QuLog: Data-Driven Approach for Log Instruction Quality Assessment

Jasmin Bogatinovski¹, Sasho Nedelkoski¹, Alexander Acker¹, Jorge Cardoso², Odej Kao¹

¹Technical University Berlin, Berlin, Germany
²Huawei Munich Research Center, Munich, Germany
{jasmin.bogatinovski,nedelkoski,alexander.acker,jorge.cardoso,odej.kao}@tu-berlin.de

1 INTRODUCTION

Logging is an important programming practice in modern software development, as software logs – the end product of logging, are frequently adopted in diverse debugging and maintenance tasks. Logs record system events on arbitrary granularity and give insights into the inner-working state of the running system. The rich information they provide enables the developers and operators to analyze events and perform a wide range of tasks. Notable task examples relying on logs are comprehending system behaviour [28], troubleshooting [13], and tracking execution status [41].

Logs are textual event descriptors generated by log instructions in the source code. Figure 1a depicts an example of a log instruction and the log message (log for short) describing the executed system event. The log instructions are commonly composed of three parts 1) static text describing the event (e.g., VM {} created in {} seconds.), 2) variable text giving a dynamic information about the event (e.g., 8), and 3) log level (e.g., info, error, warning), denoting the subjective developer opinion for the severity degree of the recorded event. The importance of log instructions makes them widely present within the source code. For example, HBase – a popular Java software system, has more than 5k log instructions. Developers use diverse logging frameworks (e.g., Log4j [15]) and logging wrappers (e.g., SLF4J [39]), which provide common logging features unifying the log instructions writing.

Many companies are adopting logging frameworks and specify guidance to their developers on the quality requirements when writing log instructions [5]. The quality requirements define different properties of log instructions quality, such as 1) assignment of correct log levels, 2) writing static text with sufficient information (i.e., sufficient linguistic properties), 3) appropriate log instruction formats, and 4) correct log instruction placement within the source code [6]. The quality guidelines aim to align the expectations from the logs for both developers and operators that may work in different teams and use the logs for different activities. Since many maintenance tasks (e.g., tracing faulty activities, diagnosing failures, and performing root cause analysis) are frequently log-based [28], they directly depend on the quality of the logs, i.e., the quality properties of the log instructions. Therefore, evaluating the quality of the log instructions emerges as a relevant task.

A central problem in this context is to write log instructions with sufficient quality. Recent studies on industrial [5] and open-source software systems [6] suggest that developers make recurrent log-related commits during development. It means that writing quality log instructions for the first time, even with given quality guidelines, is not trivial. Additionally, the guidelines can be incomplete and do
not cover every possible case. For example, in the Jira issue LOG4J2-316 \(^1\), a developer reported that the logging guidelines misguided him in proper usage of log levels.

While the logging frameworks unify logging, they do not implement mechanisms to track the quality. Thereby, the decisions about the log instructions are purely human-centric, which can result in poor logging practices (e.g., wrong log level assignment or insufficient linguistic structure) \[^{29, 44}\]. For example, in the Jira issue HDFS-4048 \(^2\) (depicted in Figure 1b), the wrong log level of the instruction `LOG.info("Cannot access storage directory " + rootPath);` resulted in a long time for localization of the failure. The developer used the log levels "error" and "warning" for log-based failure localization, but the event initially was logged on log level "info", not "error". Similarly, in the Jira issue ZOOKEEPER-2126 \(^3\) (depicted in Figure 1c), the log instruction has insufficient information about which EventThread is terminated. As reported by the developers, it becomes confusing when a new EventThread is created before terminating the previous one. The lack of a session identifier was pointed to as the main concern. The problem is resolved by adding additional words in the static text to give minimal information about the event which can be understood/comprehended by the developers. Notably, in linguistic terms, this means enriching the linguistic structure of the static text. The aforementioned issues are not isolated events. Previous works on logging practices \[^{6, 29}\] suggest that it is surprisingly common for the log levels to be over/under-estimated or the logs to have missing or excessive information. These problems are particularly challenging in complex software, with many different components developed by multiple developers located in diverse geographical regions (e.g., systems like OpenStack). It requires non-trivial knowledge and experience to construct an understandable description of the event, estimate the log levels of the instruction or conduct quality logging practices in general. Although the human-centric approach in log quality assessment is the golden standard, the aforementioned challenges imply the need for an automatic approach.

In this paper, we address the log quality assessment problem. Our goal is to develop an approach to automatically assess the quality of log instructions from an arbitrary software system. Such an approach is challenged by the heterogeneity of the software systems, the unique writing styles of developers, and different programming languages. They limit the set of testable quality properties. For example, the different syntax of the nearby code from two programming languages (e.g., Java and Python) questions the applicability of log instruction placement methods on arbitrary system \[^{5}\]. To find the empirically testable properties, we performed a preliminary manual study on the properties of the log instructions from nine open-source systems. We identified two such quality properties – 1) log level assignment assessing the correctness of the log level, and 2) sufficient linguistic structure assessing the minimal linguistic richness of the static text necessary for verbose event description. Through the preliminary study, we find that the static text of the log instruction is sufficient in assessing the two properties. Therefore, the log quality assessment is done on the static text of the instructions, independent of the other properties of the source code (e.g., code structure). This makes the log quality assessment system-agnostic. The observed dependencies between the static text on one side, and the log levels and linguistically sufficient labels on the other side allow the application of data-driven methods. Ultimately enabling the automation of the log instruction quality assessment.

Based on our observations, we propose QuLog as an approach to automatically assess log instructions quality. QuLog trains two deep learning models from the log instructions of many software systems and appropriate learning objectives to learn quality properties for the log levels and sufficient linguistic structure of static texts. To capture diverse developers logging styles, we trained the models on a carefully constructed log instruction collection with expected good quality practices similar as in related work \[^{6}\]. By adopting an approach from explainable AI, we further implemented a prediction explainer to show why the models make certain predictions, which serve as additional feedback for developers. Our experimental results show that QuLog achieves high performance in assessing the two properties outperforming the baselines. The prediction explainer has a low error for correct predictions suggestions. Thereby, QuLog helps to assess the log instructions quality while giving useful suggestions for their improvement.

