AI-enabled Log Analysis for Improving IT System Dependability

vorgelegt von
M. Sc.
Jasmin Bogatinovski

an der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
- Dr.-Ing. -

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender: Prof. Dr. Sahin Albayrak
Gutachter: Prof. Dr. Odej Kao
Gutachter: Prof. Dr. Roberto Natella
Gutachter: Prof. Dr. David Bermbach
Gutachter: Prof. Dr. Gjorgji Madjarov

Tag der wissenschaftlichen Aussprache: 20. Februar 2023

Berlin 2023

Dedication

To my family.

Acknowledgements

First and foremost, I would like to thank my supervisor Prof. Dr. Odej Kao. Without our vivid discussions and insightful questions, this work would not have been done. Prof. Odej, I am sincerely thankful for your patience, supportiveness, thoughtfulness, and guidance through my PhD. Thank you for all the given chances for personal and professional development. Thank you for accepting me as a part of your group and showing me various aspects of academia and life. Many of the shared life lessons and experiences will always stay with me and for sure will help me in my future professional and private endeavours. I am honoured, glad and lucky to have you as my supervisor. I sincerely believe that the superb workshops we organized will stick with you as small, but memorable wins.

I would also like to thank Jorge Cardoso from Huawei Munich Research, with whom I have had the pleasure to work for more than three years on high-impact industry problems. I am thankful for his support through all aspects of our collaborations including the unselfish sharing of his rich expertise and experience.

The thesis was financially supported by Huawei Technologies Co., Ltd. I am grateful for their recognition as a suitable collaborator during my PhD.

My special gratitude extends to Sasho Nedelkoski. I am very grateful for our collaborations and the many discussions on different topics, which improved my professional abilities and helped me to become a better person.

I would also like to thank my colleagues that helped me with the adaptation, advice, and all the other peculiarities within a working environment. Particularly, I would call out the names of Alexander Acker, Soeren Becker, Yevhen Yazvinskyi, Petar Ilijeski, Mihail Bogojeski, Li Wu, Dominik Scheinert, Jonathan Bader, Kevin Styp-Rekowski and Florian Schmidt from the Technical University Berlin, Ilya Shakat, Soroush Harari, Qiao Yu and Erekle Shishniashvili from Huawei Munich Research, Tome Eftimov, Ana Kostovska, Ljupčo Todorovski and Dragi Kocev from Jožef Stefan Institute, Ljubljana

Slovenia. Many thanks to all the people who voluntarily or involuntarily volunteered in the long-lasting process of my studies.

Special thanks to Jana Bechstein for helping me with all the administration and for the pleasant conversations and warm advices when there was nobody else. Jana, thank you.

The greatest burden of my PhD journey fell onto my wonderful parents and lovely sister. Thank you for being brave and supportive in every aspect of my life. Thank you for your contribution to shaping me to become the person I am today. My endless love belongs to you.

Last but not least, I am thankful to Alisa Krstova for her love, support and patience during the greatest hardships of my PhD process. Thank you.

Abstract

Modern IT systems play an indispensable role in industrial infrastructure and affect human society, as billions of users and devices constantly compute, exchange and store data. Their characteristics, such as large complexity, fast evolution, and geo-distributed development, among others, challenge the availability and the correctness of service offerings while increasing failure proneness. Failure to deliver the correct service can have severe implications. This is particularly the case for critical systems in medicine, transportation, or energy, leading to hazardous effects. The increased complexity surpasses the developers' and operators' capabilities for timely issue resolution increasing the chance of frequent failure impact.

To support the system development and operation, as means to ensure the provisioning of correct service that can justifiably be trusted (*system dependability*), automation of different tasks is needed. One important aspect of automation is the IT system's capability to externalise the system state via monitoring data such as system logs. These data are used by intelligent methods that can learn to discern frequent normal and anomalous patterns from the data. Therefore, intelligent methods can automate parts of the development and operational processes, e.g., by generating alerts about potential issues. In this context, Artificial Intelligence for IT operations (AIOps) emerged as a research area concerned with using the system (e.g., source code) and monitoring data (e.g., system logs) and methods from artificial intelligence (AI), big data, machine learning and data mining to support the automation of IT operational activities.

This thesis introduces AI-enabled methods that address different AIOps tasks during system development and operation. The methods focus on the logging process and system logs as an intrinsic data source for the IT systems. From a system development perspective, the main contributions reside in formalizing and addressing the problem of log instruction quality, as logs with sufficient quality are a precondition for successfully tackling downstream log-related tasks. (1) The thesis proposes a deep learning-based method to automatically evaluate the quality of log instructions from the system's source code. From a system operation perspective, the thesis contributes by proposing novel methods for log analysis, specifically, log-based anomaly detection. The log-based anomaly detection methods learn anomaly-related log properties that improve the (2) sentiment and (3) sequential log representations. This category of methods studies how leveraging the individual log properties impacts anomaly detection and classification in modern IT systems. The extensive evaluations with data from open-source, production systems, and testbeds show the usefulness of the proposed methods in addressing the challenges of modern IT systems while demonstrating desirable practical properties. The proposed methods and results were published in peer-reviewed international conferences, while parts were patented at the European Patent Office.

Zusammenfassung

Moderne IT-Systeme spielen eine unverzichtbare Rolle in industriellen Infrastrukturen und beeinflussen menschliche Gesellschaften, da Milliarden von Nutzern und Geräten ständig Daten berechnen, austauschen und speichern. Diese Systeme weisen Merkmale wie hohe Komplexität sowie schnelle und geografisch verteilte Entwicklung auf, womit sie eine Herausforderung für die Verfügbarkeit und Korrektheit von Dienstangeboten darstellen und die Fehleranfälligkeit erhöhen. Wird ein Dienst nicht korrekt erbracht, kann dies schwerwiegende Folgen haben. Dies gilt insbesondere für kritische Systeme in der Medizin, im Transportwesen oder im Energiesektor, was zu gefährlichen Auswirkungen führen kann. Die zunehmende Komplexität übersteigt die Möglichkeiten der Entwickler und Betreiber zur rechtzeitigen Problemlösung und erhöht die Wahrscheinlichkeit häufiger Ausfälle.

Zur Unterstützung der Systementwicklung, des Systembetriebs und als Mittel zur Sicherstellung der Bereitstellung eines korrekten Dienstes, dem man berechtigterweise vertrauen kann (Systemzuverlässigkeit), ist die Automatisierung verschiedener Aufgaben erforderlich. Ein wichtiger Aspekt der Automatisierung ist die Fähigkeit des IT-Systems, den Systemzustand über Überwachungsdaten wie System-Logs zu externalisieren. Diese Daten werden von intelligenten Methoden verwendet, die lernen können, normale und anomale Muster aus den Daten zu erkennen. Intelligente Methoden können daher Teile der Entwicklungs- und Betriebsprozesse automatisieren, indem sie z. B. Warnungen oder Korrekturmaßnahmen zu potenziellen Problemen erzeugen. In diesem Zusammenhang hat sich Künstliche Intelligenz für den IT-Betrieb (AIOps) als Forschungsgebiet herauskristallisiert, das sich mit der Nutzung von System- (z. B. Quellcode) und Überwachungsdaten (z. B. System-Logs) und Methoden aus den Bereichen KI, Big Data, maschinelles Lernen und Data Mining beschäftigt, um die Automatisierung von IT-Betriebsaktivitäten zu unterstützen.

In dieser Arbeit werden KI-gestützte Methoden vorgestellt, die verschiedene AIOps-Aufgaben während der Systementwicklung und des Betriebs adressieren. Die Methoden konzentrieren sich auf den Log-Prozess und die System-Logs als intrinsische Datenquelle für die IT-Systeme. Aus der Perspektive der Systementwicklung liegen die Hauptbeiträge in der Formalisierung und Behandlung des Problems der Qualität von Log-Instruktionen, da Logs mit ausreichender Qualität eine Voraussetzung für die erfolgreiche Bewältigung nachgelagerter logbezogener Aufgaben sind. (1) Die Arbeit schlägt eine auf Deep Learning basierende Methode zur automatischen Bewertung der Qualität von Log-Anweisungen aus dem Quellcode des Systems vor. Aus Sicht des Systembetriebs leistet die Arbeit einen Beitrag, indem sie neuartige Methoden zur Log-Analyse vorschlägt, insbesondere zur log-basierten Anomalie-Erkennung. Die log-basierten Methoden zur Erkennung von Anomalien lernen anomalitätsbezogene Log-Eigenschaften, die die (2) Sentiment-

und (3) sequentielle Darstellung von Logs verbessern. Diese Kategorie von Methoden untersucht, wie sich die Nutzung der individuellen Log-Eigenschaften auf die Anomalieerkennung und -klassifizierung in modernen IT-Systemen auswirkt. Die umfangreichen Auswertungen mit Daten aus Open-Source- und Produktionssystemen sowie Testumgebungen unterstreichen die Nützlichkeit der vorgeschlagenen Methoden bei der Bewältigung der Herausforderungen moderner IT-Systeme und demonstrieren gleichzeitig die wünschenswerten praktischen Eigenschaften. Die vorgeschlagenen Methoden und Ergebnisse wurden von Fachleuten überprüft und auf internationalen Konferenzen in Form von Fachbeiträgen veröffentlicht, sowie in Teilen beim Europäischen Patentamt patentiert.

Contents

List of Tables

List of Figures

Abbreviations

AD	Anomaly Detection
AIOps	Artificial Intelligence for IT operations
AOP	Aspect Oriented Programming
AI	artificial intelligence
ASGD	Alternating Stochastic Gradient Descent
DevOps	Development and Operation
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DL	Deep Learning
DT	Decision Trees
EPO	European Patent Office
GRU	Gated Recurrent Unit
HMM	Hidden Markov Model
kNN	k-Nearest Neighbour
LDAP	Lightweight Directory Access Protocol
LSTM	Long Short Term Memory
ML	Machine Learning
IT	Information Technologies
NEP	Next Event Prediction
NLP	Natural Language Processing
OM	Operation and Maintenance
OS	Operating System
PA	Parsing Accuracy
perAD	Performance Anomaly Detection
PCA	Principle Component Analysis
PU	Positive Unlabeled
POS	Part of Speech
PIDs	Process Identifiers
RF	Random Forest
RELU	Rectifier Linear Unit
semAD	Semantic Anomaly Detection
semATC	Semantic Anomaly Type Classification
seqAD	Sequential Anomaly Detection
seqATC	Sequential Anomaly Type Classification
SLA	Service Level Agreements
SRE	Site Reliability Engineering
SSD	Solid State Drives
SVM	Support Vector Machines
TFILF	Term Frequency Inverse Location Frequency

Chapter 1

Introduction

Contents

Modern IT systems play an important role in the industrial and human-social infrastructure. Search engines, instant messaging applications, banking software, e-commerce platforms, and others, are examples of modern IT systems humans vastly rely on. They support the continual computation, exchange and storage processes between billions of devices and users. Thereby, humans indispensably rely on the dependability of the systems - as a system's ability to deliver a service that is justifiably trusted. To support the ever-growing industrial and social demands, the overall IT ecosystem is projected to increase the number of interconnected components while diversifying the underlying hardware. For example, market analysis reports from Cisco and Gartner suggest that the number of interconnected devices by 2025 will increase to 27.3-30 billion [48, 77]. From a software development perspective, the support of the development and scalability demands shifts the focus from a monolithic system design toward a decentralized service-based design [159]. Decentralized software allows agile and reliable development of dedicated services. This improves scaling and enables large data volume processing. While the decentralized design of modern IT systems significantly improves business agility, it comes at the cost of magnified system complexity [130].

The omnipresence of IT systems in daily human activities imposes high user expectations for system dependability. However, the inevitable weaknesses in hardware and software lead to failures. *Failures* are events where a system component (hardware or software) omits the execution of the expected or required action, making them one of

the key threats to system dependability [4]. The downtime they cause can lead to customer dissatisfaction and economic losses [70]. Failures in emerging critical systems (e.g., autonomous driving and smart cities) can endanger human safety and threaten human lives. For example, software issues in the intelligent software of a Tesla (an electric car producing company) vehicle are speculated to be a direct cause for collisions with fatal consequences [138]. In addition to the term failure, the term *anomaly* is used to refer not just to failures, but to additional events that describe a degraded state [2]. As the richness of the IT systems in terms of both hardware and software grows, so does the failure proneness. Different studies examine the anatomy of the anomalies in large-scale systems (e.g., HPC systems, clouds) [53] and different types of services and components (e.g., virtual machines, network components, storage subsystems, and data mining services) [163, 171]. They show that failures are frequent and their nature is very diverse.[1] Recognizing the persistence and the polymorphic nature of the anomalies, their frequency, and hazardous effects require non-trivial efforts and expertise from the developers, and operators of the IT systems to *detect* them and guarantee systems' dependability. Therefore, the correct detection and classification of the system state different than the normal (i.e., *anomaly detection and classification*) are important for trusted service delivery.

