An empirical study on log level prediction for multi-component systems

Youssef Esseddiq Ouatiti, Mohammed Sayagh, Noureddine Kerzazi and Ahmed E. Hassan

Abstract—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 log level, which is leveraged to adjust the verbosity of logs so that only important log messages are traced. Deciding for the log 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 log 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 paper, we quantify the impact of the particularities of each component of a multi-component system on the performance and interpretability of the log level prediction model of prior work. We observe that the performance of the log 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 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 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 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.

Index Terms—Software logging, log level prediction, Multi-component systems, Machine learning

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 log 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, as shown in Figure 1. The logs provide valuable information for different stakeholders, such as software operators to spot abnormal execution [18, 35, 46], developers to diagnose failures and to better maintain a software system [18, 24, 48], and release managers to assess deployment activities [2, 4, 44].

However, assigning the appropriate log level for a logging statement is an important activity [19, 32]. For instance, the lack of logging causes a lack of information about the runtime and a reduced ability for diagnosis [47]. Logging too much, however, causes system overhead and makes logs full of noisy data and challenging to analyse [45]. Thus, coupled with the framework log level (works as a threshold), log 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

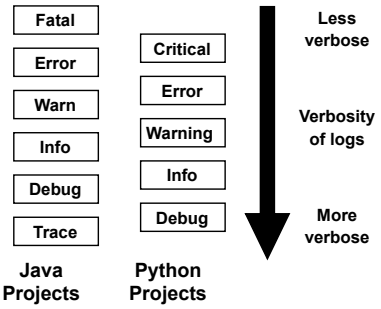


Fig. 1: Log levels verbosity in Java (Left) and Python (Right) projects

the verbosity level at the "warn" level, only the logging statements with the "warn" level or less verbose log level ("error" and "fatal") would be traced in a log file. Identifying the appropriate log level for a logging statement is a challenging task [31, 47], since the developers cannot be sure how the code will be used in later stages of the development activity [31]. Consequently, developers spend much effort adjusting the log levels of their logging statements [47].

The logging practices (e.g., the length of a log message, the variables to trace) as well as choosing the appropriate log 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 log levels as the logs were growing fast without any execution anomaly [1]. That was the opposite for the 'Nova' component, which aimed at increasing the verbosity to have detailed monitoring [2].

While an ordinal regression machine learning model to predict the log level for a new logging statement was proposed by Li et al. [19], 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 log level (shown in Figure 1). The metrics that were used by Li et al. consider five dimensions; i.e., logging statement metrics (e.g., number of variables), containing bloc metrics (e.g., number 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 paper the impact of the variation between different components on the performance and interpretation of the log level prediction model of Li et al. [19], so practitioners can better understand how to leverage that model to predict log 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 paper is to quantify the differences between the performance and interpretability of these two types of models. As prior studies [6, 25, 34] suggest using simple and interpretable models over complex ones, we focus on the ordinal regression model that is proposed by Li et al. [19], 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 global models 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 models (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 models 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 log 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. [19] 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 log 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,

1. https://github.com/openstack/swift/pull/15
2. https://bugs.launchpad.net/nova/+bug/1715785

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 on deciding for log levels for a given component.

The paper is structured as follows. Section 2 discusses related research. Section 3 covers the methodology. Section 4 presents our findings. Section 5 discusses the threats to validity. Finally, Section 6 concludes the paper.

2 Related Work

As this paper investigates how to use machine learning for the logging practices of multi-component systems, the closest work to our paper is related to logging practices, the application of machine learning techniques to software logging, and the use of cross-modeling approaches. The following subsections discuss each of these research directions.

2.1 Investigating the Logging Practices

A large body of research efforts [5, 11, 15, 18, 21, 22, 27, 32, 35, 48] focus on improving and automating logging practices. For instance, Fu et al. [5] leveraged a mixed qualitative and quantitative study to automatically determine where (i.e., in which area of the code) to log. Yuan et al. [48] developed an approach that supports the diagnosis of software failures by automatically adding casually-related information to log statements. Hassani et al. [11] suggest the need for automated tools for early detection of log-related issues. Li et al. [21] manually studied duplicate logging statements and were able to detect five types of logging smells. Pecchia et al. [32] pointed out that developers belonging to different product lines log in a similar style but for different purposes. Li et al. [18] conducted a qualitative study to analyze the benefits and costs of logging. Locke et al. [27] introduced LogAssist, a tool designed for log analysis that leverages n-gram modeling to give an organized view of logs.

Our study is different from this line of research. Rather than investigating the logging practices, we evaluate how to better leverage machine learning techniques for predicting the log level in the context of multi-component systems.

2.2 Leveraging Machine Learning for Logging

Another line of research leverages machine learning techniques to assist practitioners in their logging activities [17, 19, 20, 23, 26]. For example, Liu et al. [26] built a model that ranks the source code variables that are susceptible to be logged. Li et al. [17] used automatically extracted code snippet topics to get an idea of what events are more likely to be logged. Li et al. [20] trained random forest models to predict if a committed code change would require a log change. Recently, Li et al. [23] proposed a deep learning approach to handle log level prediction relying on information about the context of the log statement and the log message itself. Finally. Li et al. [19] used an ordinal regression model to predict log level for newly introduced log statements using metrics covering the log statement, its containing bloc, its containing file and change history of the file.

While this line of research leverages machine learning techniques to assist developers in their logging decisions, our work investigates how to use such models in the context of multi-component systems. Our paper starts with predicting the log level for a new log statement, while we motivate future work to replicate our study on further logging practices (e.g., predicting the variables to log).

2.3 Cross-modeling Approaches

Another line of research focuses on the evaluation of cross-modeling approaches [1, 29, 30, 43, 49, 50]. These approaches consist of training a model by leveraging data from one project and testing that model on another software project. For example, Li et al. [19] trained a log level prediction model on a combination of N-1 projects and evaluated that model on the remaining project. Similarly, Xia et al. [43] evaluated the cross-modeling approach for predicting build co-changes. Both of these last studies found that a model trained on a given project is less performant on other projects. To improve the cross-modeling approaches, Zimmermann et al. [50] proposed an approach that recommends which cross-project model to use for a given project. Zhang et al. [49] and Ni et al. [30] compared the performance of cross-project supervised models with within project unsupervised models for defect prediction.

While these studies focus on cross-project modeling, no prior work investigates how a model that is trained on a multi-component project performs on each of its components before evaluating it on other projects. Thus, our work complements this line of research by investigating how models' performances and interpretability change from one to another component of a multi-component system.

The closest work to our study is the work of Battenberg et al. [1] and Menzies et al. [29], both of which found that local models are more promising than global models. However, a local model for Menzies et al. [29] is a model that is trained on similar projects. Our study is different than Menzies et al. [29] as we evaluate how a project's model performs within that project itself. A local model for Buttenburg et al. [1] is a model that is trained on a cluster of similar data. Basically, they cluster the data of a project into a number of clusters for each of which they train a separate local model. However, this study has three limitations: (1) their approach requires a manual definition of the number of clusters, which might impact the performance of local models. (2) Their evaluation did not consider clusters with few data points. (3) Finally, they did not consider the multi-component architecture.

3 Methodology

The goal of this paper is to investigate the usage of the log level prediction models in the context of multi-component

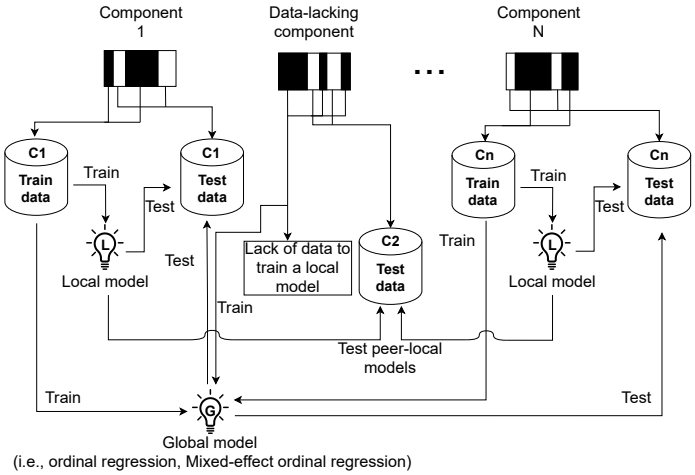


Fig. 2: An illustration of the models that we evaluated in our study according to data from different components. Note that each of these experiments are repeated 100 times using different bootstrap dataset for the training/testing.

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. [19]. That model leverages data from all the components to train a single global model (as shown in Figure 2). 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 2. 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 [9]. 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 2. 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 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 with 4 components, Hadoop is a distributed computing system with 5 components, Spring is a project that offers a large number of services for Java developers and has 17 components, and OpenStack is a cloud computing platform that has 28 components.