In a nutshell, our contributions summarize as:

1. We performed a manual analysis on the quality log properties of the log instructions on nine software systems and identified 1) log level and 2) sufficient linguistic structure assessments as two empirically testable properties.
2. We implemented a novel approach for automatic log quality assessment named QuLog, which uses deep learning and explainable AI methods to evaluate the two properties.
3. We experimentally demonstrate the usefulness of our approach in automatic log quality assessment, which achieves high accuracy for log level assignment (0.88) and a high F1 on sufficient linguistic structure (0.99) assessments.
4. We open-source the code, datasets and additional experimental results in the code repository \[^{2}\].
2 LOG INSTRUCTION QUALITY ASSESSMENT

2.1 Log Instruction Quality Properties
To assess the quality of the log instructions, we examined literature studies on logging practices. We identified two views: explicit (or developers), and implicit (or operators). The explicit view is related to (a.1) correct log level assignment, (a.2) comprehensive content of the static text and parameters, and (a.3) correct log instruction placement [21]. The implicit view is related to the operators’ expectations for the quality of the logs. By observing the properties of the logs, we can implicitly reason about the quality properties of the log instruction. For the implicit view, there are four properties [44], given as follows: (b.1) trustworthiness - refers to the valid meta-information of the log (e.g., correct log level), (b.2) the semantics/linguistic of the log - relates the word choice in verbose expression of the event, (b.3) completeness - reflects the co-occurring of logs to describe an event, and (b.4) safeness - refers to the log content being compliant with user safety requirements.

Since our goal is to provide an automatic log instruction quality assessment, we first examine the feasibility of automatically evaluating the properties. We observed that some of the properties (i.e., correct log level assignment and linguistic evaluation) depend and can be assessed just from the content of log instructions. Therefore, they can be evaluated irrelevant to the structure of the source code and the remaining logging practices. To verify our observation, we made a preliminary study of nine open-source systems, with presumably good logging practices (similarly as in related works [6, 34, 41]). Table 1 enlists the properties of the used systems. We select these systems because they serve many industries, are being developed by many experienced developers, and consequently, the logs have fulfilled their purpose in debugging and maintenance.

Table 1: Overview of the studied systems

| Software System | Version | LOC | NOL |
|-----------------|---------|-----|-----|
| Cassandra       | 3.11.4  | 432K | 1.3K |
| Elasticsearch   | 7.4.0   | 1.5M | 2.5K |
| Flink           | 1.8.2   | 177K | 2.5K |
| HBase           | 2.2.1   | 1.26M| 5.5K |
| JMeter          | 5.3.0   | 143K | 1.9K |
| Kafka           | 2.3.0   | 267K | 1.5K |
| Karaf           | 4.2.9   | 133K | 0.7K |
| Wicket          | 8.6.1   | 216K | 0.4K |
| Zookeeper       | 3.5.6   | 97K  | 1.2K |

Note: LOC and NOL stand for the number of code and log lines accordingly.

2.2 Empirical Study
2.2.1 Log Level Assignment. We assume that the static text of the log instruction has relevant features for log level assignment. Intuitively, when describing an event with the “error” log level, the static text commonly contains words like “error”, “failure”, “exit”, and similar. Whenever these words occur within the static text, it is more likely that the level is “error” than “info”. To verify our assumption, we considered an approach from information theory that defines the amount of uncertainty in the information message [11]. We analyze the relation of word groups (n-grams, n = {1, 2, 3, 4, 5}) from the static text in relation to the log level. For all the n-gram groups, we try to identify the log level using n-grams from the given static text of the log instruction. At first, given an n-gram, there is high uncertainty for the assigned log level. As we receive more information about the n-gram, we update our belief for its commonly assigned log level, reducing the entropy (uncertainty) associated with the n-gram. To measure the uncertainty, we used Normalized Shannon’s entropy [20]. We calculated the log level entropy for each n-gram from all the log instructions of the nine software systems and reported the key statistics for the distribution. For example, the n-gram ”Machine failure” may appear 100 times, from which 99 times is associated “error”, and once with “info”. By calculating the entropy we obtain 0.05, which reflects low uncertainty about the n-gram word association with a level other than ‘error’.

Table 2: Log level assignment empirical study results.

|                        | Min | 1st Qu. | Median | 3rd Qu. | Max |
|------------------------|-----|---------|--------|---------|-----|
| Average Entropy        | 0.00| 0.00    | 0.00   | 0.56    | 0.91|

Table 2 summarizes the n-grams entropy distribution. It is seen that the majority of the static text of the log instructions have low entropy. Specifically, more than 50% (the median) of the static texts have zero entropy – the n-grams appear on a unique level. Therefore, the static text has relevant features useful to discriminate the log levels, verifying our assumption.

2.2.2 Linguistic Quality Assessment. A quality log instruction should describe the event concisely and verbosely [6]. From a general language perspective, complete and concise short texts (following the maxims of text quantity and quality) have a minimal linguistic structure (e.g., usage of nouns, verbs, prepositions, adjectives) [14]. Under the term log linguistic structure, we understand the representation of the static text by general linguistic properties such as linguistic concepts (e.g., verbs, nouns, adjectives etc.). For example, in the Jira issue ZOOKEEPER-2126 (depicted in Figure 1c), the static text “EventThread shut down” linguistically is composed of “noun verb particle”. Owning to the shared properties of the general English language and language used in log instructions [22], we assume that an informative event description also has a minimal linguistic structure. The following example explains our intuition for the assumption. In the aforementioned Jira issue, developers reported that the event information is insufficient. This issue is resolved by static text augmentation with additional linguistic properties, i.e., ‘EventThread shut down for session: {}”, linguistically composed of “noun verb particle preposition noun: -LRB- -RRB-” (where “-LRB- -RRB-” denote brackets). Linguistically speaking, the static text with insufficient linguistic structure is transformed into static text with sufficient structure, improving the event comprehension.

To validate our assumption, we performed the following experiment. For the static text of each log instruction, we first extract their linguistic structure. To do so, we use part-of-speech (POS) tagging – a learning task from NLP research. It allows extraction of the linguistic structure of the static text by linking the words to an ontology of the English language (OntoNote5). We choose spacy implementation of POS tagging because its models have high
performance on the POS tagging task (>97% accuracy score) [23]. Second, we group the extracted linguistic structures such that the static text with the same linguistic group are placed together. Afterwards, the linguistic groups of the raw static text are evaluated by two experienced developers answering the research question: ’Does the static text from the examined linguistic group contains minimal information required to comprehend the described event?’ This question evaluates our assumption that the quality and self-sustained static text has a minimal linguistic structure aligned with expert intuition for a comprehensible event description.