To regulate and improve the system's dependability, developers and Operation and Management (O&M) teams rely on observability data obtained during system execution [70]. Two commonly used categories of observability data are the metric data (e.g., CPU usage) and system log messages (logs, e.g., "VM created") [157]. System logs are often used as observability data for on-field analysis, as they are intrinsic monitoring tools for every computer system (e.g., control register status bits in the CPU) [70, 92]. They are generated from log instructions that developers insert in the source code to visualise important system events and to give hints to the operators running the system as a black box. For example, the log instruction: log.info("VM took %f seconds to spawn.", createSeconds) shows the time needed to create a virtual machine. It is composed of a log level (i.e., *info*), static text describing the event (i.e., *"VM took $<*$*$>$ seconds to spawn."*), and variable parameter (i.e., *createSeconds*), denoting important variable runtime information. Logs give meaningful clues for potential failure, as they are semantically rich data written by humans for humans (i.e., they are human-centric). For example, when a switch generates the log 'System is rebooting now', the operator understands by the semantics that the *switch* is *failing*, and obtains a hint that the potential anomaly may be caused by *switch* rebooting. In addition, despite the semantics, the event co-occurrence is another log property adding to log richness as a data source. For example, several sequential repetitions of the two logs a) "Interface change state to up", and b) "Interface change state to down", reveal that the interface is *flapping*. Given that the most frequent

[1] Appendix A shows a replication study we performed to study the failure's impact on online service dependability.

reason for a *flapping* interface is a *bad cable connection*, the operators can diagnose and classify the failure on time, and act accordingly [29]. Notably, different anomalies reflect in various ways in logs [44]. Therefore, analysing the log properties jointly over the different properties (i.e., semantics and sequential) should provide maximal visibility over the failures.

The practical importance of logs has led to the appearance of the research area of log analytics. Log analytics covers the whole cycle of the logging process during both development and operation, including the creation, storage, and analysis of logs [70, 183]. Log analytics aids development by improving the writing of log instruction in the source code, i.e., the *logging code composition*. During operation, log analytics is concerned with the processing and analysis of the generated logs to detect and classify anomalies, find root causes, predict future anomalies, and similar [70]. For example, the task of anomaly detection considers detecting specific system behaviours where the system produces unexpected out-of-normal behaviour. In the task of anomaly classification (as an example of another operational task), the aim is to relate the past experiences of the anomalies with a particular anomaly class enabling the reuse of past anomaly-resolving techniques. Traditionally, in the context of supporting system dependability, log analytics depends on the domain expertise of developers and operators alike [187]. For example, during development, the log instruction writing (e.g., choosing the log instruction level) is intrinsically subjective as it depends on the experience of the developers [100]. During runtime, the system operation is concerned with the analyses of the generated logs to extract rules for the different log analytics tasks by relying on heuristics (e.g., search for "error" log levels or words like "failure", and "rejected") to detect or classify anomalies [102].

Modern systems' characteristics, i.e., increased complexity due to the increased scale, the heterogeneous hardware/software, and increased interconnectivity, render the traditional approaches based on ad-hoc rules *inefficient* and *ineffective* [131]. The inefficiency and ineffectiveness are direct consequences of the need to constantly update, or the inability to construct comprehensive rules. To account for the characteristics of modern IT systems, the automation of log analytics is being vastly researched [70, 183, 187]. The core idea of automation is inventing and using intelligent methods from machine learning, big data, and artificial intelligence on log-related data to learn task-specific patterns (as learned rules). Therefore, automation replaces handcrafted rules while improving the robustness and capabilities for extracting comprehensive patterns. Ultimately, it aids human efforts to improve system dependability [130].

Despite the vast body of research on automatic log analysis, there are several aspects of the automation of log analytics that can be further improved (during both development and operation). As modern IT systems are complex and often developed by many developers with different levels of expertise and different understanding of logging purposes,

the logging code compositions are of varying quality. Low-quality logs can hide or even present wrong information which can hurt log usability [23]. Therefore, the automation of the quality assessment of log instruction emerges as an important problem during development. Log instruction quality assessment refers to the evaluation of the alignment of the log instructions properties against the log instructions with assumed good quality. The automation of log quality assessment during development should consider and is therefore challenged by (a.1) heterogenous events (e.g., different vocabulary for diverse events, different writing styles) and (a.2) different programming languages used for developing a single system. These aspects further raise the challenge of (a.3) which logging code composition quality assessment properties can be automated.

From a system operation perspective, the automation of log analysis addresses different tasks. The timely detection and classification of anomalies are of particular interest, as they are the first step towards limiting the downtime duration, thus preventing Service Level Agreements (SLA) violations. Therefore, these two tasks are the main research focus in AIOps [130]. As the scale and update frequency of modern software increases, so does the rate of addition of (new) logs and log dependencies. This challenges the automation of methods, as it requires robust method performance, leading to many false alarms otherwise. Therefore, from the system operation perspective, the two challenges of (b.1) reducing the false alarm rates, and (b.2) limited usability of past experiences (as labels) exist. In addition, logs are complex data as they are both textual and sequential. Naturally, different anomalies affect the log data differently. The automatic methods, thereby, (b.3) are challenged with efficient utilization of the log properties (e.g., dealing with single log lines and log sequences). Considering the characteristics of modern IT systems, and the challenges from the system development (challenges a1 - a3) and system operation (challenges b1 - b3) perspectives impose the need for an automatic log analytics framework that supports the full logging cycle, from logging instrumentation to timely failure identification as support in improving system dependability.

1.1 Problem Statement

An automatic log analytics framework supports the two stages of the system lifecycle, i.e., development and operation. The research objective of this thesis is to:

"Improve the IT system development, operation and dependability by developing intelligent methods for logging code composition, anomaly detection and classification with log data."

As development and operation are two different phases in the lifecycle of the system, they face unique challenges and are addressed independently. During system runtime, the anomalies are observable in different properties of the log data (e.g., single log lines or log sequences), which prompts the need to examine the individual properties separately.

Considering this, we decompose the objective of this thesis into three parts discussed in the following text.

Logging Code Composition Quality. Modern IT systems are developed by multiple developers with diverse experiences that have different understandings of logging a particular event. The diversity leads to the existence of events written with different styles, diverse vocabulary and originating from various programming languages, all of which impose difficulties on the automation of logging code composition. Similarly, it is challenging to identify the empirically testable instruction properties as constituents of the logging code composition subject to automation. We aim to address these challenges to support the writing of quality logging code.

Single Line Log Analysis. Operators analyse the static text and the variable parameters of the individual logs to detect and classify anomalies. The dynamic changes in IT systems result in fast system evolution leading to novel logs. Therefore, the anomaly detection methods should generalize well. As the anomalies can be reflected in the parameter values, better anomaly detectability is achieved by analysing the parameters as well. The first step for performance anomaly is to correctly parse the logs and extract the parameters and the events. In this regard, log parsing is expected to have robust performance for different systems as incorrect parsing can miss the relevant parameters and the anomalies will not be detected. We aim to ameliorate the usage of single log line properties (e.g., sentiment) and log parsing to improve the generalization and robustness in single log line analysis, (e.g., log parsing, and single log line anomaly detection).

Sequential Log Analysis. As developers may have an insufficient understanding of the complexities of the running system environment during development, not all failures can be logged. Consequently, there is insufficient anomaly logging coverage. As a result, the single log line analysis misses anomalies that are not explicitly logged. Nevertheless, some anomalies are observable in log sequences, prompting sequential log analysis. Sequential log analysis is challenged by the sequence diversity due to novel events and event dependencies caused by system updates, missing events, and incorrect preprocessing. We aim to improve the sequential log analysis by improving the representation of the log sequences, which potentially decreases the negative effects of the challenges on the log sequences.

1.2 Main Contributions

This thesis contributes to the general scientific discipline of computer science. Specifically, it contributes to the fields of software (reliability) engineering and artificial intelligence. It proposes methods to aid IT system development and operational activities by advancing intelligent logging code composition, and automatic log-based anomaly detection and

classification, with the end goal to improve the support of system dependability. The three main contributions of the thesis are given as follows:

Logging Code Composition Quality. An important prerequisite in log analysis is access to logs of good quality. To that end, we develop an approach to automatically assess the quality of log instructions from software systems. The development of such an approach is challenged by the heterogeneity of the systems, the unique writing styles of developers, and different programming languages. To assess the logging quality, first, we identify a set of two automatically empirically testable quality properties in a system-agnostic manner. Second, we introduce and formalize the problem of quantification of log instruction quality assessment. Third, by leveraging our observations and the textual nature of the logs, we propose a framework for automatic model-driven log quality assessment as an intelligent tool to aid the writing of log instructions. Finally, we propose an approach to giving feedback for granular quality improvement of the log instructions.

Single Log Line Analysis. As the anomalies in single log lines can be reflected in the semantics of the static text or as abnormal parameter values, we contribute novel methods for the two. First, by analyzing log instructions from public systems we observed and show that the log instructions contain rich anomaly-related information from many different systems. The proposed method utilizes the data from the system of interest (target system data) alongside the extracted anomaly-related information as auxiliary data to learn anomaly-discriminative log representations to improve the generalization performance. As the correct extraction of the parameters from the logs is a vital step for parameter anomaly detection, we contribute with a novel method for robust log parsing. The parser formulates the problem as a masked language modeling task to learn the templates and the variable parts. The extracted events are used to create lists of parameter values where parametric anomaly detection is performed.

Sequential Log Analysis. To address the sequential properties of the logs, we introduce a novel method for improving the sequential representation. By representing the log sequences as sequences of event groups, we found that the uncertainty in the overall log event sequence is reduced. We propose a novel method that extracts event groups from a given event sequence. The learned sequences of event groups are used in anomaly detection and classification. The evaluation results demonstrate that the modified input representation improves practical properties during sequential log analysis.

The methods presented herein are implemented as prototypes and evaluated on benchmark datasets, publicly collected datasets, data from testbeds, and large-scale production data, whenever possible. In addition, the thesis contributes with two datasets 1) a dataset with more than 100 thousand log instructions from public open-source systems; and 2) a collection of online failure incidents from 70 services. Parts of this thesis have been

published in peer-reviewed articles and a patent accepted at the European Patent Office. We list the relevant contributions in the following:

[1] Jasmin **Bogatinovski** and Odej Kao. "Auto-Logging: AI-centred Logging Instrumentation". In: *45th International Conference on Software Engineering, (ICSE '23)*. To appear. 2023.

[2] Jasmin **Bogatinovski** and Sasho Nedelkoski. "Multi-source Anomaly Detection in Distributed IT Systems". In: *Service-Oriented Computing – ICSOC 2020 Workshops*. Cham: Springer International Publishing, 2021, pp. 201–213. DOI: https://doi.org/10.1007/978-3-030-76352-7_22.

[3] Jasmin **Bogatinovski**, Sasho Nedelkoski, Alexander Acker, Jorge Cardoso, and Odej Kao. "QuLog: Data-Driven Approach for Log Instruction Quality Assessment". In: *30th International Conference on Program Comprehension (ICPC '22)*. USA: ACM, 2022. DOI: https://doi.org/10.1145/3524610.3527906.

[4] Jasmin **Bogatinovski**, Sasho Nedelkoski, Jorge Cardoso, and Odej Kao. "Self-Supervised Anomaly Detection from Distributed Traces". In: *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*. 2020, pp. 342–347. DOI: 10.1109/UCC48980.2020.00054.

[5] Jasmin **Bogatinovski**, Sasho Nedelkoski, Gjorgji Madjarov, Jorge Cardoso, and Odej Kao. "Leveraging Log instructions for Log-based Anomaly Detection". In: *2022 IEEE International Conference on Services Computing (SCC)*. 2022, pp. 321–326. DOI: 10.1109/SCC55611.2022.00053.

[6] Jasmin **Bogatinovski**, Sasho Nedelkoski, Li Wu, Jorge Cardoso, and Odej Kao. "Failure Identification from Unstable Log Data using Deep Learning". In: *22nd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. NY: IEEE Press, 2022. DOI: 10.1109/CCGrid54584.2022.00044. eprint: 1646836319158.

[7] Jorge Cardoso, Jasmin **Bogatinovski**, and Sasho Nedelkoski. *Distributed Trace Anomaly Detection with Self-Attention based Deep Learning*. Approved by the European Patent Office, WO2022053163A1. 2022.

[8] Sasho Nedelkoski, Jasmin **Bogatinovski**, Alexander Acker, Jorge Cardoso, and Odej Kao. "Self-Attentive Classification-Based Anomaly Detection in Unstructured Logs". In: *2020 IEEE International Conference on Data Mining (ICDM)*. 2020, pp. 1196–1201. DOI: 10.1109/ICDM50108.2020.00148.

[9] Sasho Nedelkoski, Jasmin **Bogatinovski**, Ajay Kumar Mandapati, Soeren Becker, Jorge Cardoso, and Odej Kao. "Multi-source Distributed System Data for AI-Powered Analytics". In: *Service-Oriented and Cloud Computing*. Cham: Springer International Publishing, 2020, pp. 161–176. DOI: https://doi.org/10.1007/978-3-030-44769-4_13.