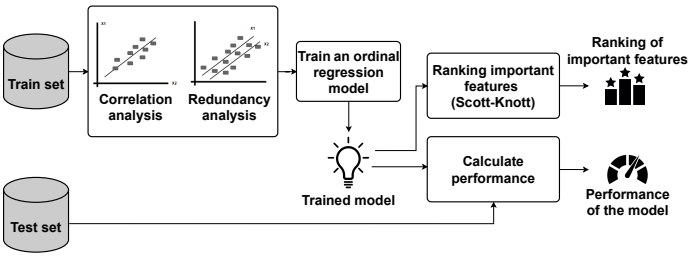


Fig. 3: An overview of our training and testing methodology

3.1 Modeling approach

In this paper, we evaluate ordinal regression models for log level prediction in the context of multi-component systems. In particular, we followed the approach summarized in Figure 3 to train and test each model, including the global models (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:

3.1.1 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 [14, 38] 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 [14]. 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 [16, 28]. 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.

3.1.2 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 paper, we chose the default 0.9 as a cutoff threshold, similarly to prior work [16, 39].

3.1.3 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 boostrap 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 AUC[3] [8], Brier Score [4] [42], and the AICc [5] [13] 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) [12]. 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 log 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 [16, 40, 41]. 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 appear in the bootstrap sample. We repeat the process of training and testing for 100 generated bootstrap samples.

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

4. Brier 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.

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

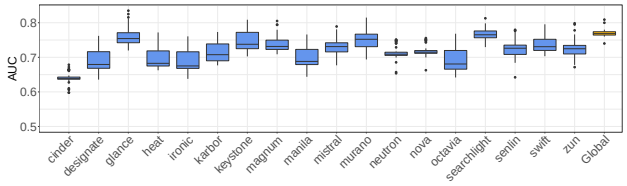


Fig. 4: The AUC performance of the global model on components and on the OpenStack project

3.1.4 Feature Importance

We used Wald's χ^2 to evaluate the impact of each metric on predicting the log level similarly to prior work [16, 41]. 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 [37] clustering technique similarly to previous studies [14, 33]. This technique uses an iterative process to generate a ranking of groups of features according to their importance.

4 Results

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 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. [19], we train the global model using a bootstrap sample of observations from all the components. We then test that model on two separate datasets; a) **global testing**: using the remaining observations that belong to the whole project out of the training bootstrap sample, and b) **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 3.1 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 for OpenStack and Figure 10 in Appendix A for our other studied projects. That indicates that the performance obtained in the prior study [19] 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 11 in Appendix B. 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; α = 0.01) between the distribution of the dependent feature (i.e., log 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 log 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 the global model on 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.

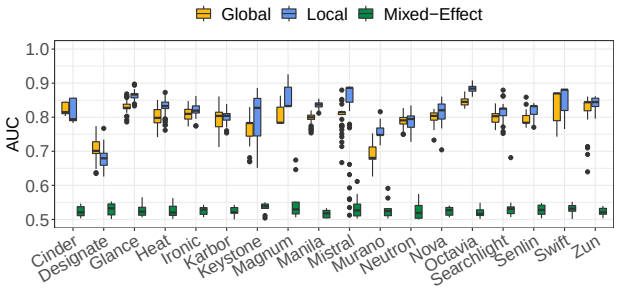


Fig. 5: AUC scores of the global, local and mixed-effect models on the OpenStack components

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 3.1. 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 global models and the mixed-effect models 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, α = 0.01) outperform their respective global models in terms of AUC, as shown in Figure 5 for OpenStack and Figure 12 in Appendix B 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 models, and only 20% (1 out of 5), 12% (1 out of 8) and 5% (1 out of 18) of the Hadoop, Spring and OpenStack local models show a statistically significantly lower AUC performances compared to the global models.

Similarly, we observe that local models show better Brier Score and AICc compared to the global models for all the components, as shown in Figures 6 and 7 for OpenStack and Figures 13 and 14 in Appendix B 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 models for Hadoop, Spring, OpenStack, Jupyter and Elasticsearch. On top of that, we observe that 3 of Spring global models reach a median Brier Score that is worse than a random guess (BS ≥ 0.8).

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 models do

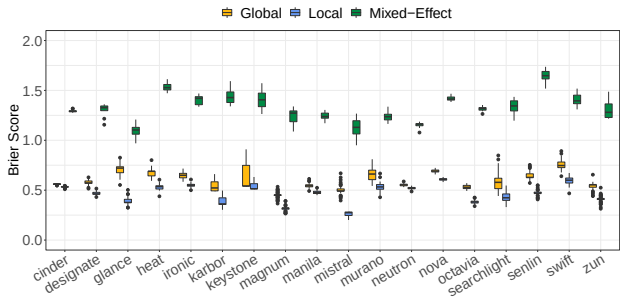


Fig. 6: Brier Score of the global, local and mixed-effect models on the OpenStack project

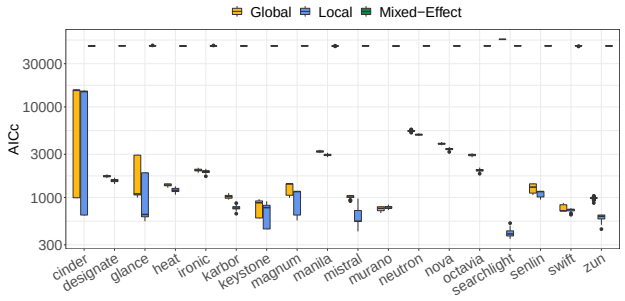


Fig. 7: Corrected AIC of the global, local and mixed-effect models on the OpenStack project

not perform well for any of our performance metrics, as shown in Figures 5, 6, and 7 for OpenStack and Figures 12, 13, and 14 in Appendix B 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 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 log 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 log 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 [log 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 log 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 et al. [35] indicates 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 [3, 36]. Consequently, according to Shang et al. [35] 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 [3, 7, 10], 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 global models 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 global model on more than one performance metric. **Our results suggest the use of local models instead of global models.**

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 log 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.

6. https://github.com/openstack/trove/commit/cf7694f0dbf9b5f81057f00c35cfe626f22a71ec
7. https://github.com/openstack/trove/commit/cf7694f0dbf9b5f81057f00c35cfe626f22a71ec

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 8 and 9 for one of OpenStack (Aodh) data-lacking components. The other data-lacking components figures can be found in Appendix C. 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 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.

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

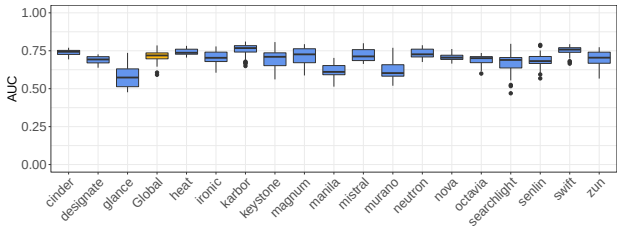


Fig. 8: The AUC performance of the global and peer-local models on the Aodh component

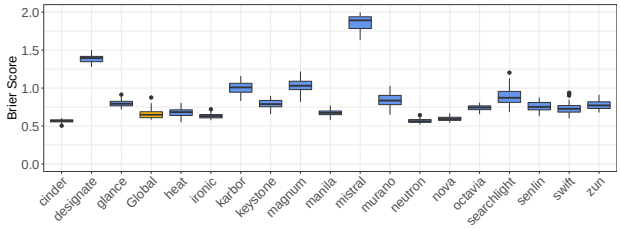


Fig. 9: The Brier Score performance of the global and peer-local models on the Aodh component

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 51 and 52 in Appendix C. 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 log 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).**

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 [19] investigated the interpretability (i.e., the most important features) for the models that predict the log 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 3.1.4. 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 log 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 log 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 log levels.

Results: We observe a low similarity between local models and global models 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 models' 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 1, 2, 5, 3 and 4 in Appendix D. 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 log 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 log 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 and local models can have contradictory effects on the prediction of the appropriate log 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 log level between the global models 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* log 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* log 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* log 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* log 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 log 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 log 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.**

5 Threats to Validity

5.1 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 log 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 log level for log 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.

5.2 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 log 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 log level prediction models.

6 Conclusion

Since the identification of the appropriate log level for a new log statement is challenging, prior studies leveraged machine learning models (e.g., ordinal regression models) to predict the appropriate log level for a new logging statement. However, these prior studies do not consider the particularities of multi-component 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 paper, we quantify the impact of the variation between different components on the performance and interpretability of log 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 global models. 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 the global models as they 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.

References

[1] N. Bettenburg, M. Nagappan, and A. Hassan. Think locally, act globally: Improving defect and effort prediction models. In *2012 9th IEEE Working Conf. on Mining Software Repositories*, pages 60–69.

[2] Y. Chen, Q. Lin, and Q. Lin. Outage prediction and diagnosis for cloud service systems. In *Proc. of the WWW'19*, 2019.

[3] M. D'Ambros, M. Lanza, and R. Robbes. Evaluating defect prediction approaches: A benchmark and an extensive comparison. *Empirical Software Engineering*, pages 1–47, 2012.

[4] N. El-Sayed, H. Zhu, and B. Schroeder. from failure across multiple clusters: A trace-driven approach to understanding, predicting, and mitigating job terminations. In *Proc. of the 37th Int. Conf. on Distributed Computing Systems*, pages 1333–1344, 2017.

[5] Q. Fu, J. Zhu, W. Hu, J. Lou, R. Ding, Q. Lin, D. Zhang, and T. Xie. Where do developers log? an empirical study on logging practices in industry. In *Proc. of the 36th Int. Conf. on Software Engineering*, page 24–33, 2014.