Table 3: Linguistic quality assessment preliminary study

| Linguistic Group | Total Log Instructions | Static Text (Example) |
|------------------|------------------------|-----------------------|
| VERB NOUN        | 106                    | serialized regioninfo |
| VERB             | 67                     | deleted               |
| VERB PUNCT       | 49                     | interrupted *         |
| NOUN             | 47                     | return                |
| NOUN NOUN        | 41                     | updating header       |

Table 3 gives the top-5 frequent linguistic groups alongside representative examples. In total, we found 5.9K linguistic groups from the studied systems. Then, we randomly sample 361 groups based on a 95% confidence interval and a 5% confidence level [46]. The sampling is stratified over the nine systems. The two human experts identified 24 linguistic groups with insufficient linguistic structure. The agreement between the two experts assessed by Cohen’s Kappa score is sustainable (0.72) [42]. The high score values show mutual agreement between the experienced developers concerning the relation between comprehensible event information within the static text and its linguistic structure. Therefore, the linguistic structure of the static text is useful in representing the minimal informative description of the log instruction.

2.2.3 Other Quality Properties. The remaining quality properties (i.e., relevant variable selection, log instruction placement, safeness, and completeness) depend on the different programming languages, design patterns, and other source code structures. These properties are challenging for assessment because of the heterogeneity of software systems and the ways programming languages organize the source code. For example, safeness requires reasoning across a complex chain of method invocations (e.g., in the issue CVE-2021-44228 the bug allows execution of any Java method through the log instruction from an LDAP server hurting the safeness). Identifying safeness requires a deep understanding of potential method invocation chains, which does not even require the method’s presence within the source code, i.e., requiring human involvement. The latter is against our effort in automatic log quality assessment. Due to the identified relationships between the static text and log level and sufficient linguistic structure on one side, and the dependence of the other quality properties on the remaining parts of the source code on the other side, we consider the log quality assessment in the narrower sense, expressed of the former two quality properties.

3 QULOG: AUTOMATIC APPROACH FOR LOG INSTRUCTION QUALITY ASSESSMENT

Inspired by our findings in the preliminary study, we propose an approach for automatic system-agnostic log instruction quality assessment. We formulate the problem in the scope of 1) evaluating the correct log level assignment and 2) evaluating the sufficient linguistic structure of the log instructions. Given the static text of the log instruction, we apply deep learning methods to learn static text properties concerning the correct log level and sufficient linguistic structure. By training the models on systems with quality logging properties, they learn information for the log level and sufficient linguistic structure qualities. Comparing the predicted log levels and the log levels assigned by developers allows a statement on the log level quality: the less deviation, the better the quality. Similarly, the sufficient linguistic structure incorporates properties of comprehensible log instructions, and its predictions directly are used to assess linguistic quality.

Figure 2 illustrates the overview of the approach, named QuLog. Logically, it is composed of (1) log instruction preprocessing, (2) deep learning framework and (3) prediction explainer. The role of the log instruction preprocessing is to extract the log instructions from the input source files and process them into a suitable learning format for the deep learning framework. The deep learning framework is composed of two neural networks (one for each of the two quality properties). The neural networks are trained separately on the two tasks. After training, the networks learn discriminative features for the log instructions with different log levels and a sufficient linguistic structure. The prediction explainer explains a certain prediction. Specifically, given the static text and predicted log level, it shows how different words contribute to the model prediction.

QuLog has two operational phases: offline and online. During the offline phase, the parameters of the neural networks and explanation part are learned on representative data from other software systems. This training procedure allows learning diverse developers writing styles, important for generalization. The learned models are stored. In the online phase, the source files of the target software system are given as QuLog’s input. QuLog extracts the log instructions, the static texts and log levels, proceeding them towards the loaded models. As output QuLog provides the predictions for the log levels, sufficient linguistic structure, and prediction explanations as word importance scores. Therefore, QuLog serves as a standalone recommendation approach to aid developers in improving the quality of the log instructions. The developers may reconsider improving the log instructions given QuLog’s suggestions or reject them. In the following, we delineate the details of QuLog.

3.1 Log Instruction Preprocessing

The purpose of the log instruction preprocessing is twofold. First, it extracts the log instructions from the source files. Second, it parses the log instructions to separate the static text and the log level. In addition, the static text is processed by the linguistic features extractor, to obtain its linguistic representation. These operations are performed by two modules, namely (1) log instruction extractor and (2) log instruction preparation, described in the following.

3.1.1 Log Instruction Extractor. The extractor module extracts the log instructions from the source code of the software system. To
that end, it iterates over all of the source files in the target software’s source code and applies regular expressions to find all log instructions. Considering the diversity of the programming languages, developers writing styles, and the lack of adoption of logging practices challenges the extraction process. The output of the extraction module is a set of log instructions of the input software system. Although our goal is to help developers in writing correct log levels, we restrain ourselves on three levels (“info”, “warning”, and “error”). The log levels “trace” and “debug” provide detailed information for the inner workings, most commonly used by developers. By studying the n-grams frequency for individual log levels, we found that there is a large overlap between the used vocabulary in “info” and “trace”/“debug” levels. This can significantly impair the performance of the data-driven methods when automatically assessing the quality of all log levels simultaneously. In addition, related work reports this scenario practically useful when different stakeholders examine logs. For example, operators care more for the high severity levels (i.e., “info”, “warning”, “error”) [34].

3.1.2 Preparation. The goal of the preparation module is to prepare the data in a suitable learning format. As input, it receives the set of log instructions from the extractor. The preparation module first iterates over the log instructions and separates the static text of the log instructions from the log level. The diverse programming languages use different names for the log levels. For example, Log[4] (a Java’s logging library) uses the tag “warn” for warning logs, while the default Python logging library uses the tag “warning”. To that end, the preparation submodule unifies the levels for all log instructions. To the static text of the log instruction, we apply Spacy [23] for preprocessing. We split the words using space and camel cases. We preprocess the static text by following preprocessing techniques, including: remove all ASCII special characters, removing stopwords from the Spacy English dictionary and applying lower case transformation of the words [8]. Once processed, we give the static text as input to a pre-trained POS tagging model from Spacy. We extract the pos tag of each word from the static text to create its linguistic structure. Finally, the output of this module is a set of tuples, where each tuple is composed of the static text of the instruction, the linguistic structure of the static text and the level.