[10] Sasho Nedelkoski [2], Jasmin **Bogatinovski**[2], Alexander Acker, Jorge Cardoso, and Odej Kao. "Self-supervised Log Parsing". In: *Machine Learning and Knowledge Discovery in Databases: Applied Data Science Track*. Cham: Springer International Publishing, 2021, pp. 122–138. DOI: https://doi.org/10.1007/978-3-030-67667-4_8.

[11] Harold Ott, Jasmin **Bogatinovski**, Alexander Acker, Sasho Nedelkoski, and Odej Kao. "Robust and Transferable Anomaly Detection in Log Data using Pre-Trained Language Models". In: *2021 IEEE/ACM International Workshop on Cloud Intelligence (CloudIntelligence)*. 2021, pp. 19–24. DOI: 10.1109/CloudIntelligence52565.2021.00013.

[12] Thorsten Wittkopp, Alexander Acker, Sasho Nedelkoski, Jasmin **Bogatinovski**, Dominik Scheinert, Wu Fan, and Odej Kao. "A2Log: Attentive Augmented Log Anomaly Detection". In: *Proceedings of the 55th Annual Hawaii International Conference on System Sciences*. Honolulu, HI: ScholarSpace, University of Hawaii at Mano, Hamilton Library, 2022. DOI: 10.24251/HICSS.2022.234.

Other publications:

[1] Acker Alexander, Wittkopp Thorsten, Nedelkoski Sasho, **Bogatinovski** Jasmin, and Kao Odej. "Superiority of Simplicity: A Lightweight Model for Network Device Workload Prediction". In: *Proceedings of the 2020 Federated Conference on Computer Science and Information Systems (FedCSIS 2020)*. Institute of Electrical and Electronics, 2020, pp. 7–10. DOI: https://doi.org/10.15439/2020F149.

[2] Jasmin **Bogatinovski**, Dragi Kocev, and Aleksandra Rashkovska. "Feature Extraction for Heartbeat Classification in Single-Lead ECG". In: *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. **Best Paper Award**. 2019, pp. 320–325. DOI: 10.23919/MIPRO.2019.8757135.

[3] Jasmin **Bogatinovski**, Yu Qiao, Jorge Cardoso, and Odej Kao. "First CE Matters: On the Importance of Long Term Properties on Memory Failure Prediction". In: *2022 IEEE International Conference on Big Data (Big Data)*. To appear. 2022.

[2] Equal Contribution

[4] Jasmin **Bogatinovski**, Ljupčo Todorovski, Sašo Džeroski, and Dragi Kocev. "Comprehensive comparative study of multi-label classification methods". In: *Expert Systems with Applications* 203 (2022). DOI: https://doi.org/10.1016/j.eswa.2022.117215. eprint: 117215.

[5] Jasmin **Bogatinovski**, Ljupčo Todorovski, Sašo Džeroski, and Dragi Kocev. "Explaining the performance of multilabel classification methods with data set properties". In: *International Journal of Intelligent Systems* 37 (), pp. 6080–6122. DOI: https://doi.org/10.1002/int.22835.

[6] Tome Eftimov, Gašper Petelin, Gjorgjina Cenikj, Ana Kostovska, Gordana Ispirova, Peter Korošec, and Jasmin **Bogatinovski**. "Less is more: Selecting the right benchmarking set of data for time series classification". In: *Expert Systems with Applications* (2022), p. 116871. DOI: https://doi.org/10.1016/j.eswa.2022.116871.

[7] Ana Kostovska[2], Jasmin **Bogatinovski**[2], Sašo Džeroski, Dragi Kocev, and Panče Panov. "A catalogue with semantic annotations makes multilabel datasets FAIR". In: *Nature Scientific Reports* 12 (). DOI: https://doi.org/10.1038/s41598-022-11316-3. eprint: 7267.

[8] Li Wu, Jasmin **Bogatinovski**, Sasho Nedelkoski, Johan Tordsson, and Odej Kao. "Performance Diagnosis in Cloud Microservices Using Deep Learning". In: *Service-Oriented Computing – ICSOC 2020 Workshops*. Cham: Springer International Publishing, 2021, pp. 85–96. DOI: https://doi.org/10.1007/978-3-030-76352-7_13.

[9] Li Wu, Johan Tordsson, Jasmin **Bogatinovski**, Erik Elmroth, and Odej Kao. "MicroDiag: Fine-grained Performance Diagnosis for Microservice Systems". In: *2021 IEEE/ACM International Workshop on Cloud Intelligence (CloudIntelligence)*. 2021, pp. 31–36. DOI: 10.1109/CloudIntelligence52565.2021.00015.

1.3 Thesis Outline

Chapter 2 provides the background on system dependability, observability and artificial intelligence for IT system operations. It focuses on the process of logging instrumentation, intelligent data-driven methods and the tasks of log-based anomaly detection and classification as key concepts required for understanding the contents of the thesis.

Chapter 3 presents a literature review on the automation of log analysis tasks related to the system's development and operation activities. Specifically, it first discusses the logging code composition approaches as part of the logging instrumentation during development. Afterwards, it describes the literature on log-based anomaly detection and classification related to system operation.

Chapter 4 describes the main challenges and assumptions for automation of log-related activities in the support of dependability through both phases of development and operation. It introduces a reference architecture positioning the methods and ideas described in the thesis. Finally, it formalizes the addressed tasks and gives an overview of the proposed methods and ideas concerning the challenges.

Chapter 5 introduces the concept of automatic log instruction quality assessment as a way to directly support the development process. It starts by presenting a preliminary study, where we examine the empirically testable properties of the log instructions. Based on the observations, it introduces a method for assessing their quality. Finally, it presents the experimental results concerning the evaluated quality properties from publicly available data.

Chapter 6 introduces the single log line anomaly detection and classification methods to directly support system operation. The single log line analysis is split into two parts, i.e., semantic and performance log analysis. The semantic log analysis is first introduced. It starts with a study on how to extract anomaly-related log information from the log instruction of public code projects. Based on the observations, it introduces a novel method for semantic log-based anomaly detection. Afterwards, it presents the experimental results and a study of its important practical properties. Next, the performance log analysis is discussed. The method for log parsing is presented. In continuation, a description of how the extracted templates are used to construct learning data that is used for anomaly detection is given. Finally, the experimental results are presented.

Chapter 7 introduces the sequential log analysis method to directly support the system operation. It starts by comparing the difference in the uncertainties of the log event sequences by using the original event sequences and a representation with event groups. Following the observations from the comparison, a novel method for learning the sequential log event groups is introduced. Furthermore, the method for log-sequence anomaly classification is discussed. The chapter concludes with a presentation of the experimental results.

Chapter 8 concludes the thesis, summarizes the findings and identifies directions for future research.

Chapter 2

Background

Contents

This chapter presents the background as a prerequisite for understanding the concepts discussed within the thesis. We first discuss the general concepts related to system dependability. Second, we discuss system observability by focusing on log-based system instrumentation and logging code composition. Third, we describe intelligent data-driven methods as means to support system dependability, particularly emphasising the task of anomaly detection as one of the central problems considered in the thesis.

2.1 System Dependability

Computer systems are expected to fail over time and, therefore, experience reduced dependability [90]. System dependability is defined as the ability of the system to deliver correct service that can justifiably be trusted [5]. Dependability has three aspects. These are (a) *attributes* (constituents to system dependability), (b) *threats* (challengers to system dependability), and (c) *means* (approaches to enable system dependability). Following the definitions introduced in Avižienis et al. [5], we discuss the different aspects of dependability and point out the dependability aspects addressed within the thesis.

As an integrating concept, dependability encompasses several *attributes*, i.e., availability, reliability, safety, integrity, and maintainability [5]. *Availability* is defined as the system's readiness for correctly providing the agreed services. It is quantified as the percentage of the execution time during which the service is delivered correctly. *Reliability* is the capability for the continual support of correct services. In quantifiable terms, it is the likelihood that the system performs without failure for a predefined time. *Safety* expresses the system's capability to deliver services without catastrophic consequences for the user(s) and the environment. *Integrity* refers to the lack of ability for improper or unauthorized system alterations. *Maintainability* is the system's capability to be subjected to modifications and repairs.

The system delivers a *correct service* when it is correctly externalising the system state to implement (agreed) system functions. *Failures, errors* and *faults* are the crucial threats that affect the *correctness*, thereby, directly affecting system dependability during both development and its usage/operation. A **failure** is an event that occurs when the delivered service deviates from the correct service. The deviation may assume different forms that are referred to as *service failure modes* [35, 89]. The total system state that leads to a deficit between the corrected and expected service state is called **error**. The hypothesised or considered cause for the error(s) is called **fault**.

Figure 2.1 illustrates the causal model of a failure developing from fault through errors. This model is commonly referred to as Laprie's fault-error-failure model [90]. Three unit blocks, each corresponding to the three model components, can be identified. The top box descriptions of each box represent the block purpose, while the appropriate detection tools are shown at the bottom.

The causes for the faults are diverse. They can be events from outside (e.g., shortages

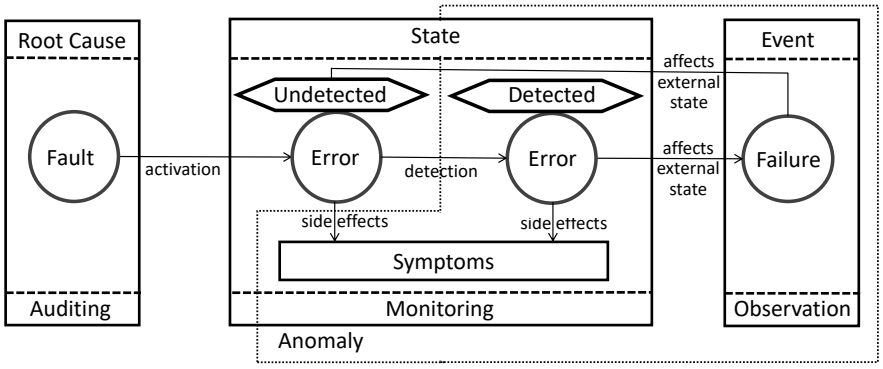


Figure 2.1: Fault-error-failure model as an aspect of dependability (adopted from Acker [2]).

in electricity supply), or changes in system functional requirements (e.g., outdated functionality). Faults can be either active (when faults manifest as an error) or dormant. The detection of dormant faults is subject to processes known as auidition [2]. A dormant fault may persist through the system life-cycle and be undetected if the conditions for its activation are not fulfilled. The active faults appear when certain system conditions are fulfilled. They are manifested as a deficit between the expected and manifested system state, i.e., cause errors. Errors can be detectable or non-detectable/latent. If there exists a signal indicative of an error presence, the error is detectable. One way to detect errors is by monitoring the components that expose the internal system state [89]. The detectable and non-detectable errors can cause a chain of system state inconsistencies manifestable as other errors, or symptoms. They can ultimately externalise to the end-users and cause failures. Usually, the mapping between the faults, errors, and failures is m-to-n. For example, several faults may result in one error, or one fault may result in several errors. Although not part of the general fault-error-failure model, the term **anomaly** is used to refer to the detectable errors and failures as an out-of-normal (anomalous) system behaviour [2]. Figure 2.1 illustrates part of Laprie's model with a dashed line to the term anomaly refers. The analysis of the monitoring data is one way to identify anomalous system behaviour, i.e., enable the detection and classification of anomalies.

To provide dependable service, several *means of dependability* exist including *fault prevention*, *fault tolerance*, *fault removal* and *fault forecasting*. *Fault prevention* aims to improve the development processes (of both hardware and software) by reducing the number of introduced faults. It is achieved by aiding system development in writing code with fewer faults/bugs (e.g., by introducing quality indicators during development [7, 33]) or by eliminating the causes of the faults in the case of hardware production (e.g., via process modifications) [32]. *Fault tolerance* focuses on avoiding the faults and errors externalising as service failures. Therefore, faults and errors are assumed to exist and fault tolerance mechanisms strive to prevent their manifestation in the outside world. Commonly, it is achieved by error/anomaly detection and system recovery activities. *Fault removal* methods aim at addressing the faults either during development (by verification testing, diagnosis and correction) or during system operation by corrective (removing faults that cause one or more errors) or preventive (uncovering faults before their manifestation as errors) means. Finally, *fault forecasting* analyses system behaviour, most commonly by using past information for the faults. It attempts to identify, classify, and rank failure modes or co-occurring events that lead to failures.

The diversity and the polymorphic manifestation of the faults (which cause failures) challenge the system's dependability. As seen from the previous discussion, dependability covers many tasks during system operation and development. From the system operation perspective, the focus of this thesis is the fault tolerance aspect of dependability.

