QuLog: Data-Driven Approach for Log Instruction Quality Assessment

Jasmin Bogatinovski
jasmin.bogatinovski@tu-berlin.de
Technical University Berlin
Berlin, Germany

Sasho Nedelkoski
nedelkoski@tu-berlin.de
Technical University Berlin
Berlin, Germany

Alexander Acker
alexander.acker@tu-berlin.de
Technical University Berlin
Berlin, Germany

Jorge Cardoso
jorge.cardoso@huawei.com
Huawei Munich Reserch Center
Munich, Germany

Odej Kao
odej.kao@tu-berlin.de
Technical University Berlin
Berlin, Germany

ABSTRACT

In the current IT world, developers write code while system operators run the code mostly as a black box. The connection between both worlds is typically established with log messages: the developer provides hints to the (unknown) operator, where the cause of an occurred issue is, and vice versa, the operator can report bugs during operation. To fulfil this purpose, developers write log instructions that are structured text commonly composed of a log level (e.g., “info", "error"), static text (“IP {} cannot be reached”), and dynamic variables (e.g. IP {}). However, opposed to well-adopted coding practices, there are no widely adopted guidelines on how to write log instructions with good quality properties. For example, a developer may assign a high log level (e.g., "error") for a trivial event that can confuse the operator and increase maintenance costs. Or the static text can be insufficient to hint at a specific issue. In this paper, we address the problem of log quality assessment and provide the first step towards its automation. We start with an in-depth analysis of quality log instruction properties in nine software systems and identify two quality properties: 1) *correct log level assignment* assessing the correctness of the log level, and 2) *sufficient linguistic structure* assessing the minimal richness of the static text necessary for verbose event description. Based on these findings, we developed a data-driven approach that adapts deep learning methods for each of the two properties. An extensive evaluation on large-scale open-source systems shows that our approach correctly assesses log level assignments with an accuracy of 0.88, and the sufficient linguistic structure with an F_1 score of 0.99, outperforming the baselines. Our study highlights the potential of the data-driven methods in assessing log instructions quality and aid developers in comprehending and writing better code.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICPC 2022, May 21–22, 2022, Pittsburgh, PA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

log quality, deep learning, log analysis, program comprehension

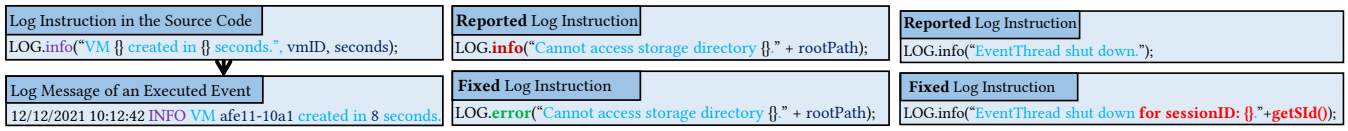
ACM Reference Format:
Jasmin Bogatinovski, Sasho Nedelkoski, Alexander Acker, Jorge Cardoso, and Odej Kao. 2022. QuLog: Data-Driven Approach for Log Instruction Quality Assessment. In *Proceedings of The 30th International Conference on Program Comprehension (ICPC 2022).* ACM, New York, NY, USA, 13 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Logging is 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



(a) An example of log instruction and generated message from the software system OpenStack.

(b) Jira issue HDFS-4048: Example of wrong log level assignment and its fix.

(c) Jira issue ZOOKEEPER-2126: Insufficient information/(linguistics) hurts event understanding.

Figure 1: (a) Example of log instruction. (b) & (c) Examples of issues related to log instruction quality and their fixes.

(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 aforenamed 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 aforenamed 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 log instructions, independent of the other properties of the source code (e.g., nearby 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 quality properties outperforming the baselines. The prediction

[1] https://issues.apache.org/jira/browse/LOG4J2-3166
[2] https://issues.apache.org/jira/browse/HDFS-4048
[3] https://issues.apache.org/jira/browse/ZOOKEEPER-2126

explainer has a low error for the correct prediction reason suggestion. Thereby, the experiments show that 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 system-agnostic 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 system-agnostic log quality assessment, which achieves high accuracy for log level assignment (0.88) and a high F_1 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.