3.2 Deep Learning Framework

3.2.1 Overall Architecture. QuLog has two independent neural networks to assess the two quality properties. They share the same architectural design and are composed of embedding layer, encoder network from Transformer architecture [45] and output layer. For clearer description, we explain the working principle for the log level assignment. Alongside it, in parenthesis, we give the mapping for sufficient linguistic structure assessment. Given the preprocessed static text (linguistic structure) at the input, the embedding layer learns numerical vector representation of the individual words (linguistic categories), we referred to as tokens, following a distributed learning representation paradigm [43]. The vector embeddings of the tokens are numerical features in a suitable learning format for the network. We then use the encoder of the Transformer architecture to learn relationships between the vector embeddings of the input tokens from the embedding layer and the log levels (sufficient/insufficient linguistic structure). The output from the encoder layer is a vector embedding of the static text (linguistic structure). After that, the output layer predicts the level (sufficient linguistic structure) from the encoder layer’s output.

3.2.2 Embedding Layer. The embedding layer receives the preprocessed instructions as input. We first transform the static text (linguistic structure) from a sequence of words to a sequence of tokens/indices. Figure 2 gives an example of this transformation. It enables the transition from textual into a numerical format as a prerequisite to applying neural networks. We further prepend the tokenized static text (linguistic structure) with a special token we refer to as Log Message Token ([LMT]). Note that this is an important detail we discuss further when describing the neural networks. Since the static texts can be of different lengths, while the neural network requires fixed-size input, we specify a hyperparameter max_len to unify the lengths. The shorter static texts (linguistic structures) are appended with a special pad token ([PD]), while the longer ones are truncated at max_len value. The embedding layer maps the input tokens into a numerical vector representation, such that each unique token is assigned a specific vector. In QuLog, these embeddings are learned during training, and the vector embeddings
are adjusted to preserve contextual properties (e.g., frequently co-occurring words for a certain log level). These vectors can also be obtained from general-purpose language models (e.g., BERT [12]).

3.2.3 Transformer Neural Network. We model the dependencies between the tokens and the two quality properties with nonlinear parametric functions represented as neural networks. As a suitable architecture, we identified the encoder of the Transformer [45] architecture. It provides state-of-the-art results in many NLP tasks (e.g., sentiment analysis, translation) [4]. By pointing to the similarities between the static text of logs and natural language [22], we further justify our design of choice. The encoder implements a multi-head self-attention mechanism that exploits higher-order relations between tokens within the static text (beyond n-gram counting). This property captures discriminative features between the words (linguistic concepts) and the different contexts they appear in, relating them to the appropriate log levels (sufficient linguistic structures). During training, the token embedding vectors and the network parameters are updated via backpropagation [35].

At the output of the encoder, we provide the vector embedding of the [LMT] token. Due to the architectural design, the vector of the [LMT] token attends over all the other token vector embeddings during training. This allows summarizing the most relevant information from the input concerning the log level (sufficient linguistic structure). Therefore, it embeds diverse contexts preserving the properties of the static text (linguistic structure). The number of heads in the multi-head self-attention, the number of layers and model size are three hyperparameters of the network.

3.2.4 Output Layer. The output layer is a three-dimensional linear layer for predicting the log level (two-dimensional for sufficient linguistic structure). It accepts the [LMT] vector embedding and applies a linear transformation. Each of the output dimensions corresponds to one of the log levels (i.e., "info", "warning", "error") or one of the two linguistic structures qualities (i.e., sufficient and insufficient). We apply a softmax function at the output neurons to produce score estimates. Each neuron gives a score estimate for the corresponding class (i.e., a number between 0-1 indicating class relevance for the given input). The scores give insights into the model’s confidence for the log level (sufficient linguistic structure) prediction. As a class prediction, we considered the class related to the neuron with the highest score.

3.3 Prediction Explainer

The prediction explainer aims to explain why a trained model makes certain predictions. It augments QuLog’s output by pointing out the specific tokens most likely contributing to the prediction. They can serve as suggestions to developers for where the static text potentially can be altered. The relevant details of the explanation module are given in the following.

3.3.1 SHAP. Prediction explainer leverages SHAP [37] (Shapley additive explanations) – an approach from explainable artificial intelligence. In general, SHAP calculates feature importance scores (how relevant is a feature for the prediction) by defining the problem as a coalitional game between the features. The goal is to find the so-called Shapley values for each feature defined as the fairest distribution of the "payout" (as importance score) for the prediction. SHAP, each token is assigned with a vector of Shapley values (with number of layers and number of tokens in unity), we express the token importance as a single number. We refer to this as a token importance score. To calculate the token importance score, we aggregate the individual Shapley values for each token. The token importance score has two parts: 1) intensity and 2) sign. The intensity shows the influence strength of the token for the prediction. The sign shows the token direction influence for the prediction (favouring or not-favouring a decision). After experimentation with different aggregation functions, we find that the second norm of the Shapley value vector and the sign of the Shapley value with the highest absolute score, are suitable for intensity and sign aggregation functions. Formally, they are given in Eq. 1 and Eq. 2.

\[
\begin{align*}
  r(t_{ji}) &= |\{S_i(t_{ji})\}|_2 \\
  e(t_{ji}) &= \text{sign}(\max_k |S_{ijk}(t_{ji})|)
\end{align*}
\]

where \( r(t_{ji}) \) is the token \((t_{ji}) \) importance score intensity, \( e(t_{ji}) \) is the token importance sign for the log instruction \( l_j \). \( S_{ijk} \) denotes the Shapley value for the \( 'k' \)-th position of the \( 'i'-th \) token of the log instruction static text (i.e., in the example the token "refused" has a Shapley value \( S_{224} = -5.22 \)).

In the example, the difference between the two instructions distinguishing the levels "info" from "error" is in the second token. We first calculate the individual Shapley values and then calculate the intensity and sign of the token importance scores. As seen by the score values, the following inequalities hold \( r_{12} > r_{11}, r_{22} > r_{21} \). The second token in both of the instructions ("established", "refused") has greater intensity compared to the first ("connection"), thereby, contributing more to the model prediction. Additionally, the token signs show that the token "established" is favourable for the class "info" \( e_{12} = + \), while the token "refused" is not-favourable

| Log Instruction | Scores: | Token Importance Scores: |
|-----------------|---------|--------------------------|
| log.info("connection established") | connection: \( (1.21, 0.41, -0.12, 0.14) \) | connection \( r_{12} = 1.36, e_{12} = + \) established \( r_{22} = 4.36, e_{22} = + \) |
| log.error("connection refused") | connection: \( (0.21, -0.20, 0.42, -0.56) \) | connection \( r_{12} = 0.76, e_{12} = + \) refused \( r_{22} = 6.65, e_{22} = - \) |

Figure 3: Prediction Explainer working procedure example

The larger the value, the more important is the feature towards the prediction. The signs of the Shapley values show the feature’s prediction favorableness (or non-favorableness for negative sign).

