-	-	-
Fan In	5	-	-	8	-
Code churn	7	11	4	-	-
log change	-	-	-	-	-
log churn ratio	-	9	-	-	-
nb of revisions in hist	-	-	12	-	-
code churn in hist	-	-	-	-	-
log churn in hist	-	12	-	-	-
log churn ratio in hist	-	-	-	-	-
log changing revs in hist	-	-	-	-	-
Tokens	3	2	3	1	3

Table A.4: Important features rankings for Elasticsearch's local and global models

Features	Global	nova	ironic	swift	neutron	keystone	zun	cinder	manila	octavia	designate	glance	karbor	searchlight	heat	senlin	mistral	magnum	murano
Static length	9	6	5	5	6	-	8	8	-	-	4	5	3	6	3	-	-	7	-
Var number	5	-	6	10	5	4	9	7	-	-	3	6	4	5	5	-	-	6	-
Bloc SLOC	11	10	-	-	-	-	7	12	10	10	-	-	8	10	6	7	-	4	-
Try bloc	-	-	8	-	-	-	-	-	-	-	-	10	-	-	-	-	-	-	5
Catch bloc	3	3	3	3	3	3	1	3	3	-	-	4	-	-	4	3	-	-	3
If bloc	13	-	4	9	9	-	6	6	6	-	-	9	-	3	-	6	-	-	6
Switch bloc	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
For bloc	7	7	-	8	10	5	-	5	9	5	-	12	-	-	-	-	-	8	-
While bloc	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Log density	10	-	-	6	7	-	-	-	-	7	-	13	-	9	7	5	-	-	-
nb log srmt	-	-	-	7	-	7	-	-	-	12	-	-	-	-	-	-	4	-	11
Avg static length	8	-	-	-	13	-	-	10	4	9	-	8	9	-	9	-	5	3	7
Avg var number	-	-	7	-	14	-	-	13	7	4	8	7	-	-	8	-	-	5	-
Avg logging level	1	1	2	2	1	2	2	2	2	1	1	1	2	2	2	2	1	2	2
file SLOC	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	10
McCabe	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Fan In	14	8	-	4	-	8	-	14	11	-	7	-	-	-	-	4	-	-	9
Code churn	-	-	9	-	11	9	4	-	8	8	6	-	-	7	11	-	7	-	8
log change	4	5	-	-	4	-	5	11	5	3	-	11	5	-	10	8	6	-	4
log churn ratio	6	4	-	-	8	-	-	4	-	11	5	3	-	4	-	-	-	-	-
nb of revisions in hist	-	-	-	-	-	-	-	-	-	-	-	-	7	-	-	-	-	-	-
code churn in hist	-	-	-	-	-	-	-	-	-	-	-	-	6	-	-	-	-	-	-
log churn in hist	12	9	-	-	-	6	-	9	-	6	-	-	-	8	-	-	3	-	-
log churn ratio in hist	-	-	-	-	12	-	-	-	-	-	-	-	-	-	-	-	-	-	-
log changing revs in hist	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Tokens	2	2	1	1	2	1	3	1	1	2	2	2	1	1	1	1	2	1	1

Table A.5: Important features rankings for OpenStack's local and global models