Specifically, we consider the detection and classification of anomalies, in the context of fault tolerance, as their timely detection enables longer time intervals for handling the failures, errors and ultimately their faults [90]. The precondition to detect anomalies resides in the system observability, i.e., the system's capability to externalize the inner system state, for example, by means of monitoring data. From the system development perspective, the thesis focuses on improving the quality of the monitoring data – the logging data, indirectly aiding the tasks stemming from logs, e.g., log-based fault-tolerance-related methods. In the following, we describe concepts related to system observability and monitoring data as mediums to externalise the inner system state and enable the detection and classification of anomalies.

2.2 System Observability

System observability is the system's ability to externalize the system state [157]. The monitoring data enable the analyses of system behaviour. They are one mechanism to achieve observability. Concerning IT systems, there are many used data for modeling, with the two commonly used monitoring data sources being the metrics and log messages (logs) [157]. Metrics are numeric values describing the utilization of the system or its components through time. Typical metric data are the CPU/memory/network utilizations, service latency, and diverse error rates, among others [124]. Logs are textual descriptions recording events during system runtime. Alongside the event descriptions, other information such as event timestamp, task identifier the event is part of, the system component that produced the event, and similar meta information, are also recorded. Considering logs, a special log subcategory is (distributed) traces [25]. Traces represent a series of causally related events in response to requests. Due to their specifics, they are sometimes recognized as separate monitoring data [157]. They are predominantly used in distributed systems to cross-link events across different processes or machines (where a scenario can be executed multiple times). However, suggestions for their applicability in complex applications, where there are nontrivial interactions between components (e.g., network, disk), also exist [124]. In the following text, we discuss the logging process and logging practices in modern IT systems.

2.2.1 Software Logging

Logging is important programming practice in modern software development, as software logs – the end product of logging – are frequently adopted in diverse development and maintenance tasks [92]. Logs record system events at different 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 tasks relying on logs are comprehending system behaviour [92],

troubleshooting [41] (e.g., anomaly detection), and tracking execution status [154]. The importance of logs is recognized by introducing legislation acts, e.g., Sarbanes-Oxley Act [151], which require logging of mandatory system information for privacy protection.

Logs are textual event descriptors generated by log instructions in the source code. Figure 2.2 depicts an example of a logging code that records two events related to receiving information from a client and sending an appropriate response to a certain address. The instructions are written in the programming language Java. As seen from lines 1 and 2, Log4J [49] a popular Java-based library, is used as a logging library. In line 5, an object that handles the logging is created. Lines 8 and 14 show two log instructions that are used to record the events of receiving from the client and sending a response to a certain address. Despite the logging object, there are three other parts of the log instructions, i.e., 1) *static text* describing the event (e.g., `Receive from client`), 2) *variable text* giving dynamic information about the event (e.g., `reg.userName` as client name), and 3) *log level* (e.g., `info`), 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 5000 log instructions. Notably, in the example, a functional code that performs client authentication between the two logging instructions exists. This practice of interwinding logging and functional code is commonly used by developers, making logging a cross-cutting concern [85]. Figure 2.3 shows the generated logs when the instructions are executed. The logs are often stored in files, referred to as log files.

```
1  import org.apache.logging.log4j.LogManager;
2  import org.apache.logging.log4j.Logger;
3  ...
4  class MyServer {
5      Logger logger = Logger.getLogger(MyServer.class);
6      void authentication(Request req, ...) {
7          Configurator.setLevel(logger.getName(), level.DEBUG);
8          logger.info("Receive from client " + req.userName);
9          OKHttpClient.Builder builder = new OKHttpClient.Builder();
10         HttpLoggingInterceptor logInter = new HttpLoggingInterceptor();
11         builder.addInterceptort(logInter);
12         // authentication process
13         reply(response, ...);
14         logger.info("Send response to " + req.IP);
15 }
16
17 private void start() {
18     Server server = new Server();
19 }
```

Figure 2.2: An example of logging code in the method MyServer.java. Adapted from Chen et al. [25].

```
Jun 20, 2022 10:30:01 AM user.MyServer Receive from client Bob
...
Jun 20, 2022 10:30:02 AM user.MyServer Sent response to 192.168.0.1
...
Jun 20, 2022 10:30:03 AM user.MyServer Receive from client Alice
...
Jun 20, 2022 10:30:04 AM user.MyServer Sent response to 192.168.0.2
```

Figure 2.3: An example of logs from the logging code given in Figure 2.2.

The logs allow a more insightful analysis and interpretation than the metric data [119]. For example, a sharp increase in the *network packet loss* (a commonly used metric for network monitoring) only indicates a problem with the network but does not provide a clue why it happens. In comparison, logs give semantically meaningful clues for the anomaly. For example, when a switch generates the log "System is rebooting now.", the operator detects that the switch is failing (potentially anomalous) and obtain a clue that the potential anomaly is caused by the switch rebooting. Furthermore, distributed tracing logging enables the causal tracing of events on a system level. In contrast, metrics can monitor just individual component states, requiring additional techniques to extract information about connected services. In addition, as the number of components increases, choosing the right metrics to track can be cumbersome because of the many possible metrics that can be subject to analysis. While logs have their challenges (e.g., the trading of the verbosity of logging and the excessive logging storage), this thesis focuses on system logs because of their (1) *wide availability* (frequently, they are the only data source for on-field analysis [92]); (2) *rich properties* (e.g., semantics, event co-occurrence, parameters, sequences); (3) *different granularity of event recordings* (up to atomic system events, e.g., control status bits in CPUs); and (4) the *human-understandable clues for the anomalies and their types* that potentially can speed up the processes of anomaly detection, classification and error handling [119].

2.2.2 Software Log Instrumentation

An important aspect of software logging is log instrumentation. Log instrumentation is the process of developing and maintaining logging code. It is composed of a three-stage sequential process concerning the choices of (1) logging approach, (2) logging utility and (3) logging code composition [25]. Given that logging and functional code (i.e., the code that is executing the function) are intertwined, each of the three stages requires careful consideration during both development and operation to provide maximal utilization at minimal performance overhead. Following Chen et al. [25], we discuss the three logging stages in the remaining part of this section.

Logging Approaches

Logging is a cross-cutting concern, meaning that functional code is often intertwined with the logging code snippets [85] (*code snippet* - a set of consecutive instructions). Choosing the most suitable logging approach is the first step when instrumenting the source code. There exist three approaches for logging (1) *conventional logging*, (2) *rule-based logging* and (3) *distributed tracing*. In *conventional* logging, developers insert logging statements in the source code intertwined with the functional code. *Rule-based* logging modularizes the logging by splitting the functional from the logging code. For example, in one specific implementation of a rule-based approach using the Aspect-Oriented Programming (AOP) design pattern, the logging is implemented in a separate source code file. Alongside the actionable logging code, a set of rules on when to invoke logging instructions are also specified within the logging library. When methods in the main function are invoked, the logging instructions from the logging source code are executed according to the specified rules. *Distributed tracing* alongside the information for the event, additionally cross-links the logs across different processes and machines in a response to a certain request. It is commonly used in distributed systems, where one scenario may be executed multiple times, and it is important to know the relation between the different events and the execution scenarios they appear in [25].

All three logging approaches have pros and cons. For example, distributed tracing provides structured logging as opposed to the free-form logging associated with traditional and rule-based approaches. The structured information reduces the post-processing overhead introduced by mixing the parameters and the static text in the instructions. However, distributed tracing and rule-based logging have lower flexibility and higher effort to be applied in various instrumentation scenarios. Distributed tracing is bounded to software components, while the rule-based approach can be applied just for the prespecified rules defined within the supporting logging library. In contrast, the traditional approach is more flexible as it allows inserting instructions at any point in the code. The rule-based approach modularizes the logging and functional code, making them both easier to update. Therefore, it is less bug-prone in comparison to the traditional approach. Ultimately, as the first step in logging instrumentation, the logging approach choice is important and should consider the advantages and disadvantages of the existing possibilities. Once the logging approach is selected, the next step is to select the appropriate logging utility [25].

Logging Utility

To implement the logging, developers usually consider different logging utilities, e.g., SL4J [141], Log4J [49] for Java, spdlog [134] for C++, logging [133] for Python and similar. The logging utilities unify the logging procedure while providing additional

capabilities such as log levels (to regulate verbosity) or enable the synchronization of logging in multi-threaded systems, which can be critical for high data volume processing systems. For various programming languages, there are many different logging utilities. For example, Chen et al. [27] studied more than 11 000 Java systems available from GitHub and identified that there are more than 800 third-party logging utilities used. Also, it is often a case that developers implement their own logging libraries. This is particularly emphasised for safety-critical systems, where system safety is of key importance. For example, in the issue CVE-2021-44228[1] of the National Vulnerability Database, it is reported that a vulnerability in the library Log4J allowed the execution of any Java method through the log instruction from an LDAP server. The violation of the integrity of the logging library in the aforenamed case motivates the need for developers to implement system-specific logging libraries. The choice of *what is the most suitable tool* is a major concern concerning logging utility.

Another important aspect with respect to logging utility is related to *how to configure* the selected tool [184]. The most frequent concerns in that regard are related to the reliability and performance of the logging storage. Other challenges arise with setting the right thresholds on verbosity or the different choices of handlers/appenders (as responsible objects to record logging requests to the destinations, e.g., files, databases and similar).

The two decisions on *what is the most suitable tool* and *how to configure it* are predominantly dependent on the type of software, security and correctness considerations, as well as the expected use of the logged information. The careful consideration of logging utility affects the overall purpose of logging and should be performed with caution [184].

Logging Code Composition

Once the logging approach and logging utilities are chosen and configured, the last step of the logging instrumentation is inserting log instructions into the source code, i.e., composing the logging code. In this regard, there exist three essential properties of the logging code composition: (1) where-to-log, (2) how-to-log, and (3) what-to-log.

Where-to-log, also known as *log placement* [19], refers to the correct placement of the log instruction in the source code. Although logs provide rich information, excessive logging leads to additional overhead concerning storage and communication that can degrade overall system performance [24]. Furthermore, the excessive log information is challenging for the log analyses as not all logs are related to issues [80]. Therefore, the developers need carefully to consider where to place the log instructions.

How-to-log is concerned with developing and maintaining high-quality logging code. While aspects of software development such as code refactoring and release management

[1] https://nvd.nist.gov/vuln/detail/CVE-2021-44228

have well-defined practices, logging does not [52, 137]. One reason for this is the cross-cutting nature of logging and its frequent update with the functional code (e.g., more than 80% of log-related commits are a consequence of functional code updates [24]). The term *anti-pattern in logging code* is used to refer to the recurrent mistakes that may hinder the understanding and maintainability of the logs (e.g., return of nullable or byte-like function calls). In addition, how-to-log is concerned with the log-bug resolution times, frequency of log instructions-related issues, or changes in logging configuration as aspects of logging code evolution.

What-to-log refers to writing log instructions with appropriate log levels, comprehensive static texts and sufficient variable contents. *Log levels* control the verbosity of the output information. If the log level is overvalued, it can lead to excessive logging reporting, while if it is undervalued important information may be missed [80]. Both cases can hurt the subsequent log analyses [100]. Therefore, it is important to set the log level to its appropriate value. The *static text* and *variable parameters* give the human-understandable information within the log instruction used as clues in analysis. Poorly written and outdated event descriptions and inconsistent dynamic variables could affect the effectiveness of log analysis [108, 117]. If the static text of the log message is incomplete or crucial variables are missing (e.g., endpoints), the end-users may be misled by log content which leads to long resolution times for emerging issues. The careful consideration of the three aspects of the *what-to-log* problem is vital for analysis as they provide the atomic units on top of which most of the analyses are executed. Thereby, having good logging code composition is a basic requirement for ensuring logs fulfilling their purpose.

2.3 Artificial Intelligence for IT Operations

Despite the monitoring data, another aspect of automating the dependability means and their subtasks (e.g., logging code quality assessment, anomaly detection, anomaly classification and similar) is the adoption and development of intelligent data-driven methods [130]. The need for intelligent methods is motivated by the cumbersome manual effort human operators invest to analyze the monitoring data [70]. For example, the logs from modern software systems are generated in large volumes (e.g., several TB per day [70]), while the suspicious logs may be hidden among a few logs describing normal events. Therefore, the operators can easily be overwhelmed by the manual analysis of thousands of logs when detecting and classifying anomalies.

AIOps is a term used to describe the combined (or sole) usage of intelligent methods, from the areas of artificial intelligence, data mining, and machine learning on one side and system data on the other side, to support the automation of IT operations and system dependability [38, 130]. Intelligent data-driven methods can glean meaningful

patterns and uncover informative trends in the data. The extracted patterns can guide the monitoring, administrating and troubleshooting of software systems, therefore, helping the Site Reliability Engineering (SRE), Development and Operation (DevOps) and O&M teams to enhance the quality of the IT system offerings.