3.3.2 Implementation. We used the original SHAP implementation with the default values for its parameters [36]. One required parameter of SHAP is a differentiable learning model (a model with gradients calculated for each network layer). To apply SHAP, we used the trained encoder network as input. While the explanation procedure is applicable for both quality properties, we described just the log level prediction explainer because of the intuitive meaning of the importance scores concerning the predictions.

3.3.3 Token Importance Scores. The Shapley values are calculated for each neuron of the input vector embeddings. Figure 3 illustrates a running example. There are two log instructions \( l_1 : \) "Connection established!" with the level "info", and \( l_2 : \) "Connection refused!" with the level "error". After running the two instructions through SHAP, each token is assigned with a vector of Shapley values (with size \( d = 4 \)). However, to reason about the influence of the token in unity, we express the token importance as a single number. We refer to this as a token importance score. To calculate the token importance score, we aggregate the individual Shapley values for each token. The token importance score has two parts: 1) intensity and 2) sign. The intensity shows the influence strength of the token for the prediction. The sign shows the token direction influence for the prediction (favouring or not-favouring a decision). After experimentation with different aggregation functions, we find that the second norm of the Shapley value vector and the sign of the Shapley value with the highest absolute score, are suitable for intensity and sign aggregation functions. Formally, they are given in Eq. 1 and Eq. 2.

\[
\begin{align*}
  r(t_{ji}) &= |\{S_i(t_{ji})\}|_2 \\
  e(t_{ji}) &= \text{sign}(\max_k |S_{ijk}(t_{ji})|)
\end{align*}
\]
for the class "info" ($e_{22} = -\infty$). Therefore, if there is a discrepancy between the developers' decision on the log level and QuLog's log level assignment, the developer examines the highlighted word, e.g., "refused", and considers changing either the level or the word. That way, QuLog automatically aids developers in improving the log quality. The output of the explanation module is an ordered list of tokens, ordered by their intensity (from highest to lowest).

4 EXPERIMENTAL EVALUATION

4.1 Experimental Setup

4.1.1 Code Repository Collection. Alongside the studied software systems, we collected log instructions from 100 other systems from GitHub. To collect this data, we crawled GitHub and searched for systems from the following topics: Java, Python, Angular, Ruby, and PHP, selecting 7039. Additionally, we collected the number of GitHub stars for each system. Similar to previous works [22], we assumed that the number of stars is a good indicator for the quality of logging and considered the top 100 in the experiments.

4.1.2 Evaluation Criteria. To evaluate QuLog, we used several evaluation criteria. First, we describe the criteria evaluating exact predictions. For the multi-class evaluation, we used accuracy. It gives the percentage of correct predictions out of all predictions. Due to the imbalances of the classes (e.g., the systems have a diverse number of "error", "warning", and "info" instructions), accuracy can be misleading [17]. Therefore, we considered precision, recall and $F_1$ scores. Precision evaluates the fraction of correct predictions out of the target class predictions. Recall evaluates the correct predictions out of the true class predictions. $F_1$ is the harmonic mean of the precision and recall evaluating the trade-off between correct class predictions and miss-classifications [17]. Additionally, we considered specificity in the binary classification. It is the measure of correct predictions of the negative class. The aforementioned criteria take values within the 0-1 range, and a higher value indicates better performance. The evaluation of the prediction explainer is done with the error@$k$ score. It measures the number of incorrect predictions when $k$-degrees of freedom are considered [24]. The smaller values indicated better performance. We additionally considered the AUC [19] score. AUC is the area under the ROC (receiver operating characteristic) curve that plots the true positive against the false positive rates. It is bounded in the 0-1 range, with a high value indicating better performance. The AUC value of 0.5 indicates a model performing not better than a random guess.

4.2 Log Level Assignment Evaluation

We first evaluate the performance of QuLog on the log level quality assessment. We split this evaluation into two parts. First, we compare QuLog against baselines. Second, we evaluate QuLog's performance on a few instances of the problem of log level assignment. The motivation for this evaluation type on one side is to examine the performance of QuLog against baselines, and on the other side, to identify problem instances where the inherited imperfect performance of data-driven approaches will not overwhelm developers with many incorrect predictions. The latter is relevant for QuLog's practical usability.

4.2.1 Comparison against baselines. In the evaluation of QuLog against baselines, we considered two QuLog models. They have the same architectural design but differ in the input data used to train them. The first model, we referred to as QuLog-8, is trained on data from eight software systems listed in Table 1. Since these systems are characterized by good logging practices, we assume that the majority of the log levels are correctly assigned, similar as in previous studies [34]. This accounts for the quality of the learning data. The second model for log level assignment we call QuLog* is trained on the collection of 100 GitHub systems. While QuLog* does not account very rigorously for the instruction quality (despite the pseudo indicator of having many stars), it enables testing for cross-software usefulness of the static text in log level assignment. As such, it aligns with the system-agnostic nature of QuLog. This is important in scenarios of log quality assessment where the software system is in the initial development stage, and there are not many log instructions for training a model. As an evaluation dataset, we considered the log instructions from one of the nine systems listed in Table 1, such that the instructions from the evaluation dataset are never seen during training the model, preventing data leakage.

Experiment Design. We compare QuLog against two baselines: DeepLV [34] and Support Vector Machines (SVM) [10]. DeepLV addresses the problem of log level assignment as an ordinal regression and trains LSTM – a deep-learning architecture, on features extracted from the log instruction. It is reported as the current best performing method for log level assignment. SVM is a popular multi-class classification method trained on the vector representation of the static text from general-purpose language models (BERT) [12] previously used for log level assignment [18]. The hyper-parameters of the baseline methods are set to the recommended values by the authors. As evaluation criteria, we used AUC and accuracy following literature standards [29, 34]. Regarding the considered hyper-parameters for QuLog's log level architecture, we considered the following ranges: model size (16, 32, 64, 128), layers number (2, 4, 6), and heads number (2, 4, 6, 8). The best results were obtained for: model size 16, heads number 2, and layers number of two. The maximal number of tokens to max_len = 50. As optimizer we used Adam [26] with learning rate $10^{-4}$ and hyperparameters $\beta_1 = 0.9, \beta_2 = 0.99$. The batch size was set to 2048.