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 assessment.

Table 1: Overview of the studied systems

Software System	Version	LOC	NOL
Cassandra	3.11.4	432K	1.3K
Elasticsearch	7.4.0	1.50M	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.

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 of information in a message [11]. In our case, 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 Shanon'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.

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 aforenamed 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. Examples of other Jira issues related to sufficient linguistic quality can be found in Appendix C.

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	interupted *
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, the safeness property 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 logging safeness). Identifying safeness in this context 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, composed 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 of the log instruction 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

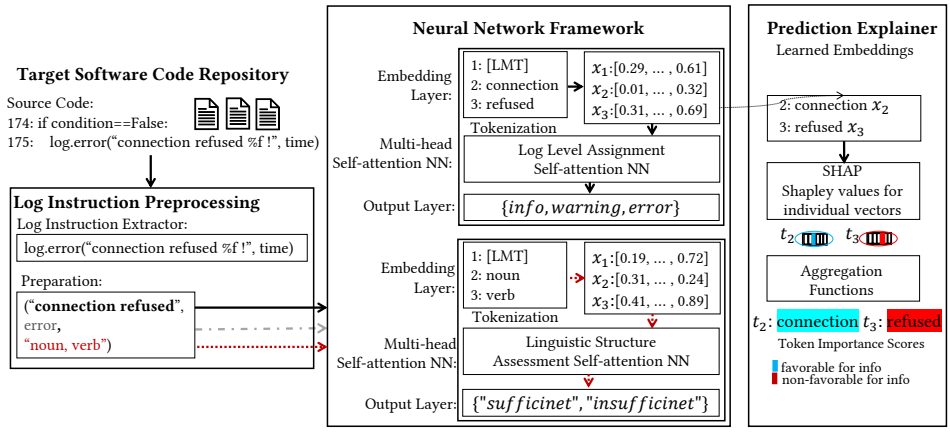


Figure 2: Internal architectural design of QuLog

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 the three QuLog's components.

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 from the remaining instruction parts. In addition, the static text is processed by the linguistic features extractor, to obtain its linguistic structure 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, Log4j (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 text 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 the preparation module is a set of tuples, where each tuple is composed of the static text of the log instruction, the linguistic structure of the static text and the log 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. To make the description easy to follow, 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 log level (sufficient/insufficient 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.

Log Instruction:	log.info("connection established!")	log.error("connection refused!")
Shapley Values (for class info):	connection: (1.21, 0.41, -0.12, 0.14) established: (2.12, 2.34, 3.01, 0.12)	connection: (0.21, -0.20, 0.42, 0.56) refused: (0.12, 2.30, 3.42, -5.22)
Token Importance Scores:	connection r_{11}= 1.34; $e_{11} = +$ established r_{12}= 4.36; $e_{12} = +$	connection r_{21} = 0.76; $e_{21} = +$ refused r_{22} = 6.65; $e_{22} = -$

Figure 3: Prediction Explainer working procedure example

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. 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 Shapely 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 Shapely 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 Shapely 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.

$$r(t_{ji}) = ||S_{ji}(t_{ji})||_2^2 \tag{1}$$

$$e(t_{ji}) = sign[\max_k |S_{jik}(t_{ji})|] \tag{2}$$

where $r(t_{ji})$ is the token (t_i) importance score intensity, $e(t_{ji})$ is the token importance sign for the log instruction l_j. S_{jik} denotes the Shapely 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 Shapely 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 Shapely 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 for the class "info" ($e_{22} = -$). 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), examined by developers in that order.

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 different systems from the following topics: Java, Python, Angular, Ruby, and PHP, selecting 7039 systems. 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. The usage of this data is described in the corresponding evaluation setting where used.