AIOps encompasses several different tasks, i.e., anomaly detection [154], root cause analysis [116], failure remediation [185], failure prediction and prevention [149], and resource provisioning [130]. The first four tasks are further conceptualized as failure management, as they are concerned with the analyses of system failures/anomalies. They are further conceptualized as *reactive* and *proactive* concerning the phases of methods application, i.e., *after* or *before* the failure occurs [130]. In the following, in accordance with Notaro et al. [130], we discuss the two groups of failure management tasks.[2]

The *reactive* group considers tasks that aim to detect and handle the failures/anomalies *after* their manifestation. This group covers the tasks of anomaly detection, root cause analysis and failure remediation, typically executed in the given order. The task of anomaly detection in system data is concerned with the detection of anomalies [70]. The timely detection allows the operators to spend more time identifying the underlying root cause or to provide insights on different failures prioritization to speed up failure diagnosis. Commonly, the task of anomaly classification can offer a more concrete narrowing down of the failures enabling the usage of past experiences in failure diagnosis. The task of root cause analysis is concerned with localizing the set of faults that caused the system errors and the observed failures. The remediation is concerned with initiating a sequence of repair actions once the root cause has been identified.

The *proactive* group considers tasks that aim to anticipate failures before their manifestation [5]. This can be done during system development (prevention) or during operation by assessing the system state before the failure (prediction). Accordingly, it covers the task of failure prediction and prevention. The task of failure prediction is concerned with the anticipation of failures by assessing the current state of the system (the detectable errors) [149]. The task of failure prevention during system development is concerned with the analysis of code or system states before its deployment, where different aspects of the software files are examined in an attempt to reduce the occurrence of failures (e.g., verification testing). In addition, intelligent methods that aim to improve the monitoring data indirectly address failure prevention. They are categorized into the proactive group.

The remaining text is structured into two parts. We first give an overview of the intelligent methods as they form the basis of all the proposed approaches. In the second part, we discuss the task of anomaly detection as one of the key tasks considered in the thesis.

[2]Note: The task in the literature are introduced concerning failures and anomalies. However, as the errors manifest in the logs before they manifest failures, without loss of generality, we use the words failures and anomalies interchangeably in the description of the tasks.

2.3.1 Intelligent Methods

The diverse AIOps tasks apply data-driven methods developed in different research disciplines, such as data mining, artificial intelligence, big data, and machine learning. To keep the discussion focused, we consider the machine learning perspective on intelligent data-driven methods and present relevant concepts in the following text.

Machine Learning (ML) is a subarea of Artificial Intelligence. It is associated with activities related to developing algorithms/methods that are able to improve their *performance* on a given *task* by gaining *experience* about the *problem instance* being addressed [122]. For such algorithms, it is said that they learn from experience. They are referred to as intelligent algorithms or intelligent methods [56]. The *experience* is provided in terms of data, referred to as *learning* or *training data*. Each element of the learning data represents a training/learning sample denoting an instance of the given problem. The *problem instances* come from different domains, like computer science (e.g., failure detection [12], shell code writing [101]), medicine (e.g., drugs-interaction side effects prediction [189]), biology (e.g., predicting protein folding [82]), mathematics (e.g., aiding mathematicians in theorem proving [39]), physics (e.g., calculating electrical energy properties of molecules [13]) and similar.

The first key concept in machine learning is finding a suitable *representation* of the training data samples. The data can come in very different formats, including vectors of elementary data types (e.g., numeric, ordinal, binary or categorical values), sequences of characters (e.g., text) or numbers (e.g., time series from sensory measurements), images and image sequences (e.g., videos), graphs with homogeneous or heterogeneous nodes and similar. The goal of feature representation is to find and expose the most relevant information from the data for the corresponding task and the addressed problem. In general, there are two main directions on how to extract features (a) *expert-constructed* features and (b) *feature learning*. *Expert-constructed* features involve manual feature construction. This extraction process attempts to capture relevant problem characteristics by using expert knowledge of the domain to the learning method. The *feature learning* approach automatically learns features from the input data. Both approaches have their pros and cons. For example, the features constructed by an expert can lead to smaller and easily-interpretable models if good features are found. However, this activity is cumbersome and requires significant time and cross-discipline collaborative efforts. The feature learning approach is often fruitful when large amounts of data are available [40, 82]. Nevertheless, the learned features often lack interpretability, and their application may be limited in some domains (e.g., financial), where model transparency is important.

Considering the assumptions for the type of the *experience*, three groups of learning exist (1) supervised, (2) unsupervised, and (3) reinforcement learning [66]. *Supervised*

learning assumes that for the training instances, there is available information in a form of a label(s) (output(s) or target(s)) subject to prediction. For example, in the case of logs, a label can be the information for the log level of a given log instruction [9]. *Unsupervised* learning assumes that there is no available information during learning from the exact problem instance being addressed. Identifying groups of similar logs based on their static text is one example of this type of task. *Reinforcement* learning is concerned with the learning of a set of actions on how an agent (e.g., computer hardware or software) interacts with the environment given the input sensory information and/or prior episodes.[3]

Considering the experience type and the problem being addressed, different *tasks* exist. For example, in the case of supervised learning, if the label is of binary type (i.e., has two classes), the task is referred to as binary classification. Assuming that the majority of the samples belong to one of the two classes, the resulting binary classification problem can be seen as a problem of (unbalanced) classification of rear samples (as non-conforming samples). If there are multiple classes a model can learn to classify, the task is called multiclass classification. Similarly, if the label has a numeric value, then the task is called regression. In the case of unsupervised classification there exist multiple tasks, with clustering being among the most popular [66]. The goal of clustering is to group samples based on their similarities into (most often) an unknown number of groups. Notably, based on the assumptions and the modeling choices, a single task can be defined under different learning paradigms. For example, the task of anomaly detection can be defined as a special case of extremely unbalanced binary classification or as a clustering task where distant clusters with a small number of samples can be considered anomalies [102]. Without loss of generality, Equation 2.1 defines the general problem addressed by machine learning methods:

$$\min_{\hat{f} \in \mathcal{H}} \mathcal{J}(f(\mathbf{x}_i), \hat{f}(\mathbf{x}_i; D_t)) \tag{2.1}$$

where $\mathcal{J}$ is a loss function evaluating the similarity between the ground truth function $f(\mathbf{x}_i)$ of the observed problem phenomena, and the function $\hat{f}(\mathbf{x}_i; D_t)$ that is learned on the training data D_t, with $\mathbf{x_i}$ being a learning instance. The learned function $\hat{f} \in \mathcal{H}$ is an element of $\mathcal{H}$, which is the set of hypothesis functions modeled by the intelligent algorithm. Therefore, the goal of machine learning is to find the most suitable model $\hat{f}(\mathbf{x_i})$ from the set of possible functions that best represents the ground truth function phenomena $f(\mathbf{x_i})$ as evaluated by the loss $\mathcal{J}$.

The last important aspect of Machine Learning (ML) definition is evaluating the *performance* of the method. It is assessed by performance functions on a separate test dataset. Evaluating the method on a separate dataset assesses model capabilities when dealing

[3]The reinforcement learning paradigm is not part of the scope of the thesis and is not discussed further.

with new instances of the problem being modeled. The choice of performance functions depends on the properties of the task and the problem itself. A natural evaluation can be the value of the loss function $\mathcal{J}$. While for certain problems (e.g., regression) the optimized loss is a convenient choice, for other problems (e.g., classification), other performance scores may be better suited [54]. For example, in the case of binary classification common choice for performance criteria is the model *accuracy*. If the structure of the problem is such that there is a great imbalance between the two classes, accuracy overestimates the performance [66]. In those cases, more desirable performance functions are criteria that can account for both, the correct and the incorrect model predictions. Effective application and design of machine learning methods should carefully consider the problem formulation, its representation, and careful choice of performance criteria based on the task and type of available experience/data. The end of this chapter discusses the performance criteria used to evaluate the proposed intelligent methods.

2.3.2 Anomaly Detection

Anomaly Detection (AD) as a multidisciplinary task that has been researched within diverse research areas and application domains from natural sciences and engineering disciplines [22]. Examples of different applications include fraud detection in finance, insurance, and telecommunication [3, 15], industrial fault detection [110, 143], event detection in the earth sciences [47], scientific discovery in natural sciences as chemistry [57], genetics [162], physics [21, 72] and others. Formal definitions date back to the 19th century in which the term "*discoherent observations*" is used, although, it is likely that it is being studied informally even earlier [45]. Other terms such as *novelties* and *outliers* are also used to describe non-conforming observations.

Anomaly detection is extensively applied in computer science as well. Examples include the monitoring of IT infrastructure [70], cybersecurirty [44], code defect prediction [105], or as preprocessing step when analyzing monitoring data or mining common execution patterns. The type of data in computer science is commonly in the form of time series, sequences of events, textual descriptions (e.g., source code) or graphs. As the software systems increase in size and heterogeneity, the anomaly detection task likewise is expected to have persistent importance.

The multidisciplinary omnipresence of anomaly detection comes from the universality of the challenge of finding anomalies as "deviating observations from the assumed concept of normality" [146]. There are two important aspects of this definition of anomalies, i.e., the "concept of normality" and "deviating observations". The term "concept of normality" refers to the law of normality represented by the observations. The term "deviating observations" refers to existing observation properties that are considerably different from the ones given with the assumed normality law. Thereby, the main challenge in

anomaly detection is to most suitably represent the normal observations, such that the observations with deviating properties are detectable. Equation 2.2 formalizes the task of anomaly detection as:

$$\mathcal{A} = \{\mathbf{x} | a_1 > p^+(\phi(\mathbf{x})) || p^+(\phi(\mathbf{x})) > a_2, \mathbf{x} \in \mathcal{X}\} \tag{2.2}$$

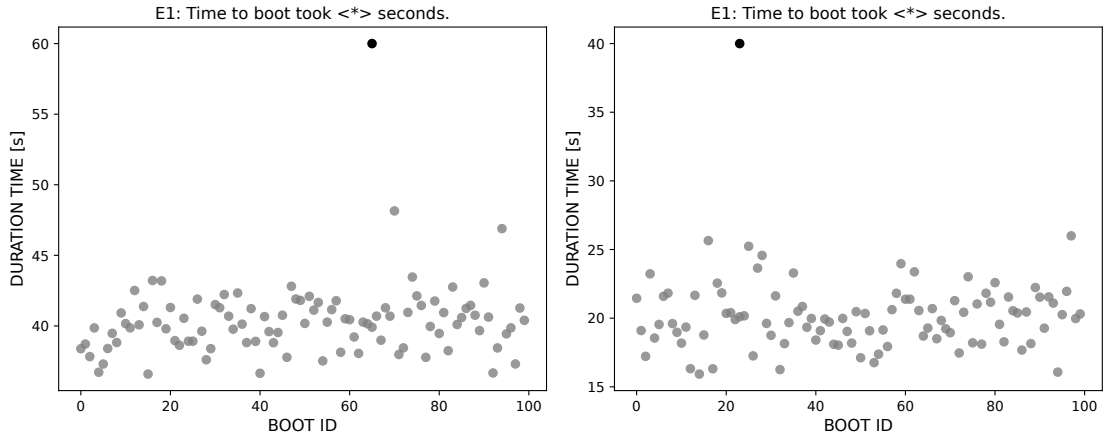
where $\mathcal{A}$ denotes the set of detected anomalies, $p^+(\phi(x)) : \mathbb{R}^d \mapsto \mathbb{R}$ is a function denoting the law of normality (normality function), $\phi(\mathbf{x}) : \mathcal{X} \mapsto \mathbb{R}^d$ denotes the representation function for the observation $\mathbf{x}$ in a d-dimensional vector space, $a_1, a_2 \in \mathbb{R}$ denote thresholds of what is considered a significant deviation ($a_1 < a_2$), and $\mathcal{X}$ is the observation object set of arbitrary data type. The main goals in anomaly detection are, therefore, to find suitable estimates for the normality function $p^+(.)$, discriminative representation of the observations $\phi(\mathbf{x})$ and good estimates for the thresholds a_1, a_2.

Defining the former three goals in a practical application depends on the *anomaly type* and *information type* available for modeling. Conditioned on the structure of the anomaly, there are three *anomaly types* [22, 146]:

1. **Point anomaly** is an anomaly type where an individual observation significantly deviates from the remaining normal observations.
2. **Contextual anomaly** is an anomaly type where the observation is considered an anomaly within the context of an external invariant, such as time, space, a graph of an arbitrary object, or additional attributes.
3. **Group anomaly** is an anomaly type where a set of observations exhibit some form of dependency between themselves and can be considered anomalous just within the observed dependency. The anomalous behaviour does not necessarily occur if the observations are examined individually.

Given the heterogeneous information exposed by computer systems, all three types of anomalies occur within them. Figure 2.4 illustrates an example of three different anomaly types reflected in logs. Figure 2.4a shows a point anomaly of the event booting an OS system. As seen, the majority of the booting time takes 40 seconds on a traditional HDD, however, one of the events took 60 seconds. Therefore, it can be considered anomalous. However, if the OS is stored on an SSD, as depicted on Figure 2.4b, the event of booting the OS for 40 seconds may be considered anomalous, as the booting time on SSD is faster (e.g., normally it takes around 20 seconds). The latter is an example of contextual anomaly, where the context is given by the type of storage used for the OS. Figure 2.4c depicts an example of a group anomaly. Specifically, it shows the flapping of a switch interface [119]. When the switch is flapping, a sequence of log events denoting the repeated sequential change of the interface's state from up to down (and vice versa) in short time intervals is observed. This implies that different anomaly types

occur in computer systems, therefore, the anomaly detection methods should consider the uniqueness of the anomaly types.