Results and discussion. Table 4 gives the results of the evaluation of QuLog against baselines. We first compare the QuLog-8 model against the two baselines. Following the AUC criteria, it is seen that QuLog-8 achieves the best performance for all of the nine systems. Since AUC evaluates how good a model is in predicting the true level (e.g., "error") as correct (as "error") rather than predict incorrect level as the true one (e.g., "warning" instead of "error"), it means that QuLog-8 can discriminate the different log levels better. However, AUC evaluates scores instead of exact decisions for a particular log level. By deciding for a log level (i.e., maximal score estimate as a class prediction), we assign an actual log level for the particular log level. By deciding for a log level (i.e., maximal score estimate as a class prediction), we assign an actual log level for the
Table 4: Evaluation on log level quality assessment

| Systems       | QuLog-8 | DeepLV | SVM     | QuLog-9 | DeepLV | SVM     | QuLog   | DeepLV | SVM     |
|---------------|---------|--------|---------|---------|--------|---------|---------|--------|---------|
| Cassandra     | 0.94    | 0.78   | 0.80    | 0.96    | 0.63   | 0.61    | 0.94    | 0.67    | 0.67    |
| Elasticsearch | 0.95    | 0.71   | 0.71    | 0.94    | 0.59   | 0.51    | 0.95    | 0.60    | 0.60    |
| Flume         | 0.94    | 0.74   | 0.77    | 0.95    | 0.62   | 0.60    | 0.62    | 0.71    | 0.71    |
| HBase         | 0.91    | 0.77   | 0.80    | 0.92    | 0.59   | 0.61    | 0.64    | 0.63    | 0.63    |
| JMeter        | 0.92    | 0.73   | 0.74    | 0.95    | 0.59   | 0.55    | 0.53    | 0.68    | 0.68    |
| Kafka         | 0.93    | 0.68   | 0.70    | 0.98    | 0.58   | 0.51    | 0.51    | 0.69    | 0.69    |
| KoaF          | 0.93    | 0.73   | 0.79    | 0.94    | 0.63   | 0.57    | 0.58    | 0.64    | 0.64    |
| Wicket        | 0.94    | 0.74   | 0.75    | 0.95    | 0.75   | 0.56    | 0.59    | 0.78    | 0.78    |
| Zookeeper     | 0.92    | 0.68   | 0.74    | 0.94    | 0.59   | 0.50    | 0.57    | 0.62    | 0.62    |
| **Average**   | 0.93    | 0.74   | 0.75    | 0.95    | 0.62   | 0.56    | 0.58    | 0.67    | 0.67    |

our approach is useful in assessing the correctness of log levels for the considered systems outperforming the baselines.

Next, we compare QuLog* against QuLog*. The results on the two evaluation criteria show that QuLog* outperforms QuLog-8 by 1-9% on accuracy and 1-5% on AUC for different systems. These results indicate the existence of shared system-agnostic properties of the static text and the log levels, independent of the software systems examined in the preliminary study. The instructions originate from different programming languages and publicly accessible software systems from GitHub, representing diverse developers writing styles. Therefore, by their leveraging, QuLog* learns a wide range of characteristics of the static text concerning the log levels (e.g., large vocabulary used in similar event descriptions). The good performance across different systems and the system-agnostic training of QuLog* suggest that QuLog is suitable for an automatic assessment of the quality of the log instructions, represented by their correct log level assignment.

Table 5: Log level misclassification contingency table (the averaging is done over nine software systems given in Table 1)

| True/Predicted | Info    | Warning | Error |
|----------------|---------|---------|-------|
| Info           | -       | 21.1%   | 16.1% |
| Warning        | 10.7%   | -       | 40.3% |
| Error          | 4.3%    | 19.3%   | -     |

4.2.2 Log Level Problem Instances. The previous experiment shows that QuLog performs better than the baselines on log level assignments. However, the results between 0.60-0.78 on accuracy across different systems, although good, indicate that there are incorrect assignments. The misclassifications can impair the practical usability of QuLog. Therefore, we study the misclassification types. Based on the observations, we identified instances of the log level assignment problem having improved results facilitating the practical applicability of QuLog. To study QuLog’s misclassification types, we calculated the misclassification contingency table. It shows the percentage of misclassification prediction rates for the three classes. Table 5 gives the contingency table. It is seen that some class pairs have a low misclassification rate (e.g., true “error” predicted as “info” is 4.3%), however for others, it is significantly high (e.g., true “warning” predicted as “error” is 40.3%). To understand the potential reasons, we examined the n-gram frequency shared between the different log levels similar to the preliminary study (Section 2.2.1). We find that n-grams shared between the log level pairs “error-warning” is 14.2%, and it is higher compared to “error-info” (4.9%) and “warning-info” (9.7%). Relating it to the contingency matrix, we see that the class pairs with higher n-grams overlap have higher misclassification rates. We use this observation to construct three simplified instances of the log level quality assessment. Instead of predicting the three classes, we considered the prediction of two classes, namely “info-warning” (IW), “info-error” (IE) and “error-warning” (EW). The examination of individual class pairs has practical relevance because different stakeholders have different expectations from logs. For example, the operators usually examine the log levels “error” and “warning”. Therefore, misclassifying an error event as “info” (e.g., Jira issue HDFS-4048) can hide important events from operators, increasing the maintenance costs.

**Experiment design.** We considered QuLog* log level assignment approach because it is system-agnostic. To train QuLog* on the three two-class problems, we modified the output layer to have two classes instead of three. The experiment is designed as follows. We start with the 100 software systems collected during the data collection procedure. We randomly sampled 60% of the repositories for training, 20% for validation and 20% for evaluation. To reduce the variance of the results due to the random repositories selection, we repeated the sampling procedure 30 times and reported the average results alongside the standard deviations. To assess the correctness of the decisions, we used F1, precision and recall, instead of accuracy because they are exposing the imbalances of the class distributions better than accuracy. We used the same baselines as in the previous experiment trained with the same data as QuLog*.