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 of the predictions. Due to the imbalances of the target classes (e.g., different systems have a diverse number of "error", "warning", and "info" instructions), accuracy can be misleading [17]. Therefore, we further considered *precision*, *recall* and F_1 scores. *Precision* evaluates the fraction of correct predictions out of all class predictions. *Recall* evaluates the correct predictions out of all true class predictions. F_1 is the harmonic mean of the precision and recall evaluating the trade-off between correct class predictions and the miss-classifications [17]. Additionally, we considered *specificity* in the binary classification setting. It is the measure of correct predictions of the negative class. The aforenamed 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

Table 4: Evaluation on log level quality assessment

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

(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 set the number of heads to two, the model size to 16, the number of layers is set to two, and the maximal number of tokens to $max\_len = 50$. For training the model we used Adam optimizer [26] with learning rate10^{-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 given input instruction. To evaluate the correctness of the log level decisions, we use accuracy. Comparing QuLog-8 against DeepLV it is seen that it is outperforming it in 8/9 systems while failing to do so in 1/9 (HBase) systems. Comparing QuLog-8 against SVM shows that QuLog performs better on 6/9 datasets, performs worse in 2/9 systems and ties on 1/9 (Flink). The evaluation criteria show that our approach is useful in assessing the correctness of log levels for the considered systems outperforming the baselines.

Next, we compare QuLog-8 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. Examples of when QuLog is outperforming the baselines can be found in Appendix A.

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 assignment. 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 F_1, 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*.

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 F_1 score

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

Scores	Scenario	QuLog	DeepLV	BERT_SVM
F_1	IE	**0.88±0.03**	0.82±0.02	0.87±0.04
	IWE	**0.73±0.03**	0.67±0.03	**0.73±0.04**
	IW	**0.68±0.06**	0.61±0.08	0.64±0.05
	WE	**0.61±0.04**	0.56±0.06	0.54±0.05
Precision	IE	0.88±0.02	0.79±0.04	**0.92±0.01**
	IWE	0.72±0.03	0.66±0.03	**0.74±0.05**
	IW	**0.75±0.04**	0.72±0.06	0.58±0.05
	WE	**0.69±0.09**	0.59±0.08	0.51±0.07
Recall	IE	**0.89±0.05**	0.86±0.06	0.84±0.06
	IWE	**0.73±0.03**	0.68±0.03	**0.73±0.04**
	IW	0.62±0.08	0.54±0.1	**0.72±0.09**
	WE	0.56±0.07	0.54±0.08	**0.59±0.08**

(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 (BERT) [12]. We train QuLog's linguistic quality assessment part with the same values of the hyperparameters as for the log level quality assessment, with setting the batch size to 64. As evaluation criteria, we used F_1 and specificity. Additional evaluation against rule-based approaches can be found in Appendix B.

Table 7: Sufficient linguistic structure assessment (performance evaluation)

	F_1			Specificity		
System	QuLog	SVM	RF	QuLog	SVM	RF
Cassandra	1.00	0.99	0.99	1.00	1.00	0.96
HBase	0.96	0.96	0.97	0.97	0.94	0.92
Kafka	0.99	0.98	0.92	1.00	1.00	0.74
Zookeeper	0.99	0.99	0.98	1.00	0.98	0.94
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 F_1 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 combing these results with the high performance on F_1 (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 manually investigated by two developers and modified such that a manually selected word of the static text is replaced with its antonym. This changes the event description creating an opposite event. For example, we start with the original static text "Connection established" with the original log level "info". We change the token "established" into its antonym "refused", obtaining a modified static text, i.e., "Connection refused" and modified word "refused". Since the modified token is an antonym, we change the original log level ("info") into the modified log level ("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 F_1 score 0.88) as opposed to the IWE model (average F_1 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 IW 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.

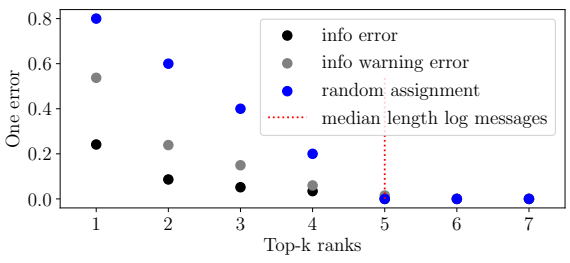


Figure 4: Evaluation of the explanation module

Table 8: Use Case Study Results

	Log Instructions	Log Level Recommendations	Linguistic Structure Recommendations
System 1	63	52 (5/11)	56 (4/7)
System 2	138	120 (8/18)	133 (3/5)

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 aforenamed 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 of the static text concerning the different log levels and 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 assessments.