[6] W. Fu and T. Menzies. Easy over hard: A case study on deep learning. page 49–60, 2017.

[7] T. Graves, A. Karr, J. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, pages 653–661, 2000.

[8] J. Hanley and B. Mcneil. The meaning and use of the area under a receiver operating characteristic (roc) curve. *Radiology*, 143:29–36, 1982.

[9] F. E. Harrell. *Regression Modeling Strategies*. Springer Int. Publishing, 2001.

[10] A. E. Hassan. Predicting faults using the complexity of code changes. In *Proc. of the 31st Int. Conf. on Software Engineering*, pages 78–88, 2009.

[11] M. Hassani, W. Shang, E. Shihab, and N. Tsantalis. Studying and detecting log-related issues. *Empirical Software Engineering*, 2018.

[12] C. M. Hurvich and C. Tsai. A corrected akaike information criterion for vector autoregressive model selection. *Journal of Time Series Analysis*, pages 271–279, 1993.

[13] C. M. HURVICH and C.-L. TSAI. Regression and time series model selection in small samples. *Biometrika*, pages 297–307, 1989.

[14] C. Jiarpakdee, J.and Tantithamthavorn and A. Hassan. The impact of correlated metrics on the interpretation of defect models. *IEEE Transactions on Software Engineering*, pages 320–331, 2019.

[15] S. Kabinna, W. Shang, C. Bezemer, and A. E. Hassan. Examining the stabity of logging statements. In *Proc. of the 23rd Int. Conf. on Software Analysis, Evolution, and Reengineering*, pages 326–337, 2016.

[16] D. Lee, G. Rajbahadur, D. Lin, M. Sayagh, C. Bezemer, and A. Hassan. An empirical study of the characteristics of popular minecraft mods. *Empirical Software Engineering*, 2020.

[17] H. Li, T. Chen, W. Shang, and A. Hassan. Studying software logging using topic models. *Empirical Software Enggineering*, page 2655–2694, 2018.

[18] H. Li, W. Shang, B. Adams, M. Sayagh, and A. E. Hassan. A qualitative study of the benefits and costs of logging from developers' perspectives. *IEEE Transactions on Software Engineering*, pages 1–1, 2020.

[19] H. Li, W. Shang, and A. Hassan. Which log level should developers choose for a new logging statement? *Empirical Software Engineering*, page 1684–1716, 2017.

[20] H. Li, W. Shang, Y. Zou, and A. Hassan. Towards just-in-time suggestions for log changes. *Empirical Software Engineering*, 2017.

[21] Z. Li, T. Chen, J. Yang, and W. Shang. Dlfinder: Characterizing and detecting duplicate logging code smells. In *Proc. of the 41st Int. Conf. on Software Engineering*, pages 152–163, 2019.

[22] Z. Li, T. Chen, J. Yang, and W. Shang. Studying duplicate logging statements and their relationships with code clones. *IEEE Transactions on Software Engineering*, pages 1–1, 2021.

[23] Z. Li, H. Li, T. Chen, and W. Shang. Deeplv: Suggesting log levels using ordinal based neural networks. In *Proc. of the 43rd Int. Conf. on Software Engineering*, pages 1461–1472, 2021.

[24] Q. Lin, K. Hsieh, Y. Dang, H. Zhang, K. Sui, Y. Xu, J. Lou, C. Li, Y. Wu, R. Yao, M. Chintalapati, and D. Zhang. Predicting node failure in cloud service systems. In *Proc. of the 26th ACM Joint Meeting on European Software Engineering Conf. and Symposium on the Foundations of Software Engineering*, page 480–490, 2018.

[25] Z. Liu, X. Xia, A. Hassan, D. Lo, Z. Xing, and X. Wang. Neural-machine-translation-based commit message generation: How far are we? In *Proc. of the 33rd ACM/IEEE Int. Conf. on Automated Software Engineering*, page 373–384, 2018.

[26] Z. Liu, X. Xia, D. Lo, Z. Xing, A. E. Hassan, and S. Li. Which variables should i log? *IEEE Transactions on Software Engineering*, pages 1–1, 2019.

[27] S. Locke, H. Li, T. Chen, W. Shang, and W. Liu. Logassist: Assisting log analysis through log summarization. *IEEE Transactions on Software Engineering*, pages 1–1, 2021.

[28] Y. A. B. McIntosh, S. Kamei and H. A. E. The impact of code review coverage and code review participation on software quality. In *Proc. of the Working Conf. on Mining Software Repositories*, page 292–201, 2014.

[29] T. Menzies, A. Butcher, D. Cok, A. Marcus, L. Layman, F. Shull, B. Turhan, and T. Zimmermann. Local versus global lessons for defect prediction and effort estimation. *IEEE Transactions on Software Engineering*, pages 822–834, 2013.

[30] c. Ni, X. Xia, D. Lo, X. Chen, and Q. Gu. Revisiting supervised and unsupervised methods for effort-aware cross-project defect prediction. *IEEE Transactions on Software Engineering*, pages 1–1, 2020.

[31] A. Oliner, A. Ganapathi, and W. Xu. Advances and challenges in log analysis. *Communications of the ACM*, page 55–61, 2012.

[32] A. Pecchia, M. Cinque, G. Carrozza, and D. Cotroneo. Industry practices and event logging: Assessment of a critical software development process. In *Proc. of the 37th Int. Conf. on Software Engineering*, pages 169–178, 2015.

[33] G. Rajbahadur, G. Oliva, A. Hassan, and J. Dingel. Pitfalls analyzer: Quality control for model-driven data science pipelines. In *2019 ACM/IEEE 22nd Int. Conf. on Model Driven Engineering Languages and Systems*, pages 12–22, 2019.

[34] C. Rudin. Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead, 2019.

[35] W. Shang, M. Nagappan, and A. Hassan. Studying the relationship between logging characteristics and the code quality of platform software. *Empirical Software Engineering*, page 1–27, 2015.

[36] E. Shihab, Z. Jiang, W. Ibrahim, B. Adams, and A. Has-

san. Understanding the impact of code and process metrics on post-release defects: A case study on the eclipse project. 2010.

[37] C. Tantithamthavorn. *ScottKnottESD: The Scott-Knott Effect Size Difference (ESD) Test*, 2018.

[38] C. Tantithamthavorn and A. Hassan. An experience report on defect modelling in practice: Pitfalls and challenges. In *Proc. of the Int. Conf. on Software Engineering: Software Engineering in Practice Track*, 2018.

[39] C. Tantithamthavorn, A. Hassan, and K. Matsumoto. The impact of class rebalancing techniques on the performance and interpretation of defect prediction models. *IEEE Transactions on Software Engineering*, pages 1200–1219, 2020.

[40] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto. The impact of automated parameter optimization on defect prediction models. *IEEE Transactions on Software Engineering*, pages 1–1, 2018.

[41] P. Thongtanunam and A. Hassan. Review dynamics and its impact on software quality. *IEEE Transactions on Software Engineering*, pages 1–13, 2018.

[42] D. Wilks. *Statistical methods in the atmospheric sciences*, volume 100. 2011.

[43] X. Xia, D. Lo, S. McIntosh, E. Shihab, and A. Hassan. Cross-project build co-change prediction. In *2015 IEEE 22nd Int. Conf. on Software Analysis, Evolution, and Reengineering*, pages 311–320, 2015.

[44] Y. Xu, K. Sui, R. Yao, H. Zhang, Q. Lin, Y. Dang, P. Li, K. Jiang, W. Zhang, J. Lou, M. Chintalapati, and D. Zhang. 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, 2018.

[45] D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. U. Jain, and M. Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *Proc. of the 11th Conf. on Operating Systems Design and Implementation, Systems Design and Implementation*, pages 249–265, 2014.

[46] D. Yuan, S. Park, P. Huang, Y. Liu, M. M. Lee, X. Tang, Y. Zhou, and S. Savage. Be conservative: Enhancing failure diagnosis with proactive logging. In *Proc. of the 10th USENIX Conf. on Operating Systems Design and Implementation*, page 293–306, 2012.

[47] D. Yuan, S. Park, and Y. Zhou. Characterizing logging practices in open-source software. In *Proc. of the 34th Int. Conf. on Software Engineering*, pages 102–112, 2012.

[48] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage. Improving software diagnosability via log enhancement. In *Proc. of the Sixteenth Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, page 3–14, 2011.

[49] F. Zhang, Q. Zheng, Y. Zou, and A. Hassan. Cross-project defect prediction using a connectivity-based unsupervised classifier. In *2016 IEEE/ACM 38th Int. Conf. on Software Engineering*, pages 309–320, 2016.

[50] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy. Cross-project defect prediction a large scale experiment on data vs. domain vs. process. In *ESEC/FSE '09*, pages 91–100, 2009.

Youssef Esseddiq Ouatiti is a PhD student at the School of Computing of Queen's University in Canada, under supervision of Prof. Ahmed E. Hassan. His research at the Software Analysis and Intelligence Lab (SAIL) focuses on empirical studies about the usage of Machine learning within software systems. He obtained his enngineering degree in Data analytics from (ENSIAS) Mohammed V University - Rabat in 2020.