Table 6: Performance scores on the task of log level assignment. The best results per scenario are bolded.

| Scores | Scenario | QuLog   | DeepLV | SVM     | BERT-SVM |
|--------|----------|---------|--------|---------|----------|
| F1     | IE       | 0.88±0.03 | 0.82±0.02 | 0.87±0.04 |          |
|        | IW       | 0.75±0.03 | 0.67±0.03 | 0.73±0.04 |          |
|        | WE       | 0.68±0.06 | 0.61±0.08 | 0.64±0.05 |          |
|        |          | 0.61±0.04 | 0.56±0.06 | 0.54±0.05 |          |
|        |          | 0.88±0.02 | 0.79±0.04 | 0.92±0.04 |          |
|        |          | 0.72±0.03 | 0.66±0.03 | 0.74±0.05 |          |
|        |          | 0.75±0.04 | 0.72±0.06 | 0.58±0.05 |          |
|        |          | 0.69±0.09 | 0.59±0.08 | 0.51±0.07 |          |
|        |          | 0.89±0.05 | 0.86±0.06 | 0.84±0.06 |          |
|        |          | 0.75±0.03 | 0.68±0.03 | 0.73±0.04 |          |
|        |          | 0.62±0.08 | 0.54±0.10 | 0.72±0.09 |          |
|        |          | 0.56±0.07 | 0.54±0.08 | 0.59±0.08 |          |

**Results and discussion.** Table 6 enlists the performance scores for the four problem instances of log level quality assessment. Comparing the absolute values for the scores across the four scenarios reveals that the IE problem achieves the highest values on F1 score (average of 0.88), i.e., trades-off the precision (0.88) and recall (0.89) quite well. Therefore, this model is very reliable for correctly assessing the “info” and “error” log instructions. The good performance is attributed to the observed differences in the vocabulary between the “error” and “info” log instructions (i.e., a 4.9% n-gram overlap). Therefore, this model won’t overwhelm developers with many incorrect predictions. On the IW and EW problem instances, although QuLog does not perform as good, still outperform the baselines when different software systems are considered.
4.3 Linguistic Quality Assessment Evaluation

4.3.1 Experimental Design. To evaluate the sufficiency in the linguistic structure of the static text, we used the data from the preliminary study as given in Section 2.2.2. We trained QuLog on the linguistic representations from the eight systems and evaluated the remaining one. Notably, we identify log instructions with insufficient linguistic structure in four of the tested systems: Cassandra, HBase, Kafka, and Zookeeper, and we report the results for them. As baselines, we considered two popular binary text classification methods, i.e., SVM and Random Forest (RF) [3], trained on the general-purpose representation of the linguistic categories (BiERT) [12]. We train QuLog’s linguistic quality assessment part with the same values of the hyperparameters as for the log quality assessment, setting with the batch size to 64. As evaluation criteria, we used F1 and specificity.

| System  | QuLog | SVM | RF | QuLog | SVM | RF |
|---------|-------|-----|----|-------|-----|----|
| Cassandra | 1.00  | 0.96 | 0.97 | 0.97  | 0.94 | 0.92 |
| HBase    | 0.99  | 0.98 | 0.92 | 1.00  | 1.00 | 0.74 |
| Kafka    | 0.99  | 0.99 | 0.98 | 1.00  | 0.98 | 0.94 |
| Zookeeper| 0.99  | 0.96 | 0.96 | 0.99  | 0.98 | 0.89 |

Average 0.98 0.97 0.96 0.99 0.98 0.89

4.3.2 Results and discussion. Table 7 enlists the evaluation results. It is seen that QuLog achieves a high average F1 score of 0.98 while outperforming the baselines by slight margins. The good performance of the three methods is attributed to the discriminative linguistic features between the two classes. For example, the HBase’s log instruction ‘failed parse’, from the class hadoop.hbase.zookeeper.ZKListener, has a linguistic structure “verb noun”. Notably, it does not contain information to which the parsing failure refers (i.e., lacks sufficient linguistic structure). As a comparison, in another log instruction “failed parse data for znode”, within the same class of HBase, the linguistic structure “verb noun” has four additional linguistic properties, i.e., it has the form “verb noun noun apposition noun parameter”. This additional linguistic structure has two advantages. From a learning perspective, the richer linguistic structure is useful for discriminating between the classes. From a comprehension perspective, it encodes verbose information on the type of failed parsing. The better performance of QuLog against the baselines can be attributed to its ability to extract a better representation of the linguistic structure. QuLog exploits log specific concepts, while the general-purpose language models are trained on datasets from general literature, which may average-out log specific properties.

The results on specificity are high for both QuLog and SVM while being a bit lower for RF. Since specificity evaluates methods’ performance in the correct prediction for the insufficient class (true negative class), the results show that QuLog can correctly identify the instructions with an insufficient linguistic structure. By combining these results with the high performance on F1 (as a trade-off between incorrect sufficient predictions), we conclude that QuLog detects the linguistically insufficient instructions without compromising the performance on the sufficient class. The high values for the two scores show that QuLog is useful in automatically assessing the sufficient linguistic structure in a system-agnostic manner. By pointing out the log instructions that may benefit from enriching the static text, QuLog improves the comprehensibility of the log instructions, ultimately improving the logging quality.

4.4 Prediction Explainer Evaluation

Experiment design. To evaluate the prediction explainer, we construct an artificial dataset as in the following: We start by randomly sampling 100 static texts of the instructions with a correct log level prediction of an already trained model (i.e., QuLog* for log level assignment). Each static text is examined by two developers and modified by randomly replacing a word with its antonym. This creates an event with an opposite meaning. For example, we start with the original static text “Connection established” with an original log level “info”. We change the token “established” into its antonym “refused”, creating a modified static text, i.e., “Connection refused”, and modified word “refused”. The modified static text describes an erroneous event, and we set its log level to “error”. Therefore, for each static text we obtain a tuple of five elements – 1) original static text, 2) modified static text, 3) modified word, 4) original level, and 5) modified level. From the initial 100, the two annotators initially agreed on 42 changes. In a subsequent discussion, the number was increased to 65 instructions used in the evaluation. The original and modified static texts are given to the prediction explainer that generates the ordered token list of importance scores. The modified token is used as ground truth. We check how many tokens should the developer examine before finding the modified token, and we measure it by the error@k performance score. We considered two log level models, the two-class IE, due to its high performance and the three-class log level assignment IWE. As a baseline, we consider suggesting a randomly chosen token as the most relevant.