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 assessments 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_1 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 Barik, 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. 2021. *QuLog: Github repo link*. Anonymous. 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 Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, Vol. 33. Curran Associates, Inc., 1877–1901.

[5] Jeanderson Cândido, Haesen Jan, Maurício Aniche, and Arie van Deursen. 2021. An Exploratory Study of Log Placement Recommendation in an Enterprise System. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE Computer Society, Los Alamitos, CA, USA, 143–154. https://doi.org/10.1109/MSR52588.2021.00027

[6] Boyuan Chen and Zhen Ming (Jack) Jiang. 2017. Characterizing logging practices in Java-based open source software projects – a replication study in Apache Software Foundation. *Empirical Software Engineering* 22 (2017), 330–374.

[7] Boyuan Chen and Zhen Ming Jiang. 2019. Extracting and Studying the Logging-Code-Issue- Introducing Changes in Java-Based Large-Scale Open Source Software Systems. *Empirical Softw. Engg.* 24 (2019), 2285–2322.

[8] Tse-Hsun Chen, Stephen W. Thomas, and Ahmed E. Hassan. 2016. A Survey on the Use of Topic Models When Mining Software Repositories. *Empirical Softw. Engg.* 21 (2016), 1843–1919.

[9] Marcello Cinque, Domenico Cotroneo, and Antonio Pecchia. 2013. Event Logs for the Analysis of Software Failures: A Rule-Based Approach. *IEEE Transactions on Software Engineering* 39 (2013), 806–821.

[10] Corinna Cortes and Vladimir Vapnik. 1995. Support-vector networks. *Machine Learning* 20 (1995), 273–297.

[11] Thomas M. Cover and Joy A. Thomas. 2006. *Elements of Information Theory (Wiley Series in Telecommunications and Signal Processing)*. Wiley-Interscience, USA.

[12] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. , 4171—4186 pages. https://doi.org/10.18653/v1/N19-1423

[13] Rui Ding, Hucheng Zhou, Jian-Guang Lou, Hongyu Zhang, Qingwei Lin, Qiang Fu, Dongmei Zhang, and Tao Xie. 2015. Log2: A Cost-Aware Logging Mechanism for Performance Diagnosis. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*. USENIX Association, USA, 139–150.

[14] Edward Finegan. 2014. *Language: Its structure and use* (7 ed.). Cengage Learning, Florence, AL. 289 pages.

[15] The Apache Software Foundation. 2022. *Logging Service Project*. Appache. Retrieved January 2, 2022 from https://logging.apache.org/

[16] Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. 2014. Where Do Developers Log? An Empirical Study on Logging Practices in Industry. In *Companion Proceedings of the 36th International Conference on Software Engineering*. Association for Computing Machinery, New York, NY, USA, 24–33.

[17] Eva Gibaja and Sebastián Ventura. 2015. A Tutorial on Multilabel Learning. *Comput. Surveys* 47, 3 (2015), 52:1–52:38.

[18] Anu Han, Chen Jie, Shi Wenchang, Hou Jianwei, Liang Bin, and Qin Bo. 2019. An Approach to Recommendation of Verbosity Log Levels Based on Logging Intention. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, New York, USA, 125–134.

[19] Robert J. Hand, David J.and Till. 2001. A Simple Generalisation of the Area Under the ROC Curve for Multiple Class Classification Problems. *Machine Learning* 45 (2001), 171–186.

[20] Ahmed E. Hassan. 2009. Predicting faults using the complexity of code changes. In *2009 IEEE 31st International Conference on Software Engineering*. IEEE Computer Society, USA, 78–88. https://doi.org/10.1109/ICSE.2009.5070510

[21] Mehran Hassani, Weiyi Shang, Emad Shihab, and Nikolaos Tsantalis. 2018. Studying and Detecting Log-Related Issues. *Empirical Softw. Engg.* 23 (2018), 3248–3280. https://doi.org/10.1007/s10664-018-9603-z

[22] Pinjia He, Zhuangbin Chen, Shilin He, and Michael R. Lyu. 2018. Characterizing the Natural Language Descriptions in Software Logging Statements. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. Association for Computing Machinery, New York, NY, USA, 178–189.