Mohammed Sayagh is an assistant professor at ETS (Quebec University). Before that, he was a postdoctoral fellow at the Software Analysis and Intelligence Lab (SAIL) at Queen's University, under the supervision of Prof. Ahmed E. Hassan. He obtained his PhD from the Lab on Maintenance, Construction, and Intelligence of Software (MCIS) at Ecole Polytechnique Montreal (Canada), under the supervision of Prof. Bram Adams. His research interests include empirical software engineering, multi-component software systems, and software configuration engineering. More details about his work are available on "http://msayagh.github.io".

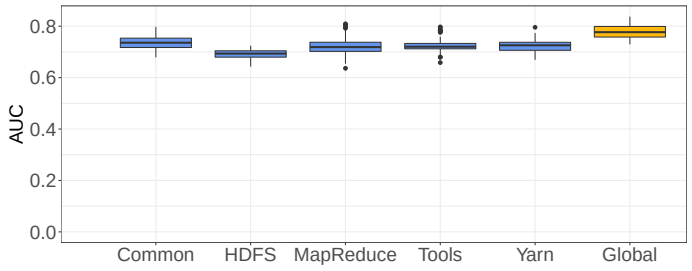
Noureddine Kerzazi is an assistant professor at (ENSIAS) Mohammed V University, Morocco. His research interests include improvement of software processes and practices, Continuous Integration/Delivery, Branching strategies, Post-release failures Analysis, and software team members' Coordination. Noureddine obtained his PhD degree in computer science engineering from Polytechnique Montreal in 2010. He has been Integrator Release Manager at an online payment processor for more than three years. He leads development of DSL4SPM, a conceptual tool for software process modeling.

Ahmed E. Hassan is a Canada Research Chair in Software Analytics and the NSERC/Blackberry Industrial Research Chair with the School of Computing, Queens University, Kingston, ON, Canada. His industrial experience includes helping architect the Blackberry wireless platform, and working for IBM Research at the Almaden Research Lab and the Computer Research Lab at Nortel Networks. Early tools and techniques developed by his team are already integrated into products used by millions of users worldwide. He is the named inventor of patents at several jurisdictions around the world including the United States, Europe, India, Canada, and Japan. Dr. Hassan serves on the editorial board of the IEEE Transactions on Software Engineering, the Journal of Empirical Software Engineering, and PeerJ Computer Science. He spearheaded the organization and creation of the Mining Software Repositories (MSR) conference and its research community.

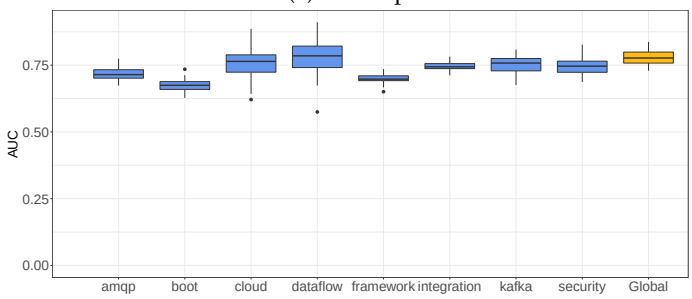
Appendix A

Performance of the global model on components

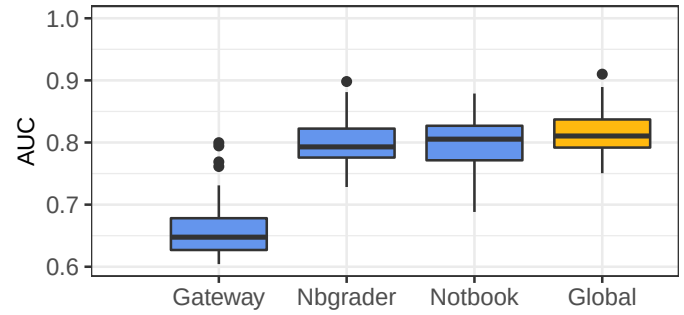
Figures 10 and 11 show the AUC and Brier Score performance of the global model trained using data from the entire projects on the components of these projects.



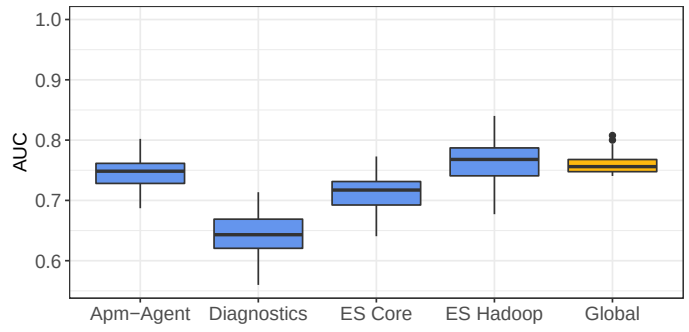
(a) Hadoop



(b) Spring

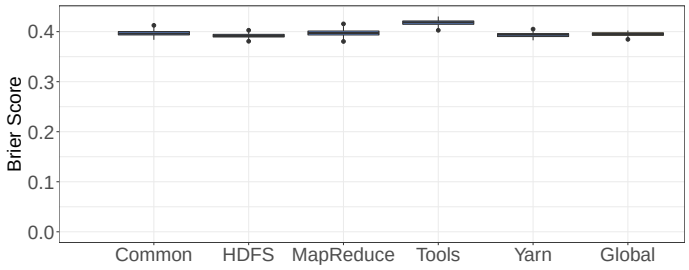


(c) Jupyter

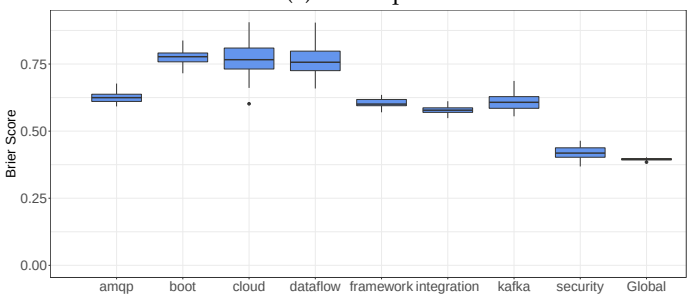


(d) Elasticsearch

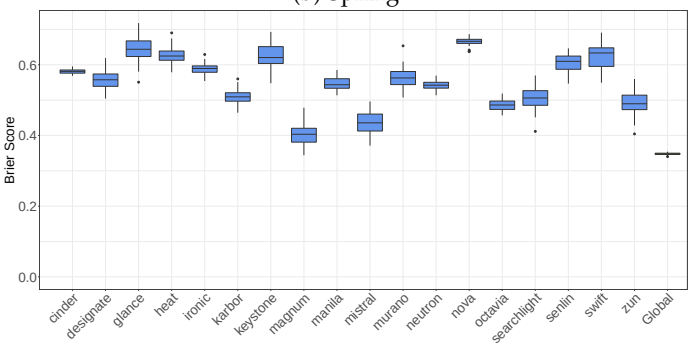
Fig. 10: The AUC performance of the global model on components and on the entire project



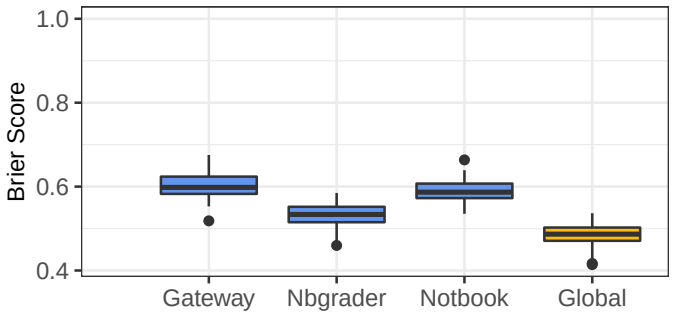
(a) Hadoop



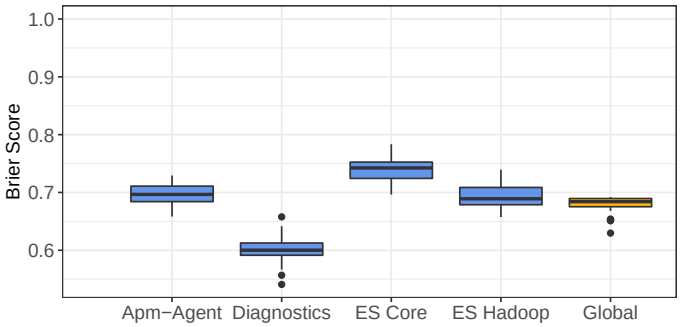
(b) Spring



(c) OpenStack



(d) Jupyter



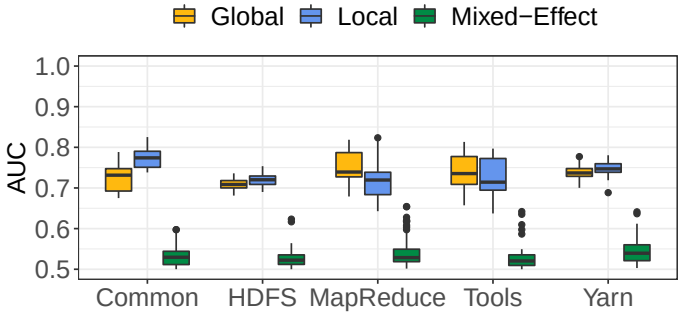
(e) Elasticsearch

Fig. 11: The Brier Score performance of the global model on different components and on the entire project