(a) Point Anomaly (OS stored on HDD) (b) Contextual Anomaly (OS stored on SSD)

Jun 20, 2022 10:30:02 [SIF pica_sif] Interface te-1/1/11, changed state to down
Jun 20, 2022 10:30:03 [SIF pica_sif] Interface te-1/1/11, changed state to up
Jun 20, 2022 10:30:04 [OSPF] Neighbour(rid:, addr:) on vlan20, changed state from Init to ExStart
Jun 20, 2022 10:30:06 [OSPF] Neighbour(rid:, addr:) on vlan20, changed state from ExStart to Exchange
Jun 20, 2022 10:30:07 [OSPF] Neighbour(rid:, addr:) on vlan20, changed state from Exchange to Loading
Jun 20, 2022 10:30:08 [OSPF] Neighbour(rid:, addr:) on vlan20, changed state from Loading to Full
Jun 20, 2022 10:30:08 [OSPF] Neighbour(rid:, addr:) on vlan20, changed state from Full to Down
Jun 20, 2022 10:30:10 [SIF] Vlan-interface vlan20, changed state to down
Jun 20, 2022 10:30:11 [SIF] Vlan-interface vlan20, changed state to up

(c) Group Anomaly

Figure 2.4: Examples of different anomaly types in computer systems.

Despite the *anomaly type*, the *information type* available during modeling is another aspect to consider. In this regard, there are two important considerations, i.e., (1) *information availability* (or lack thereof) for the observations, and (2) the *data representation*. The availability of information is concerned with the existence of labels for individual samples regarding which observations are anomalous. It determines the approach towards modeling, such as the choices of modeling objectives. The data representation determines the set of representation functions used to describe the observations. In the following text, we discuss these two aspects with a focus on logs having both textual and sequential properties.

Anomalous Information Availability

Concerning the *availability of anomalous information*, there are three groups of anomaly detection methods (a) *unsupervised*, (b) *semi-supervised*, and (c) *supervised*. The *unsupervised* methods assume that there is no available information about the anomalies. Since anomalous observations originate from arbitrary generating phenomena different from normal ones, they have very diverse properties. Given that the anomalies reflected in system logs are infrequent and diverse, it is challenging to obtain labels for them. Thereby, the unsupervised assumption is the closest to a real-world scenario and it is arguably the most frequently considered in anomaly detection literature [22, 136, 146].

In the *semi-supervised* setting the assumption is that despite the given unlabeled observations $\mathcal{X} = \{x_1, x_2 \dots x_n\}$, we are also given a small set of labeled samples $(\hat{x}_1, \hat{y}_1), \dots (\hat{x}_m, \hat{y}_m) \in \mathcal{X} \times \mathcal{Y}$ ($\hat{y} \in \{0, 1\}$, (where 0 stands for normal, and 1 for anomalies), some of which may be anomalies (with $n >> m$). The labels can be obtained by domain experts or from prior manifestations of anomalies. The key idea in semi-supervised anomaly detection is to use the labeled and the large set of unlabeled data when learning a model of normality. The most common approach is to use the *normal* labeled samples and the unlabeled samples when learning a model (e.g., Positive Unlabeled (PU) Learning) [104]. Although it is possible to use anomalous and unlabeled data, due to the lower number of anomalous samples it is usually a less popular approach [148].

The *supervised* approach assumes the availability of representative labels for the normal and anomalous data. Given this, the anomaly detection task can be defined as a binary classification problem and be addressed with standard binary classification methods. One should consider that often in this setting the learning is extremely imbalanced, therefore, suitable method adaptation should be done. Due to the expensiveness of labeling anomalies, the supervised setting is the least common [146].

Data Representation

Good data representation is another important aspect of anomaly detection. One important aspect of good data representation for log-based anomaly detection is related to common procedures for their preprocessing. The static text and the variable parameters can both convey useful information for the analysis [44]. As the log messages are composed of intertwined static texts and parameters, the raw log messages are characterized by greater diversity. Therefore, to extract useful information for the analysis and reduce the uncertainty in the modeling data, it is important to decouple them from one another. *Log parsing* is a general procedure concerned with the decoupling of the static text and log parameters, which is commonly used by different log analysis methods to represent the generated logs [187]. The parsing can be done either by regular expressions or by

learning models to capture the characteristics of the log parsers, referred to as *automatic log parsing*. Constructing good parsers is generally considered a challenging problem as it is expected to be useful logs from different systems that are characterized by diverse events and thus diverse log messages [187].

In general, there are no constraints on the considered representation approaches for log-based anomaly detection. Both, expert-driven and feature-learning representations are used [44, 172]. Despite that expert-driven approaches can provide strong performance, in the case when large volumes of high-dimensional homogeneous data (as for logs), learning of feature representations often leads to superior performance [136, 146]. The key advantage of the feature learning approach is the possibility to enforce certain properties over the learned representations (e.g., compactness, reduced dimensionality). The importance of these approaches is recognized as a separate research direction, named *deep anomaly detection* [136]. In the following, in accordance with Pang et al. [136], we discuss relevant aspects of deep anomaly detection.

Deep anomaly detection refers to the deep learning-based methods for anomaly detection [136]. Deep learning-based methods are a family of machine learning methods that are universal function approximators, modeled as computational graphs [56]. The basic blocks of deep learning methods are neurons. They are singleton computational units that summarize the input by linear combinations (multiplication and addition). The neurons propagate information between themselves through their connections which are called network *parameters*. The parameters are updated during the learning procedure. The neurons are organized in layers, where each layer models a multivariate function. By stacking multiple layers and sequentially propagating the information between them (output from one layer is input to another), a (deep) neural network is created. By introducing diverse special dependencies between the neurons different types of layers are formed. Examples include linear layers, recurrent layers (tailored for modeling sequences), convolutions layers (tailored for learning image features), self-attention layers (suitable for sequences), graph neural layers (for graph modeling) and similar. The output of the neurons is often followed by activation functions. They increase the modeling power of the network by introducing nonlinearities. There are different activation functions, such as ReLU (Rectifier Linear Unit), sigmoid, tanh and similar. All of them have their properties and usages. The parameters of the network are updated using the backpropagation learning algorithm [103]. The backpropagation algorithm optimizes the design-chosen learning objective to best approximate the data.

The state of the neurons of the last layer of the network is called the neural network's output. By careful construction of the learning objective, the parameters of the network can be learned such that the output preserves a desired property. The neural network output can serve as a numerical representation of the presented input. In addition, the

output of any other layer can be considered as an input representation (e.g., a bottleneck of an autoencoder – a type of neural network architecture [56]). The flexible design choices enable the learning of discriminative input representations. The availability of a large set of training data enables the learning of rich properties of the input. Formally, for a given observation dataset $D_t = \{x_1, \ldots, x_n\}$, with $x_i \in \mathcal{X}$ the goal of the deep learning methods, as most frequently considered here, is to find the representation function $\phi(\mathbf{x}; \theta) : \mathcal{X}_{|D_t} \mapsto \mathbb{R}^d$, where $\mathcal{X}_{D_t}$ denotes that the representation is learned on the given observation data D_t, and θ are the parameters of the neural network. In the context of anomaly detection, the learned representation should characterize the assumed normality concept in the data well. Whenever an anomalous observation is presented to the network, ideally, it will represent the input significantly different from the normal ones. The learned representations are given as input to the normality function p^+ to calculate the normality score. In some cases, it is possible to construct an optimization function that directly calculates the normality function p^+ for the input sample (where the representation is calculated implicitly) in an end-to-end manner [136].

When learning the representations, the objective function may or may not involve a term that enforces learning certain representation properties. If the objective function does not include a term for an explicit constraint, the deep learning method acts as a dimensionality reduction technique. It means that, given the high-dimensional complex input, it extracts low-dimensional feature representations. One advantage of this approach is that for many domains, there are already pretrained deep learning models (e.g., BERT [40] for textual data) that can be used to extract good representations fast. However, for the normality function, it may be more challenging to assign smaller values for the anomalies because the learned representations do not account for the exact specifics of the normal input data D_t [136]. If the learning objective includes explicit constraints over the learning data, the learned representations are better suited for the given observational data as they capture the underlying data regularities. To achieve the latter, several different approaches are being considered including data reconstruction, generative modeling, predictability modeling, and self-supervised classification.

A final consideration for the data representation, with regard to the specifics of logs, is the choice of the structural dependency between the parameters θ within the network. Given the observed similarities between logs and textual data [68], the network should enable preserving and exploiting the log properties. Some of these properties are word vocabularies, log semantics, the position of a textual event within a sequence, and different representations for different contexts, among others. Therefore, an important aspect of this work is the effective construction of deep learning architectures and their learning objectives that most effectively exploit the aforenamed properties of the log data. In the next section, we describe one architecture the proposed methods in this thesis rely on.

Transformer's Encoder Architecture

Studies show that system logs experience high similarity with general language [68, 70]. A natural choice for modeling the textual and sequential data is therefore architectures from the Natural Language Processing (NLP) domain. Of particular interest in contemporary literature is the Transformer architecture [166]. Since its introduction, different modifications using different parts of the architecture are being proposed to advance the state-of-the-art research in NLP [17, 40]. Although different modifications are being proposed for different tasks, we describe and use its original implementation. Particularly, we focus on the encoder part of this architecture as a core building block of the proposed methods.

Figure 2.5 depicts the encoder architecture. The encoder is composed of two main elements: the self-attention layer and the feedforward layer. The encoder is fed with tokenized input. Depending on the input type (e.g., sequence or string), the token can refer either to a word or an event. The encoder first applies two operations on the input token vectors: token vectorization and positional encoding. The subsequent encoder structure takes the result of these operations as input. The output from the encoder usually proceeds toward the next step (which can be a similar architecture to the encoder, or as simple as a linear layer with softmax activation). In the following, we provide a detailed explanation of each model element.

Since all subsequent elements of the model expect numerical inputs, initially the tokens are transformed into randomly initialized numerical vectors $\mathbf{x} \in \mathbb{R}^d$. These vectors are referred to as token embeddings and are part of the training process, which means they are adjusted during training to represent the meaning of tokens depending on their context. These numerical token embeddings are passed to the positional encoding block. In contrast to, e.g., recurrent architectures, self-attention-based models do not contain any notion of input order. Therefore, the positional information needs to be explicitly encoded and added with the input vectors to account for the token positions. This block calculates a vector $\mathbf{p} \in \mathbb{R}^d$ representing the relative position of a token. The positional encoding is achieved by adding *sine* and *cosine* functions.

$$p_{2k} = sin\left(\frac{j}{10000^{\frac{2k}{v}}}\right), \quad p_{2k+1} = cos\left(\frac{j}{10000^{\frac{2k+1}{v}}}\right). \tag{2.3}$$

Here, $k = 0, 1, \ldots, d-1$ is the index of each element in $\mathbf{p}$ and $j = 1, 2, \ldots, M$ is the positional index of each token. Within the equations, the parameter k describes an exponential relationship between each value of vector $\mathbf{p}$. Additionally, sine and cosine functions are interchangeably applied. Both allow better discrimination of the respective positions by a specific vector $\mathbf{p}$. Furthermore, both functions have an approximately

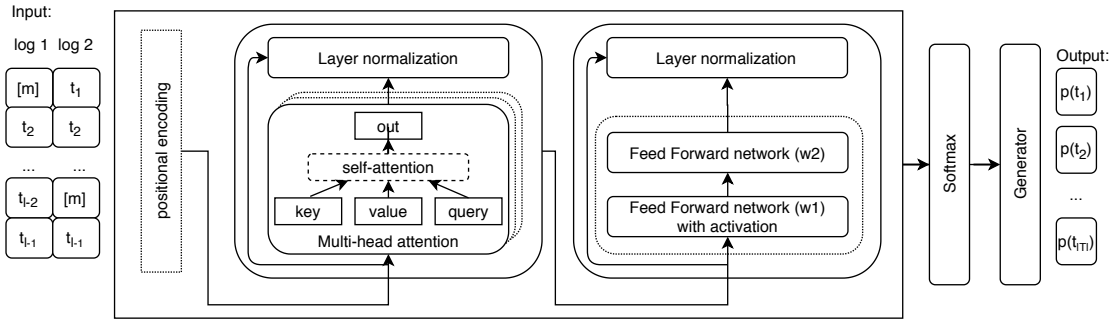


Figure 2.5: Model architecture of the Transformer's encoder.

linear dependence on the position parameter j, which is considered to make it easy for the model to attend to the respective positions. Finally, both vectors can be combined as $\mathbf{x}' = \mathbf{x} + \mathbf{p}$.