[23] Matthew Honnibal, Ines Montani, Sofie Van Landeghem, and Adriane Boyd. 2020. *spaCy: Industrial-strength Natural Language Processing in Python*. Explosion.ai. https://doi.org/10.5281/zenodo.1212303

[24] Thorsten Joachims. 2005. A Support Vector Method for Multivariate Performance Measures. In *Proceedings of the 22nd International Conference on Machine Learning*. Association for Computing Machinery, New York, NY, USA, 377–384.

[25] Suhas Kabinna, Cor-Paul Bezemer, Weiyi Shang, Mark D. Syer, and Ahmed E. Hassan. 2018. Examining the Stability of Logging Statements. *Empirical Software Engineering* 23 (2018), 290–333.

[26] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization.

[27] Heng Li, Tse-Hsun (Peter) Chen, Weiyi Shang, and Ahmed E. Hassan. 2018. Studying Software Logging Using Topic Models. *Empirical Softw. Engg.* 23 (2018), 2655–2694.

[28] Heng Li, Weiyi Shang, Brayan Adams, Mohammed Sayagh, and Ahmed E. Hassan. 2020. A Qualitative Study of the Benefits and Costs of Logging from Developers' Perspectives. *IEEE Transactions on Software Engineering* 47 (2020), 2858–2873. https://doi.org/10.1109/TSE.2020.2970422

[29] Heng Li, Weiyi Shang, and Ahmed E. Hassan. 2017. Which Log Level Should Developers Choose for a New Logging Statement? *Empirical Softw. Engg.* 22, 4 (2017), 1684–1716.

[30] Jian Li, Yue Wang, Michael R. Lyu, and Irwin King. 2018. Code Completion with Neural Attention and Pointer Networks. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*. AAAI Press, 4159–25.

[31] Zhenhao Li. 2020. *Towards Providing Automated Supports to Developers on Writing Logging Statements*. Association for Computing Machinery, New York, NY, USA, 198–201. https://doi.org/10.1145/3377812.3381385

[32] Zhenhao Li, Tse-Hsun (Peter) Chen, and Weiyi Shang. 2020. Where Shall We Log? Studying and Suggesting Logging Locations in Code Blocks. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. Association for Computing Machinery, New York, NY, USA, 361–372.

[33] Zhenhao Li, Tse-Hsun Peter Chen, Jinqiu Yang, and Weiyi Shang. 2021. Studying Duplicate Logging Statements and Their Relationships with Code Clones. *IEEE Transactions on Software Engineering* (2021). https://doi.org/10.1109/TSE.2021.3060918

[34] Zhenhao Li, Heng Li, Tse-Hsun Chen, and Weiyi Shang. 2021. DeepLV: Suggesting Log Levels Using Ordinal Based Neural Networks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE Press, NJ, USA, 1461–1472. https://doi.org/10.1109/ICSE43902.2021.00131

[35] Seppo Linnainmaa. 1976. Taylor expansion of the accumulated rounding error. *BIT Numerical Mathematics* 16 (1976), 146–160.

[36] Scott Lundberg. 2019. *SHAP Github Implementation*. GitHub. Retrieved January 2, 2022 from https://github.com/slundberg/shap

[37] Scott M. Lundberg, Gabriel Erion, Hugh Chen, Alex DeGrave, Jordan M. Prutkin, Bala Nair, Ronit Katz, Jonathan Himmelfarb, Nisha Bansal, and Su-In Lee. 2020. From local explanations to global understanding with explainable AI for trees. *Nature Machine Intelligence* 2, 1 (2020), 56–67.