Results and discussion. Figure 4 depicts the experimental results. It is seen that the prediction explanation module has a low error on correct word suggestions (the error@1 is 0.25) for the IE model. The prediction explanation model for the IWE model is a bit higher (the error@1 is 0.52), however, both explainers show better performance than the considered baseline. The observed discrepancy between the prediction explanations of the IE and IWE is due to the better performance of the IE model (average F1 score 0.88) as opposed to the IWE model (average F1 score of 0.73). It indicates that a
better performing model learns discriminative features better. By considering $k$ relevant tokens (i.e., a developer examines the $k$ highest-ranked tokens), the three explanation models reduce the error, with IWE and IWE having sharp decreases, achieving 0.05 and 0.23 on error@2 correspondingly. The low value of the one error shows that QuLog’s log level prediction explainer correctly explains the predictions. Therefore, the prediction explainer gives valuable suggestions on static text updates to improve quality.

### 4.5 Use Cases
We further applied QuLog on two internal systems. For log level assignment, we considered QuLog” IE model. Table 8 summarizes the results. For System 1, on the task of log level assignment, QuLog agrees in 52/63 cases with the original log levels and 56/63 on sufficient linguistic structure assessment. Developers examined the 11 disagreements for log level and the seven disagreements on sufficient linguistic structure. They decided to change 5/11 log levels and augmented 4/7 with additional linguistic structures. The remaining linguistic suggestions were considered as "unimportant". Similarly, for System 2, QuLog agreed in 120/138 cases on log level, with 8/18 log levels being changed. On the sufficient linguistic evaluation, QuLog identified five instructions with insufficient static text, three of which were accepted. These two examples showcase how QuLog’s automatically assess the log instructions, giving useful suggestions for improving the logging quality.

| Log Instructions | Log Level Recommendations | Linguistic Structure Recommendations |
|------------------|---------------------------|--------------------------------------|
| System 1         | 63                        | 52 (5/11)                            |
| System 2         | 138                       | 120 (8/18)                           |

### 4.6 Threats to Validity
The key threats to the validity of this study related are to the included datasets and the implementation details. We chose vetted systems for the preliminary study following related works [34]. We further complemented it with another public dataset collection to mitigate data selection artefacts. The datasets used for the sufficient linguistic structure and prediction explainer might be biased by the human annotation procedure. Therefore, we considered two annotators to construct each of them. A third-party evaluation may help to further mitigate the biases from the annotation.

### 5 RELATED WORK
Logging practices We discussed the studies on quality properties in Section 2.1. Alongside the quality properties, several literature studies are examining diverse logging practices. In one of the first studies, Yuan et al. [48] quantify the log pervasiveness and the benefit of software logging in C/C++ systems while proposing proactive logging strategies. They found that developers spent significant efforts modifying the log levels, static texts, and the parameters of log instructions but do not change their locations often. Similar observations are made for Java [6, 7, 25, 40, 52] and Android software systems [50]. These studies augment the aforementioned by introducing new research questions like studying log bug resolution time [6], log instruction update types [7] and evolution of logging configuration [52]. For example, Kabina et al. [25] identified that 20-45% of the log instructions change through system lifetime. The importance of logging practices is widely recognized in the industry, seen by several logging practice studies of industrial systems [1, 5, 38]. In a field study, Li et al. [28] demonstrates the different costs of logging from developers and research perspectives. The similarities in the conclusions regarding the various logging practices in different programming languages motivate our work on software systems cross-examination when evaluating log instruction quality.

**Automatic Logging Enhancement**. There are several methods that support the automatic enhancement of log instructions [27, 31, 33, 47, 53]. Based on the enhancement type, we distinguish two groups of methods, i.e., methods addressing the log instructions placement problem (where-to-log) [5, 9, 16, 27, 32], and methods addressing the choice of relevant logging information (what-to-log) [30, 33, 47, 49]. The latter can further be separated into three groups: 1) log message generation [22], 2) relevant variables placement [51], and 3) log level suggestion [18, 34]. Different from previous work, we utilize shared language properties between diverse software systems to develop an automatic system-agnostic approach for log instruction quality assessment, defined as 1) log level assignment and 2) sufficient linguistic structure assessment.

### 6 CONCLUSION
Writing log instruction with sufficient quality is challenging due to the absence of complete logging guidelines and developers incomplete understanding of system complexity. In this work, we address the problem of automating log quality assessment. We first do a preliminary study on nine software systems to study the quality properties of the log instructions. The results of our study identified 1) log level assignment and 2) sufficient linguistic structure assessment as two quality properties identifiable solely by the static text of the log instruction. Based on our observations, we propose a deep learning-based approach for automatic log instruction quality assessment on a given target system. Our approach uses the static text and its linguistic structure representation to evaluate the two properties. In addition, we adopt an approach from explainable AI to reason about model predictions and give suggestions for potential improvements of the instruction. Our approach outperforms the considered baselines, achieving high accuracy for log level assignment (0.88) and a high $F_\beta$ score for sufficient linguistic structure assessment (0.99). The results highlight future research opportunities in using cross-systems log instructions not just in automatically assessing log instruction quality, but also to automatically enhance them (i.e., automatic log instruction writing).

### REFERENCES

[1] Titus Bank, Robert DeLine, Steven Drucker, and Danyel Fisher. 2016. The Bones of the System: A Case Study of Logging and Telemetry at Microsoft. In 2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C) Association for Computing Machinery, New York, NY, USA, 92–101.
[2] Jasmin Bogatinovski. 2022. QuLog. Github repo link. GitHub. Retrieved January 02, 2022 from https://github.com/qulog/QuLog
[3] Leo Breiman. 2001. Random Forests. Machine Learning 45, 1 (2001), 5–32.
[4] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack
[52] Chen Zh, Jianwei Yin, Shuiguang Deng, Maoxin Ye, Min Fu, and Tao Xie. 2019. An Exploratory Study of Logging Configuration Practice in Java. In 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE Press, New York, USA, 459–469. https://doi.org/10.1109/ICSME.2019.00079

[53] Jieming Zhu, Pinjia He, Qiang Fu, Hongyu Zhang, Michael R. Lyu, and Dongmei Zhang. 2015. Learning to Log: Helping Developers Make Informed Logging Decisions. In Proceedings of the 37th International Conference on Software Engineering - Volume I. IEEE Press, River Street, USA, 415–425.