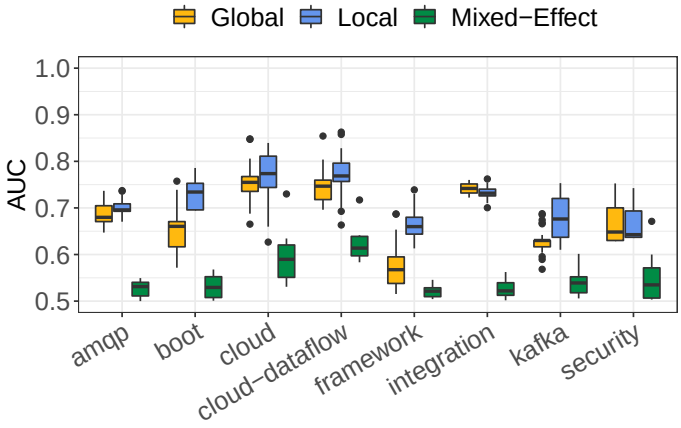
Appendix B

Local models Vs. Global models

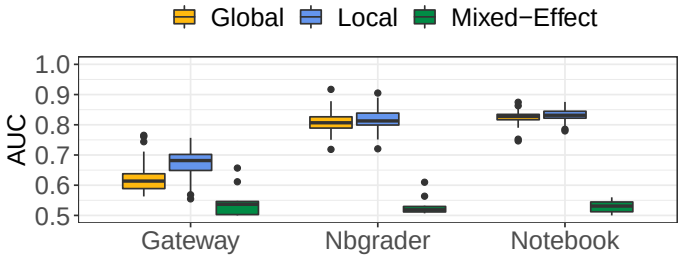
Figures 12, 13 and 14 show the comparison between the performances and quality of fit of the global, local and mixed effect models.



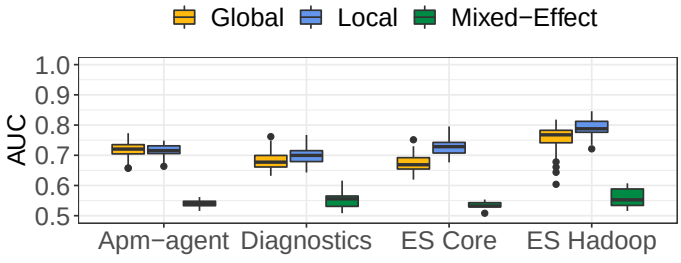
(a) Hadoop



(b) Spring

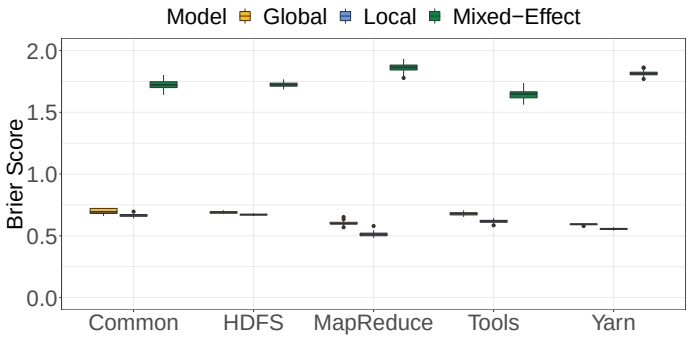


(c) Jupyter

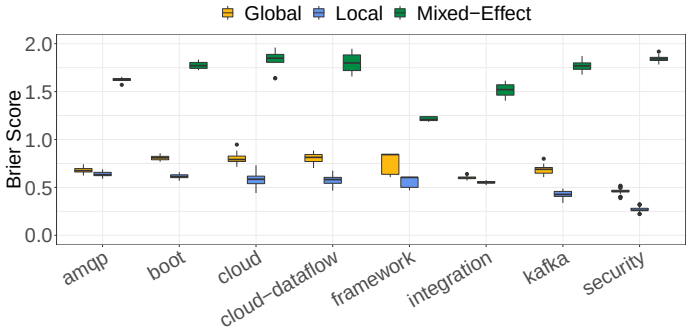


(d) Elasticsearch

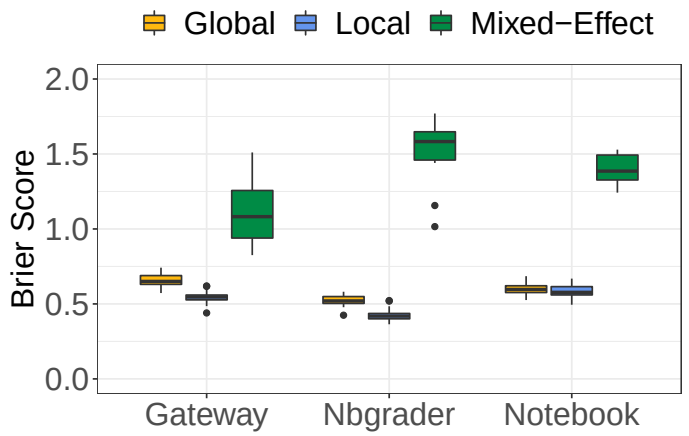
Fig. 12: AUC scores of the global, local and mixed-effect models



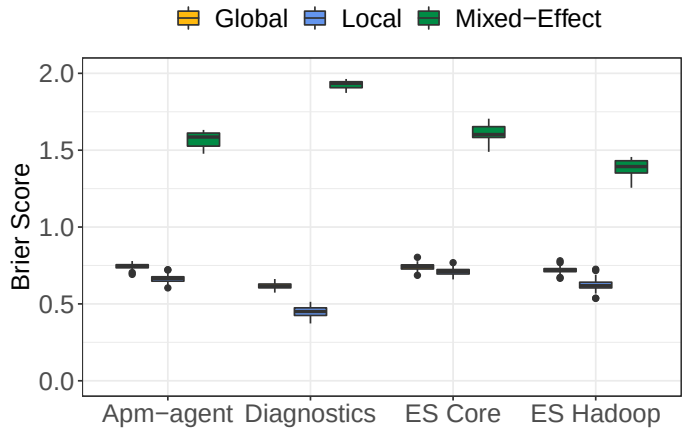
(a) Hadoop



(b) Spring

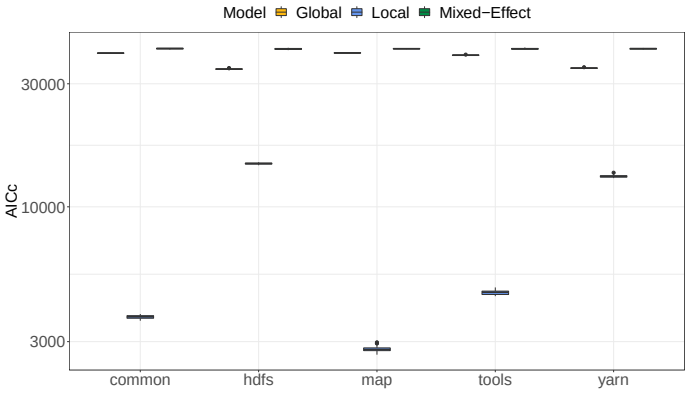


(c) Jupyter

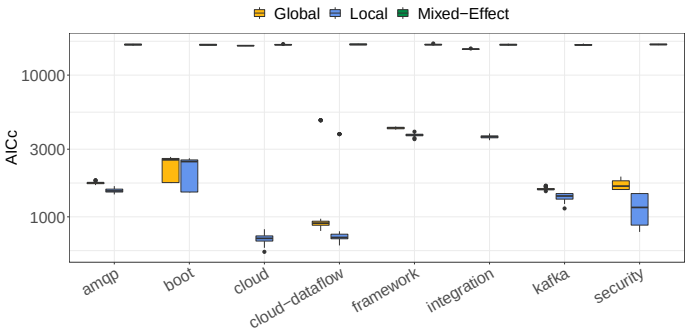


(d) Elasticsearch

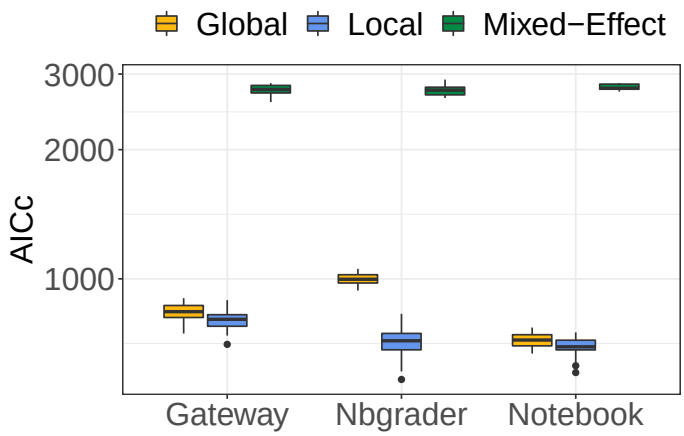
Fig. 13: Brier scores of the global, local and mixed-effect models



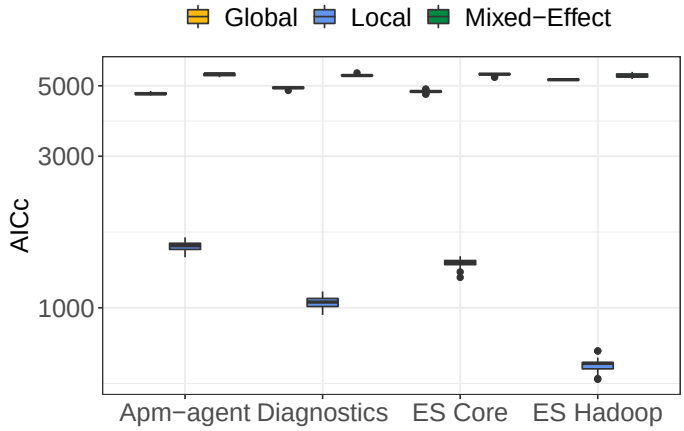
(a) Hadoop



(b) Spring



(c) Jupyter



(d) Elasticsearch

Fig. 14: AICc scores of the global, local and mixed-effect models (log-scale)

Appendix C

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.