[38] Antonio Pecchia, Marcello Cinque, Gabriella Carrozza, and Domenico Cotroneo. 2015. Industry Practices and Event Logging: Assessment of a Critical Software

Development Process. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. IEEE Press, New York, USA, 169–178.
[39] QOS. 2022. *Simple Logging Faced for Java.* QOS. Retrieved January 2, 2022 from https://www.slf4j.org/
[40] Weiyi Shang, Zhen Ming Jiang, Bram Adams, Ahmed E. Hassan, Michael W. Godfrey, Mohamed Nasser, and Parminder Flora. 2014. An exploratory study of the evolution of communicated information about the execution of large software systems. *Journal of Software: Evolution and Process* 26 (2014), 3–26.
[41] He Shilin, Zhu Jieming, He Pinjia, and Lyu Michael, R. 2016. Experience Report: System Log Analysis for Anomaly Detection. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE Press, River Side, USA, 207–218.
[42] Julius Sim and Chris C Wright. 2005. The Kappa Statistic in Reliability Studies: Use, Interpretation, and Sample Size Requirements. *Physical Therapy* 85 (2005), 257–268.
[43] A. Plate Tony. 1995. Holographic reduced representations. *IEEE Transactions on Neural Networks* 6, 3 (1995), 623–641. https://doi.org/10.1109/72.377968
[44] van der Aalst and et al. 2012. Process Mining Manifesto. In *Business Process Management Workshops*. Springer Berlin Heidelberg, Berlin, Heidelberg, 169–194.
[45] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, undefinedukasz Kaiser, and Illia Polosukhin. 2017. Attention is All You Need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*. Curran Associates Inc., Red Hook, NY, USA, 6000–6010.
[46] Andrew Watters and Sarah Boslaugh. 2008. *Statistics in a Nutshell: A Desktop Quick Reference.* O'Reilly Media, USA.
[47] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael M. Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. 2012. Be Conservative: Enhancing Failure Diagnosis with Proactive Logging. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. USENIX Association, Hollywood, CA, 293–306.
[48] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. 2012. Characterizing Logging Practices in Open-Source Software. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, River Street, USA, 102–112.
[49] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. 2012. Improving Software Diagnosability via Log Enhancement. *ACM Trans. Comput. Syst.* 30, 1 (2012), 3–14.
[50] Yi Zeng, Jinfu Chen, Weiyi Shang, and Tse-Hsun (Peter) Chen. 2019. Studying the characteristics of logging practices in mobile apps: a case study on F-Droid. *Empirical Software Engineering* 24 (2019), 3394–3434.
[51] Xu Zhao, Kirk Rodrigues, Yu Luo, Michael Stumm, Ding Yuan, and Yuanyuan Zhou. 2017. Log20: Fully Automated Optimal Placement of Log Printing Statements under Specified Overhead Threshold. In *Proceedings of the 26th Symposium on Operating Systems Principles*. Association for Computing Machinery, New York, NY, USA, 565–581.
[52] Chen Zhi, 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 1*. IEEE Press, River Street, USA, 415–425.

A SAMPLE ANALYSIS: WHEN QULOG OUTPERFORMS THE BASELINES?

One set of examples where we find that QuLog's log level prediction outperforms other methods is in the logs like "created with buffersize=* and maxpoolsize=*" (with original log level INFO, QuLog prediction is "info", "DeepLV" and "BERT_SVM" are predicting "warning") and "stopping the wall procedure store, isabort= | (self aborting)" (with original log level INFO, QuLog prediction is "info", "DeepLV" and "BERT_SVM" are predicting "error"). These two messages contain words like "maxpoolsize", "buffersize" or "isabort" which are not standard in general language. Therefore, the log representation by general language models is inferior in comparison to QuLog, which directly learns log representations from the static text. Since QuLog, is directly trained on log-specific words it can learn better representations and relate the words to the correct target.

When considering the linguistic comparison, as seen by the results, QuLog and BERT_SVM are performing similarly. We find one consistent example where QuLog performs better than SVM, i.e., when HBase is used for testing QuLog correctly predicts the linguistic group "VERB VERB" as insufficient. We hypothesise that QuLog can extrapolate this information having the groups "VERB" and "VERB NOUN VERB" as insufficient during training. The two baselines, SVM and RF are performing slightly differently on the same representation, we consider the observed difference to reside in the type of decision boundary they fit. SVMs are characterized to perform better in very high dimensional spaces (because of the properties of such spaces). RF is characterized by diverse volume pockets in different regions. Since the samples in the high dimensions are very far apart from one another the pocket regions created by RF lead to more miss-classifications.