The encoder block starts with a multi-head attention element, where a softmax distribution over the token embeddings is calculated. Intuitively, the softmax calculates the significance of each embedding vector for the prediction of the target within its context. All token embedding vectors are summarized as rows of a matrix X' and are transformed as in Equation 2.4.

$$X_l'' = softmax\left(\frac{Q_l \times K_l^T}{\sqrt{w}}\right) \times V_l, \text{ for } l = 1, 2, \ldots, L, \tag{2.4}$$

where L denotes the number of attention heads, d is a token embedding size, $w = \frac{d}{L}$ is the size of a subtoken (part of the token that represents a key, value or query), and $d \bmod L = 0$. The parameters Q, K and V are matrices, that correspond to the query, key, and value elements in Figure 2.5. They are obtained by applying matrix multiplications between the input X' and respective learnable weight matrices W_l^Q, W_l^K, W_l^V:

$$Q_l = X' \times W_l^Q,\ K_l = X' \times W_l^K,\ V_l = X' \times W_l^V, \tag{2.5}$$

where $W_l^Q,\ W_l^K,\ W_l^V \in \mathbb{R}^{M \times w}$. The division by $\sqrt{w}$ stabilizes the gradients during training. After that, the softmax function is applied and the result is used to scale each token embedding vector V_l. The scaled matrices X_l'' are concatenated to a single matrix X'' of size $M \times d$. As depicted in Figure 2.5 there is a residual connection between the input token matrix X' and its respective attention transformation X'', followed by a normalization layer $norm$. These are used for improving the performance of the model by tackling different potential problems encountered during the learning such as small gradients and the covariate shift phenomena. Based on this, the original input is updated by the attention-transformed equivalent as $X' = norm(X' + X'')$.

The last element of the encoder consists of two feed-forward linear layers with a ReLU activation in between. It is applied individually on each row of X'. Thereby, identical weights for every row are used, which can be described as a convolution over each attention-transformed matrix row with kernel size one. This step serves as additional information enrichment for the embeddings. Again, a residual connection followed by a normalization layer between the input matrix and the output of both layers is used. This model element preserves the dimensionality X'. The output from the encoder is used as input to other layers depending on the task being addressed. In each of the proposed methods, different modifications of the encoder architecture and the learning procedures are considered. These specifics are appropriately discussed for each proposed method.

Performance Evaluation Criteria

The final important consideration related to the intelligent methods is the correct choice of evaluation performance criteria. To evaluate different aspects of the proposed methods, we consider diverse performance evaluation criteria that evaluate different aspects of the methods. In the following, we discuss their definitions and usages [54].

After a certain model generates the predictions, there exist two sets of labels, i.e, true labels and predicted labels. In the case of binary classification and anomaly detection, this results in a contingency matrix with four different category groups, i.e., *true positives (tp)*, *true negatives (tn)*, *false positives (fp)* and *false negatives (fn)*. In the case of multi-class classification, there is a contingency table for each class, prompting the need for aggregation. It is also possible to create a single contingency matrix with a size equal to the number of classes, where each column corresponds to the prediction made for one class, and each row corresponds to the true class predictions. Depending on the type of aggregation there are two approaches, *micro* and *macro*. The *macro* approach calculates the single-label measure for each of the classes and averages the result over them. The *micro* approach considers predictions from all instances together (aggregating the values for all the contingency tables), and then it calculates the criteria across all of the classes.

The single-label criteria considered in this thesis are accuracy, specificity, precision, recall and F1. Their definition is given further on. The index *st* emphasizes that they are calculated for two classes (positive and negative).

Accuracy (single target) is the fraction of correctly predicted labels. 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 or the anomaly ratios), accuracy can be misleading [54].

$$accuracy_{st} = \frac{tp + tn}{tp + tn + fp + fn} \tag{2.6}$$

For anomaly detection, due to class imbalance, of interest are metrics that consider the correct and the incorrect predictions. To that end, we use precision, recall, and F_1 score. We give their definitions in the following.

Precision (single target) evaluates the fraction of correct predictions out of all class predictions.

$$precision_{st} = \frac{tp}{tp + fp} = \frac{\#DetectedAnomalies}{\#ReportedAnomalies} \tag{2.7}$$

Recall (single target) evaluates the correct predictions out of all true observations.

$$recall_{st} = \frac{tp}{tp + fn} = \frac{\#DetectedAnomalies}{AllAnomalies} \tag{2.8}$$

F_1 (single target) is the harmonic mean of the precision and recall evaluating the trade-off between correct class predictions and the miss-classifications [54]. For anomaly detection in logs, on one side, it is important not to miss anomalies (missing an anomaly can lead to severe outages). On the other side, reporting many false positives overwhelms the operators, leading to alarm fatigue [92], making the method's practical usability questionable. Therefore, F_1 is used as the primary evaluation criterion for anomaly detection.

$$F_{1_{st}} = \frac{2 \times precision_{st} \times recall_{st}}{precision_{st} + recall_{st}} \tag{2.9}$$

Specificity is the measure of correct predictions of the negative class.

$$specificity_{st} = \frac{tn}{fp + tn} \tag{2.10}$$

If B denotes one of the previously mentioned single label criteria (without accuracy), and $|L|$ denotes the number of classes, then macro and micro criteria are defined as follows:

$$B_{macro} = \frac{1}{|L|} \sum_{i=1}^{|L|} B(tp_i, tn_i, fp_i, fn_i) \tag{2.11}$$

$$B_{micro} = B(\sum_{i=1}^{|L|} tp_i, \sum_{i=1}^{|L|} tn_i, \sum_{i=1}^{|L|} fp_i, \sum_{i=1}^{|L|} fn_i) \tag{2.12}$$

The aforenamed criteria take values within the 0-1 range, and a higher value indicates better performance.

Additionally, the calculation of the *tp*, *fn*, *fp*, *tn* provides an opportunity to plot the ROC (receiver operating characteristic) curve. This curve is obtained such that *tp*, *fn*, *fp*, *tn* are re-calculated for varying the threshold used to obtain the prediction. ROC is obtained with plotting of the *FPR* (false positive rate) on the x-axis and *TPR* (true positive rate,

sensitivity, recall) on the y-axis. Integral under the curve is another measure used to access the performance, the AUC score. We additionally considered the AUC [61] score. AUC is the area under the ROC (receiver operating characteristic) curve. 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 in the case of binary classification. Notably, we do not use AUROC for evaluation of anomaly detection as they evaluate its goodness, without considering the threshold decisions. Since the thresholds are an important step for practical usability we consider the joint evaluation.

As the last performance criteria to evaluate some aspects of the proposed methods, we used the $error@k$ score. It measures the number of incorrect predictions when k-degrees of freedom are considered [81]. The smaller values indicate better performance.

To evaluate the log parsing we used the *parsing accuracy* and *edit distance*. Their definitions are given in the following.

Parsing Accuracy (PA) is a commonly used performance criterion to evaluate parsing. It is defined as the ratio of correctly parsed log messages over the total number of log messages. After parsing, each log message is assigned to a log template. A log message is considered correctly parsed if its log template corresponds to the same group of log messages as the ground truth does. For example, if a log sequence $[e_1, e_2, e_2]$ is parsed to $[e_1, e_4, e_5]$, we get $PA = \frac{1}{3}$ since the second and third messages are not grouped together. The larger values indicate better performance [187].

Edit distance. The PA metric is considered the standard for the evaluation of log parsing methods, but it has limitations when it comes to evaluating the template extraction in terms of string comparison. Consider a particular group of logs produced from a single *print("VM created successfully")* statement that is parsed with the word *VM <*>*. As long as this is consistent over every occurrence of the templates from this group throughout the dataset, PA would still yield a perfect score for this template parsing result, regardless of the obvious error. To address this limitation, edit distance is used [128]. This is a way of quantifying how dissimilar the two log templates (the produced and the ground truth) are to one another by counting the minimum number of operations required to transform one template into the other. We used as operations insert, delete and replace, with costs 0.5, 0.5 and 1. The lower values indicate better performance.

Chapter 3

Related Work

Contents

This chapter discusses the related work. We divide the discussion into two parts. In the first part, we elaborate on the related works on the current state of logging instrumentation, the quality properties of the code composition of log instructions and the different logging practices. In the second part, we discuss the related work on log analysis tasks that enable the automation of anomaly detection and anomaly classification. Specifically, we focus on the tasks of log parsing (as a preprocessing step), and log-based anomaly detection and classification.

3.1 Logging Code Composition Quality

An important aspect concerning the logging code composition is the quality of the logging instructions. The lack of sufficient information can hurt the comprehensibility of the generated logs, directly affecting their usability in all log analytics tasks. From a system development perspective there are four important considerations about placing log instructions within the source code, i.e., (1) correct *log level assignment*, (2) *comprehensive*

content of the static text and parameters, (3) *correct log instruction placement*, (4) correctness of the supporting logging code (e.g., log level guard checking) [64]. The correct log level assignment and the comprehensive content of the static text and parameters are also known as *what-to-log*, while correct location placement is known as *where-to-log*, the correct log level checker is part of *how-to-log*. In the following, we discuss the related work on *what-to-log* as the main contribution relates to this area. In addition, we complement the discussion by describing different logging practices and demonstrating the evolution of the logging code. The latter studies are further referred to as *how-to-log* and we describe the contribution and the relation of the work with it. For completeness on the logging code practices, we also give an overview of the approaches for *where-to-log*.

3.1.1 What-to-Log

The methods on *what-to-log* are separated based on the part of the log instructions they address, i.e., *log level*, *static text* or *variable parameters*. Concerning the *log level*, there exist several approaches. Li et al. [94] propose to recommend the most appropriate log level treating the problem as ordinal regression, where the order of log levels (e.g., debug, info, warning, error, critical) is considered in the model training. Different features from the file (e.g., existing logging instructions in the file), method changes (e.g., newly added logging code) and historical information are collected and used to train a linear ordinal model for log level recommendations. Similarly, DeepLV [100] considers the problem of log level recommendation as a problem of ordinal regression. DeepLV extracts syntactical features using an abstract syntax tree (AST) as a structural representation of the nearby logging code. It also extracts semantic features from the static text of the log instructions represented in a numerical vector format. LSTM [74] model (a type of deep learning architecture) is used to learn dependencies between the input features (both syntactical and semantical), and the log level. Kim et al. [86] consider the problem of log level prediction as a multi-class classification. They use semantic features of the static text (obtained by word2vec [121] textual representation) and syntactic features from the nearby code structure (e.g., the number of log levels in the file, the code block type, i.e., for/if/while blocks) as input representation. Different from the previous works, the authors consider the problem as a problem of multiclass classification (instead of ordinal regression) and train multiclass models (i.e., RandomForest [16] and k-Nearest Neighbour (kNN) [155]) accordingly. Anu et al. [59] propose VerbosityLevelDirector as a method for log level assessment. It uses different numerical features describing the code snippets (e.g., triggered methods, logging content, exception type, code comments) to train a multiclass classification model for log level prediction (e.g., Support Vector Machines [34] and RF). The method was evaluated on in-house production software systems. The results show that the method detected eight incorrect logging levels unknown to developers. Yuan et al. [177] observe that if the logging code within similar code snippets is inconsistent

in terms of log levels then at least some of the levels are incorrect. Based on this, they propose LevelChecker which first identifies all the code clones in the source code. Then it performs a pairwise code clone comparison to find if they contain logging code and if the log levels are consistent.

The problems of *what-to-log* from the perspectives of *static text* and *variable parameters* have a generative and relevance ranking nature, i.e., require generating texts and ranking variable parameters based on their relevance. Concerning the *static text*, He et al. [68] propose using an Natural Language Processing (NLP) approach to generate static text for new log instructions. They observe that the static texts in logging code are *endemic*, i.e., the static texts in the same file or the same contexts tend to use similar static texts to describe system behaviour. Specifically, they represent the source code as n-grams (groups of tokens). A target code snippet (Snippet A) is matched against a corpus of code snippets with log instructions, and the most similar code snippet (snippet B) is extracted. The matching is performed based on the Levenshtein distance. The static text of snippet B is then used as a candidate static text for snippet A. Liu et al. [107] assume that a similar code context should share a similar log description. Based on this observation they consider the problem of static text generation as a retrieval Q&A task, where the question is the code snippet, and the answer is the log instruction. The relationships between the code and the text are learned by neural networks (pretrained on a large corpus of data). As a retrieval task, the static text is generated based on the most similar existing static text in the knowledge base. Ding et al. [42] propose to consider the problem of static text generation as a translation task, where the input is a code context, and the target output is the static text. They utilize a machine-learning approach for the translation task. While the prior two approaches cannot generate new static text, the flexibility provided by learning many diverse code contexts allows the third method to automatically generate new static text. The usefulness of the approach is performed by an additional user study of 42 participants that find the quality of the static text useful as reported by the authors.