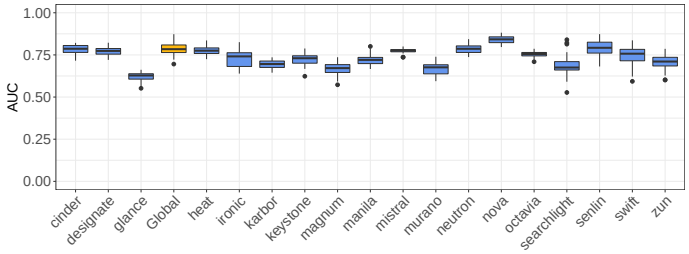


Fig. 15: OpenStack - Blazar

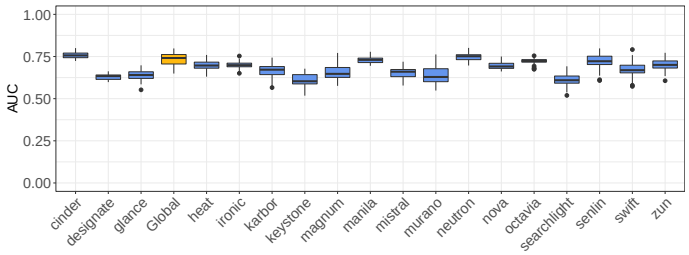


Fig. 16: OpenStack - Cyborg

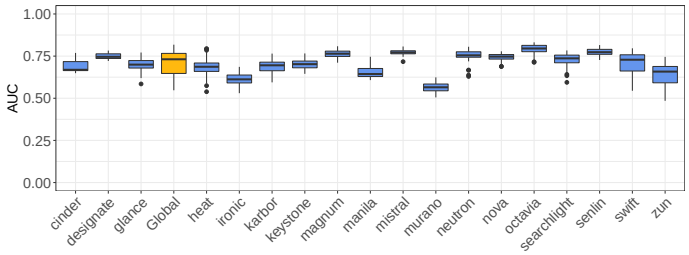


Fig. 17: OpenStack - ec2-api

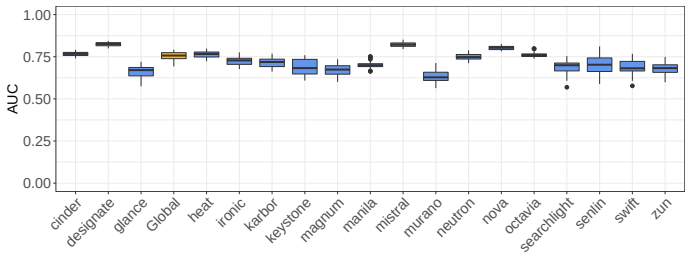


Fig. 18: OpenStack - Horizon

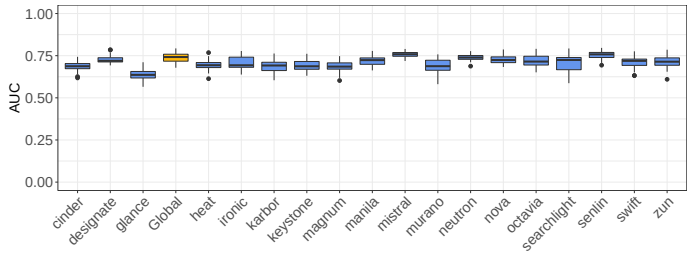


Fig. 19: OpenStack - Masakari

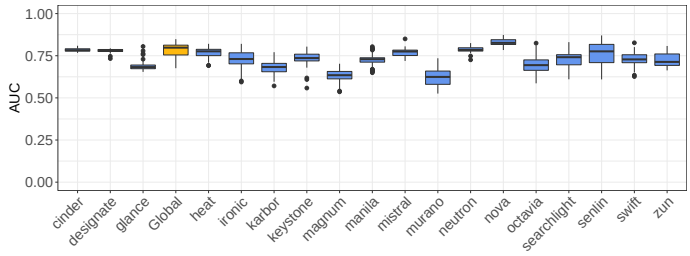


Fig. 20: OpenStack - Placement

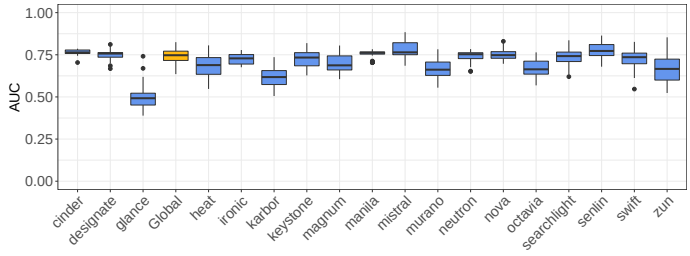


Fig. 21: OpenStack - Qinling

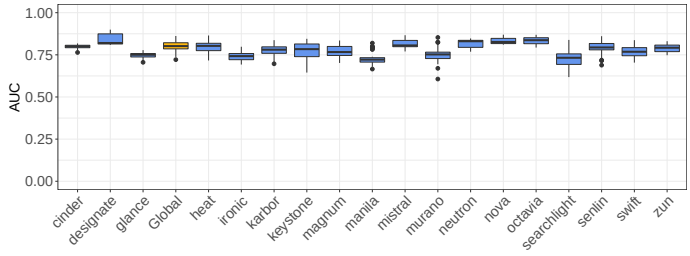


Fig. 22: OpenStack - Solum

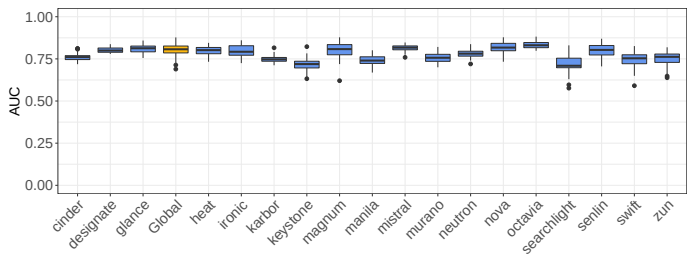


Fig. 23: OpenStack - Zaqar

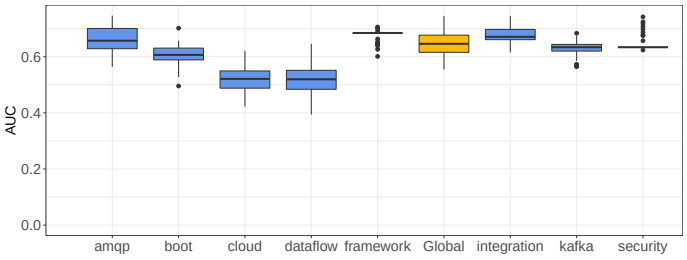


Fig. 24: Spring - Batch

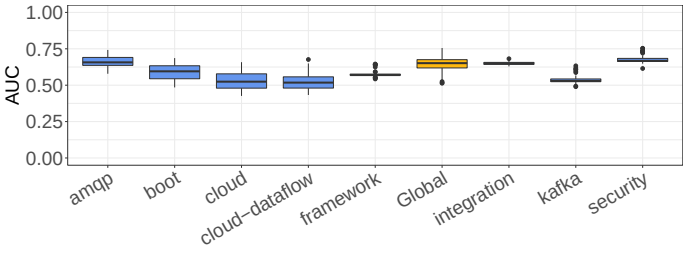


Fig. 25: Spring - Cloud-Commons

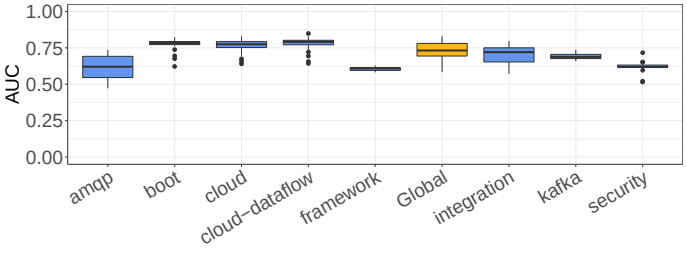


Fig. 26: Spring - Data-Commons

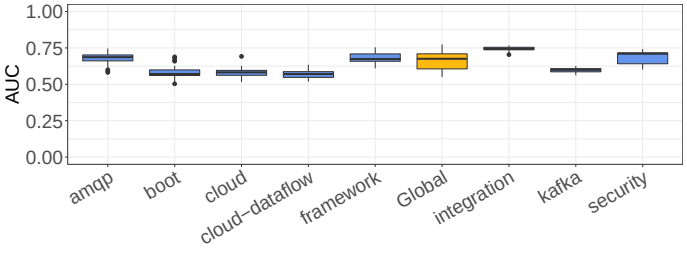


Fig. 27: Spring - Ldap

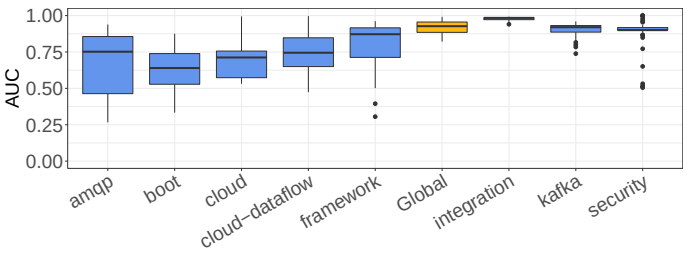


Fig. 28: Spring - Roo

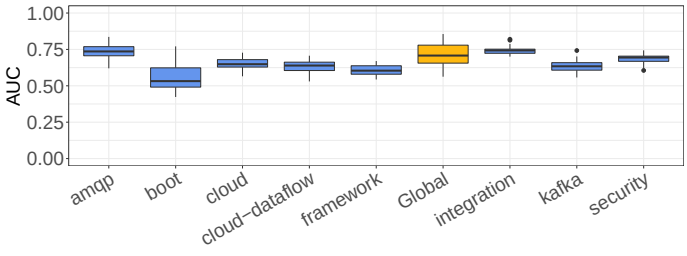


Fig. 29: Spring - Session

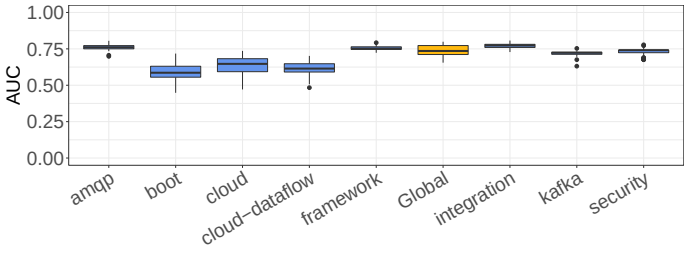


Fig. 30: Spring - Statemachine

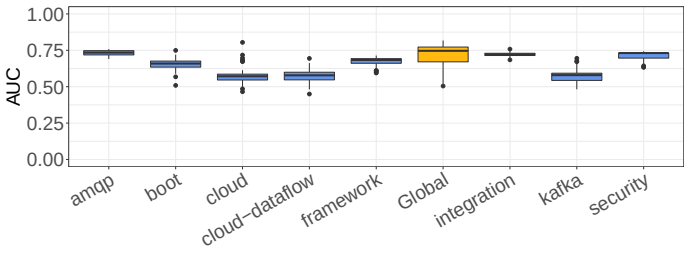


Fig. 31: Spring - Vault

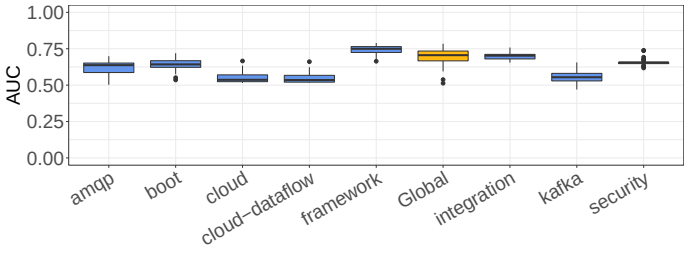


Fig. 32: Spring - Ws

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.