B COMPARING QULOG LINGUISTIC MODULE AGAINST RULE-BASED APPROACH

The process of identification of sufficient linguistic structure was conducted as follows. Two human annotators examined the linguistic structure and the raw static texts organized by linguistic groups. A linguistic group is a set of static text instructions'. A linguistic group is identified by the sequence of POS tags. Each human annotator examined the individual static text within each group (within each of the randomly sampled 361 groups). Since each log message should describe an event verbosely and convey sufficient information on one side, and following the maxim of quality and quantity for short texts from general language properties, on the other side, the static text should have minimal linguistic structure for sufficient expression of the information. By examining the raw static text within each group the annotators relate their understanding of the sufficient information a log message has. They assign labels 0 for "linguistically sufficient" or 1 for the "linguistically insufficient" group. Then the labels provided by the two human annotators are compared, and the linguistic groups with overlapping labels are retained. Each of the static texts is then labeled with the label of the linguistic group associated with it. However, the model is trained using the linguistic group representation of the static text obtained by the pos tag. The labels can be used as rules what is a linguistically good and bad static text.

The generated rules can be used to detect logs with sufficient and insufficient structure (that is how they are generated). Whenever we have a new system, we calculate their linguistic representations and match them against the rules. Any match is considered as a log with insufficient quality. For example, if we find a static text with "NOUN NOUN" from the new system, we can say that the static text is of "insufficient" quality. However, we found examples, where QuLog maybe interpolates between two nearby rules, and correctly identify a rule from a new project that was not part of the training data. For example, when HBase is in the test set, (the rules in the training set are extracted from other systems), the rules do not cover the linguistically insufficient group "VERB VERB". In contrast, QuLog correctly predicts this group as "insufficient" because of its similarities with the linguistically insufficient groups of "VERB" and 'VERB NOUN VERB' (that are slightly different from "VERB VERB"). Since QuLog has strong performance, and can

interpolate between rules we opt for the given design. Nevertheless, there is still much space for improving the input training dataset that will lead to new insights.

C LINGUISTIC QUALITY ASSESSMENT ADDITIONAL EVIDENCE

The idea for "sufficient linguistic quality" assesment is built around Jira issues like ZOOKEEPER-2126, ZOOKEEPER-3659 and Zookeeper-259. They show that the minimal information in the static text hurts comprehensibility. By relating the information about minimality (e.g., minimal number of tokens, linguistic token categories) with the (in)sufficient linguistic structure of the static text we can evaluate the type of linguistic categories (e.g., verb, nouns, adjectives, and similar) participating in the verbose and comprehensive description of the events. By enriching the event description with additional linguistic properties, the log messages are easier for reading, comprehension, and contain sufficient verbose information.

In the following, we further describe two Jira issues related to linguistic structural problems. In ZOOKEEPER-3659 it is reported that WatchManagerFactory log is not sufficiently readable. The fix of this issue is to change the prior static text *Using org.apache.zookeeper.server.watch.WatchManager as watch manager*, into *dataWatches is using org.apache.zookeeper.server.watch.WatchManager as watch manager.*. From the linguistic perspective this means that the linguistic structure ['VERB', 'PUNCT', 'ADP', 'NOUN', 'NOUN'] is transformed into [**'NOUN', 'AUX',** 'VERB', 'PUNCT', 'ADP', 'NOUN', 'NOUN']. The additional linguistic concepts improve the comprehensibility of the event.

Similarly in the Jira issue Zookeeper-259 despite the correction of the log levels, the text in the log instructions is changed as well. One example is the change on lines 524, through 526. Specifically, the prior static text "Got ping sessionid:0x" was replaced with "Got ping response for sessionid:0x". Linguistically speaking this means that the text is changed from ['AUX', 'VERB', 'NOUN'] into ['VERB', 'NOUN', 'NOUN', 'ADP', 'NOUN'] which improves the comprehensibility and makes the log line easer for the operators to understand.