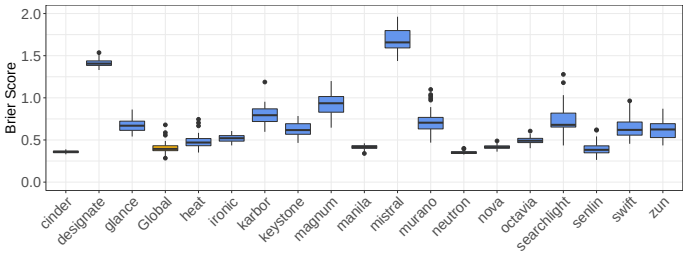


Fig. 33: OpenStack - Blazar

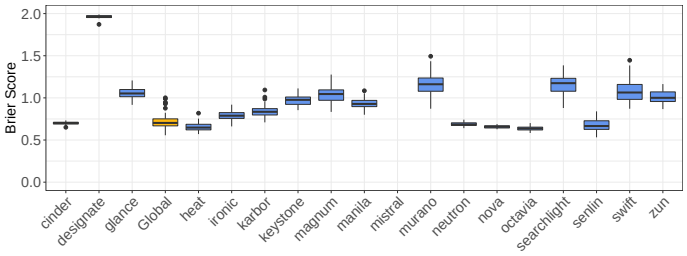


Fig. 34: OpenStack - Cyborg

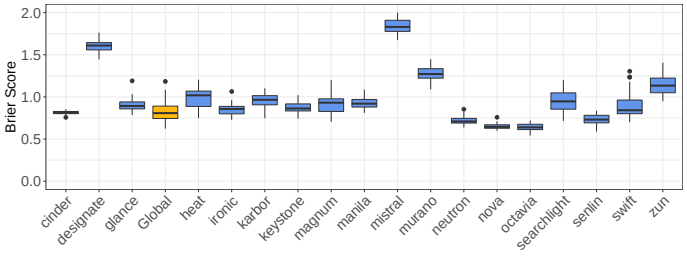


Fig. 35: OpenStack - ec2-api

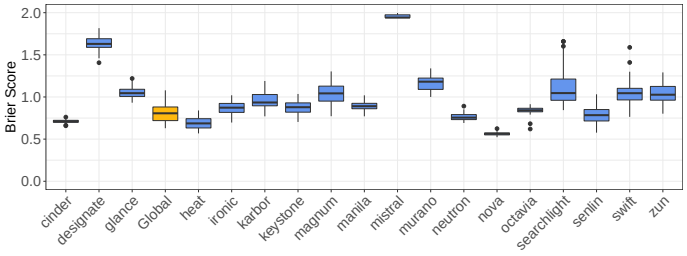


Fig. 36: OpenStack - Horizon

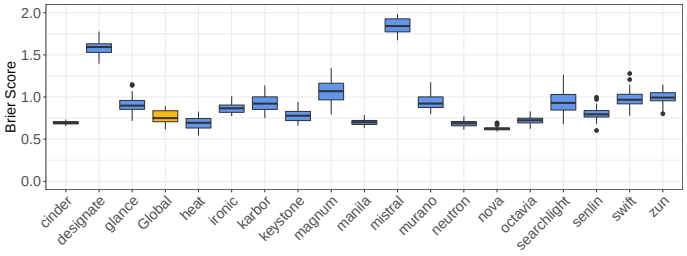


Fig. 37: OpenStack - Masakari

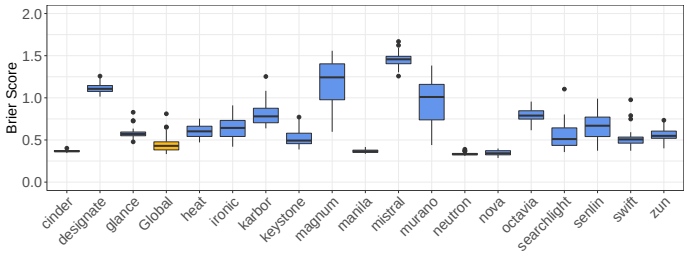


Fig. 38: OpenStack - Placement

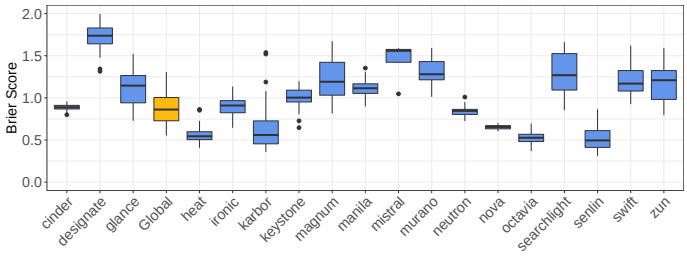


Fig. 39: OpenStack - Qinling

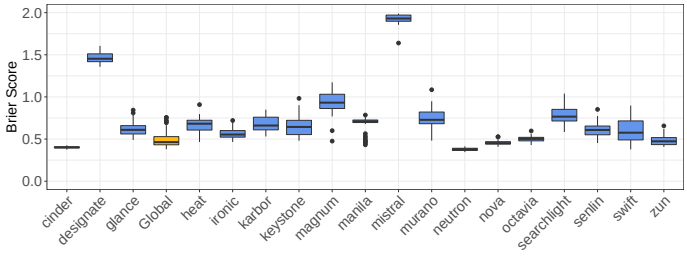


Fig. 40: OpenStack - Solum

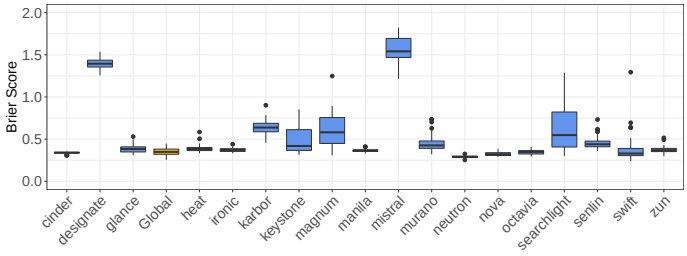


Fig. 41: OpenStack - Zaqar

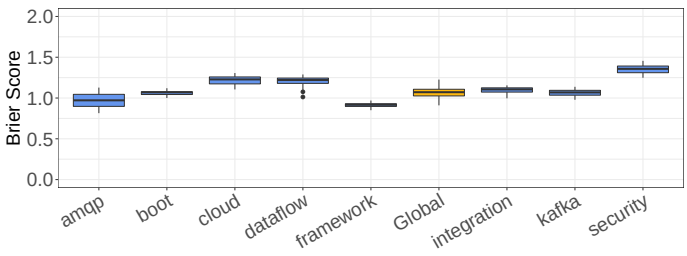


Fig. 42: Spring - Batch

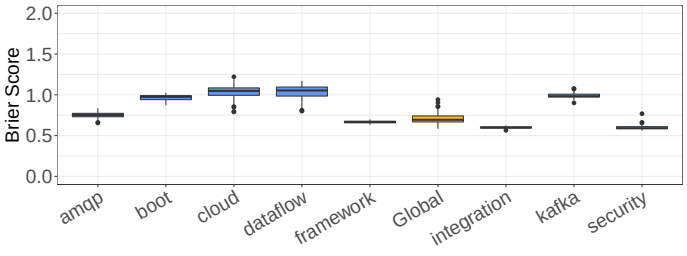


Fig. 43: Spring - Cloud-Commons

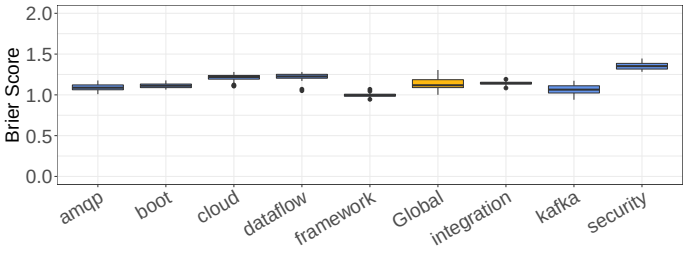


Fig. 44: Spring - Data-Commons

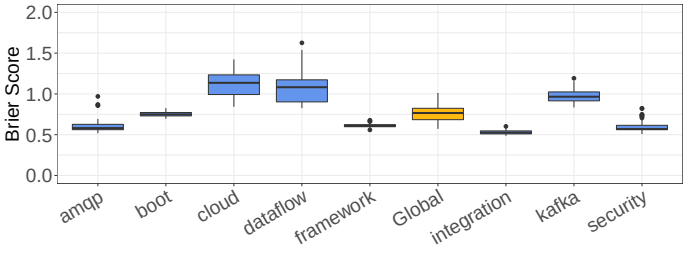


Fig. 45: Spring - Ldap

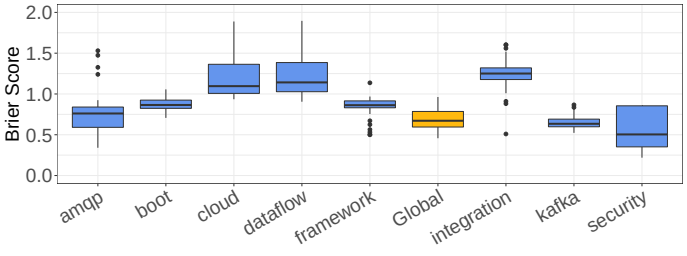


Fig. 46: Spring - Roo

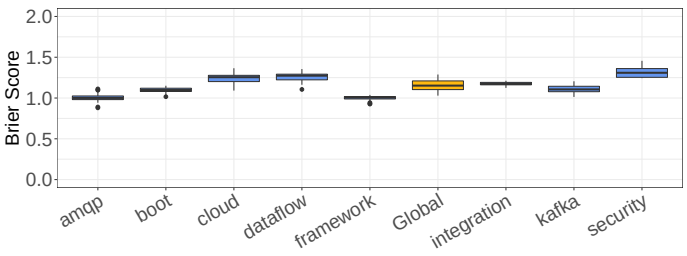


Fig. 47: Spring - Session

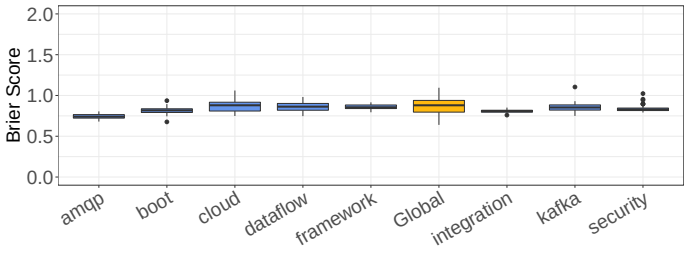


Fig. 48: Spring - Statemachine

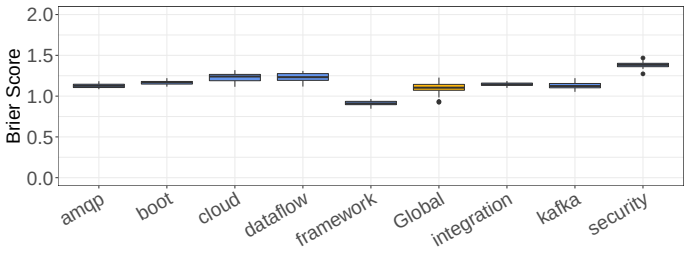


Fig. 49: Spring - Vault

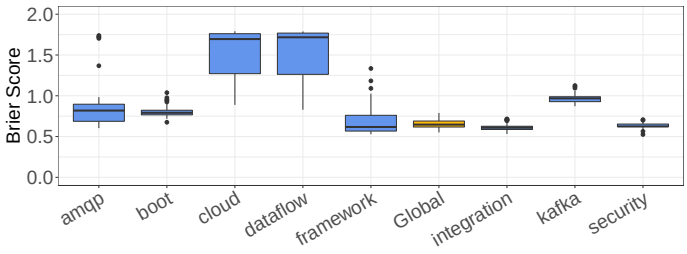


Fig. 50: Spring - Ws

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

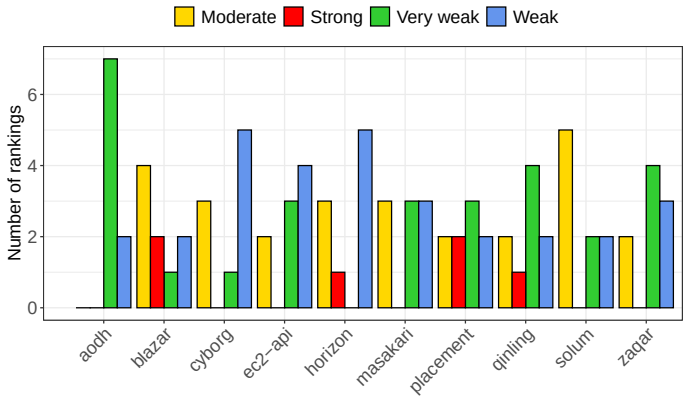


Fig. 51: Level of Correlation between the rankings of best performing peer-local models across the data lacking components of OpenStack.

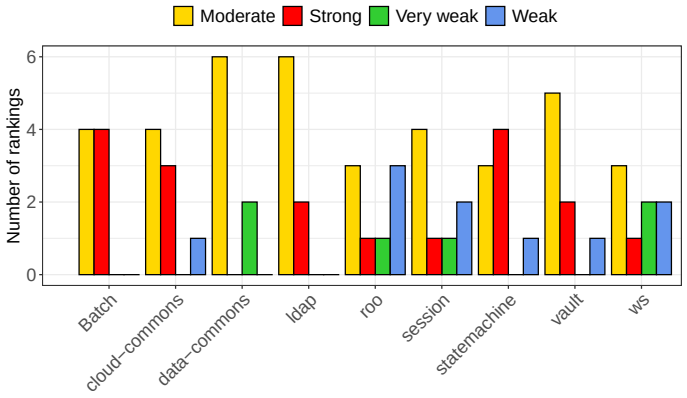


Fig. 52: Level of Correlation between the rankings of best performing peer-local models across the data lacking components of Spring.

Appendix D

Feature importance across components

Tables 1, 2, 5, 3 and 4 show the ranking of the most important features for deciding the log 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 stmnt	-	13	-	-	7	-
Avg static length	9	16	10	7	6	-
Avg var number	7	10	6	6	8	6
Avg log 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 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 stmnt	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 log 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 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 log 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 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_stmnt	-	6	9	-	-
Avg static length	-	-	11	6	5
Avg var number	-	-	7	-	7
Avg log 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 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 stmnt	-	-	-	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 log 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 5: Important features rankings for OpenStack's local and global models