Another aspect of *what-to-log* is finding the most *relevant variables* to log. Logging excessive information can cause confusion when examining the logs or hurt the application performance due to logging overhead. To address the issue, several works have been proposed. Rabkin et al. [144] propose a visualization-based solution to log important variables. First, they extract all the variables from the logs and link them based on the same identifiers (e.g., endpoints), extracting graphs. Since the identifiers can be missing, inconsistent or ambiguous, the graph is subjected to human inspection to identify the missing information (e.g., missing edges in the graph may indicate a lack of important variable information). LogEnhancer [178] augments the log instructions by analysing control and data flow paths in the source code to identify missing information. Liu et

al. [109] try to predict if a variable in a given code snippet should be logged or not based on historic information from the same or code from other software projects. They propose to use a binary classifier that recommends (ranks) the variables that should be logged by treating the code as a sequence of code snippets. One problem when recommending the variables to log is the existence of new words that describe the variables, i.e., out-of-vocabulary words. To address this issue, the authors use word vector representations from general language for the individual words within the instructions. In the case of compound words (e.g., errorMessage), the text is represented as an average of the individual tokens. The evaluation results show that this method achieves high performance in suggesting the correct logging variables for nine popular open-source systems.

3.1.2 Where-to-Log

Several approaches to the log placement problem exist. They can be split into two groups 1) *static analysis*; and 2) *runtime placement* tools. The static analysis tools do not require running of the system when instrumenting, while the runtime placement tools involve system running to perform the instrumentation. We first discuss the category of static analysis tools. SmartLog [78] gives recommendations for placing log instructions as a static code analysis tool. Different code snippets (sets of code instructions) are analyzed and characterized to generate log intention models. The log intention models represent the logging decisions, i.e., if the code snippet has log instructions or not. By training intelligent models from the constructed dataset, logging intention models can be learnt. The new code snippet is then given as input to decide if they need to be logged or not. Errlog [176] is another approach that is concerned with the placement of log instructions with log level error. By first studying 250 bug reports, the authors identify exception patterns that require additional logging (e.g., function return errors, exception signals and unexpected cases). Errlog is then proposed as a static code-checking tool to scan the code for these types of error blocks and to suggest log locations. Zhu et al. propose LogAdvisor [186], which constructs a set of features describing relevant code properties that are used as input to machine learning techniques to make the log decisions. The extracted features are grouped into three groups, i.e., structural features (e.g., error type, method name), textual features (e.g., using code as flat text) and syntactic features (e.g., binary value if a method throws an exception or not). The authors conducted a user study and found that 68% out of 37 participants consider LogAdvisor useful. Li et al. [99] propose using deep learning methods to learn block locations (e.g., if/for branching blocks) by representing the source code as a sequence of such blocks. They aim to predict if the next block requires a log statement or not. Cândido et al. [19] combine code metrics on a method level (e.g., depth of a method, coupling between objects) as features and machine learning techniques to learn when a method requires a log statement. In another work, Li et al. [93] use code snippets to automatically calculate

the topic of a method in the source code. By training a machine learning model with the calculated topic features and 14 other method-based features (e.g., number of *for* loops in a method, line of codes, and similar), they obtain high values (more than 80%) on correct log placement suggestions for diverse systems. Similarly, Fu et al. [52], focus on predicting which `catch` and `return-value-check` code snippets need logging. First unlogged code snippets are collected and labeled. Second, contextual keywords, such as the residing function name are extracted as features. Finally, a decision tree model is learnt to predict if a code snippet needs to be logged.

Despite the static analysis tool, several methods that involve source code execution, before deciding the log instruction locations are proposed [175, 181]. Zhao et al. [181] propose Log20 as a tool to automatically suggest where-to-log. It attempts to find the points in the execution paths that resolve the largest uncertainty when diagnosing a problem. To achieve full logging placement, Log20 requires several workloads that invoke all the execution paths within the system. Log4Pref [175] is another approach for suggesting logging points. It starts by building performance models by running performance tests. The performance influencing points in the source code are then identified, and their points of method entrance and exit are instrumented with a logging code. The performance tests are re-executed, and the newly logging code of the methods is filtered based on predefined criteria (e.g., methods with constant execution time). Due to the diverse syntax used by the programming languages, the proposed methods are specific for each programming language (predominantly Java is used in the studies).

3.1.3 How-to-Log

Despite the initial writing of log instructions, another important aspect is the overall life cycle changes of the logs during system usage. In the following, we review the related work on "how-to-log". Yuan et al. [177] present one of the earliest comprehensive works on how developers perform and maintain logging. They find that logging is pervasive (omnipresent) in the source code. Therefore, it is actively maintained by developers. However, its updates are commonly performed as after-thoughts, implying that developers do not consider it of primary importance. Chen et al. [23, 26] in two consecutive studies examine the logging evolution by studying issues from 21 Java projects. They consider both the log-instruction specific issue [23] and feature plus log-instruction issues [26] to cover the different aspects of log evolution. Based on their observation, they propose a six-level taxonomy of log-related updates covering the log evolution aspects. Another study of log-related issues [65] from two software systems identifies four important observations concerning log instruction updates given in the following. Firstly, the log-related issues are identified as relatively long (median of around 300 days), but their resolution time is fast (e.g., the median time is five days). Next, the root causes of the

log-related issues are related to inappropriate log levels, runtime issues, library changes or an overwhelming number of log lines. They also find that the files most related to log issues have undergone significantly more changes than other files. Finally, the log-related updates are most often (in more than 70% cases) performed by developers that are not original authors of the instructions. This indicates that the developers may not have a sufficient understanding of the purpose of logging a particular event. Shang et al. [153] makes a similar observation. Chen et al. [24] identifies four anti-patterns in logging code, i.e., checking for double method invocation (e.g., the method is invoked once in a function and again in the log instruction), explicit casting, nullable object return and malformed output. Based on their observations, they propose LCAnalyizer as a tool that identifies these anti-patterns and reports them to developers. It is claimed that 72% of the suggestions are confirmed and fixed by developers on unseen projects. Shang et al. [152] study the evolution of the communicated information from the log instructions. They find that two types of information are communicated with logs, i.e., long-lived (information about domain-level activities, e.g., opening a bank account) and short-lived information (information about implementation-level details, e.g., database connection establishing).

Despite the studies on open-source projects, there are also studies on industrial production systems. We discuss them in the following text. Fu et al. [52] study how developers in Microsoft use logging. By examining two online systems, they find that logging in commercial systems is pervasive (around 1% of all the instructions are logging instructions). They identify five key points when developers use logging, i.e., during assertion checks, return-value checks, exception logging, logic branch and additional non-classified logging types (referred to as observing-point logging). Additional analysis of individual cases further aims to discriminate causes for the developer's decisions to log. Pecchia et al. [137] study more than two million log instructions from the software systems of an industrial company for critical software. They find similarities in the logging process among different software systems. For example, common developers' objectives with logging are to dump the program state, trace the program execution, and report events. Li et al. [96] observe that log code snippets that share similar contexts frequently undergo similar code modifications. Based on this, they propose LogTracker as a tool to automatically learn log revision rules based on the semantics of the context. The rules are extracted by hierarchical agglomerative clustering. The rules are used to suggest modification of the log code snippets. Kabina et al. [83] identify that 20%-45% of the logging statements in four studied systems are changed. They also report that the median time between adding a log instruction and its first update is between 1 and 17 days conditioned on the system. This gives insights into the log-related bug resolution issues. Salfner et al. [150] propose to measure the quality of log files as a post-product of logging. Specifically, they propose a set of several metrics that evaluate the frequency of logs (the rearer are more informative), logline information, logfile quality and the logfile information entropy. To

calculate the metrics domain knowledge for the required event information is needed. In addition, they specify five types of information a comprehensible log file should have. Those are (1) general information (common for all log files, e.g., timestamp), (2) structural information (that identifies entities in the structure of the software, e.g., cluster node reference), (3) runtime environment (information that is dependant on the system state when the event is logged, e.g., cluster node), (4) resource information (resource the event is associate with, e.g., resource usage) and (5) event characterization (describing the event itself, e.g., state information).

The work presented within the thesis draws inspiration from the area on *how-to-log* on existing issues related to writing quality log instructions. We examined diverse issues discussed in various related works [1, 23, 24, 26, 64, 100, 177], among other public log-related Jira issues, to better understand the properties of the log instructions in modern systems and reason for their quality. We observed that it may be possible to combine the similarities of the log instructions among different programming languages with the general language [68], and the general expressions for the logging intent to assess diverse properties of the log instructions. The work in this thesis further shares close relations with the methods from *what-to-log*, specifically, to the works on log level, and static texts [59, 95, 100]. The major differences and similarities from the related works are further discussed in the corresponding chapter.

3.2 Log Analysis

Log analysis is the process of collecting and analyzing log data to visualize, comprehend, identify, diagnose, classify, detect, predict and mine diverse systems events to aid developers and operators in developing, operating and maintaining IT systems [70]. Figure 3.1 illustrates a typical approach in log analysis [70]. It is composed of three steps: *log collection*, *log parsing* and *log mining*. Once the system is instrumented with logs and it is running, the logs are generated. Depending on the application, different tools are used to retrieve, compress, and store the generated logs. Since the logs are unstructured (e.g., due to intertwined parameters and static text, or different system information exposed by different logging utilities), they are processed by log parsing. Log parsing is a log preprocessing technique that separates the static text and variable parameters from the unstructured logs [187]. Once the logs are parsed they are used to address different tasks such as anomaly detection, classification, and failure diagnosis among others.

The effective addressing of the collection, compression and storage involves reducing memory occupancy while preventing information loss, among the most significant challenges [70]. For example, many logs (e.g., if written in a `for` block) are redundant, and storing all of them may lead to overhead and degrades system performance. On

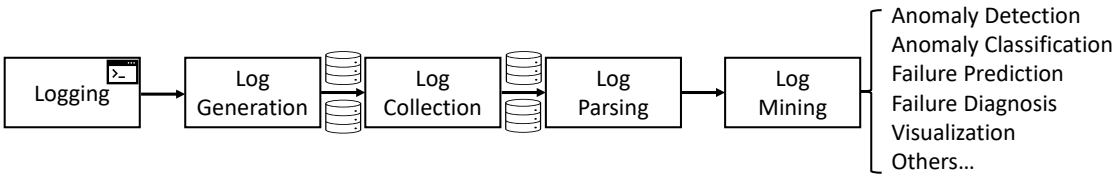


Figure 3.1: General workflow of log analysis (adopted from He et al. [70]).

the other side, some legislation acts require up to two years of log storage for auditing purposes [151]. Therefore, a natural trade-off between the log generation and storing the relevant information from the logs emerges. Addressing each of these questions is beyond the scope of the thesis. In the following text, we discuss the related works on log parsing and log mining parts, with a focus on anomaly detection.

3.2.1 Log Parsing

The generated raw system log message texts are unstructured, i.e., the static text and the parameters are intertwined. Since both the static text and the parameters can convey useful information they need to be decoupled [44]. For example, the parameter anomaly detection is executed on the variable parts of the logs. Therefore, their correct disentanglement from the static text is an important step to detect them [44]. Log parsing is a log preprocessing procedure that decouples the log templates from log parameters/variables (the variable part in a log), directly extracting input in a usable modeling format. Furthermore, the parsed texts are characterized by lower variability than their intertwined complements. A common way to log parsing is to extract the templates by regular expressions or `grock` patterns. However, a great number of logs requires maintaining a large database of regular expressions. In addition, the frequent updates of the log instructions impose constant updating of the regular expressions making them hard to maintain. Therefore, automatic log parsing is preferred [187]. Due to the importance of log parsing, and its relevance to our contributions, we give an overview of the related log parsing methods in the following text.

There exist a plethora of log parsing techniques. They can be categorized in various aspects, based on the *choice of modeling approach*, the *operation mode*, and if they use *additional preprocessing* [187]. Considering the *operational* mode, the parsers can be *offline* (require all the data in advance to execute) or *online* (can parse the logs as they arrive). *Preprocessing* refers to the need to additionally preprocess the logs, before executing the parsing. With respect to the *choice of modeling approach*, there are four identifiable groups, i.e., (1) *frequent-pattern mining*, (2) *clustering*, (3) *heuristics*, and (4) *evolutionary*. In the following, we discuss the related log parsing methods based on the *choice of modeling approach*.

The *frequent pattern mining* log parsing group assumes that a log message type (e.g., the static text of the log instruction) is a frequent set of tokens that appear throughout the logs. Representative parsers for this group are SLCT [164], LFA [125], and LogCluster [126]. The general procedure these methods follow is to iterate several times over the logs. During the first iteration, usually, a data summary is built. Afterwards, groups of similar logs are built using the summary information. Finally, groups with certain properties are filtered, and the remaining candidates are represented with the corresponding prototype. Different rules for variable selection are implemented, e.g., the threshold over the frequency token counts. These methods are all *offline* because they require the presence of the data before building a model.

The *clustering* group considers the log parsing problem as a clustering problem. The logs that describe the same event are forming natural cluster groups because of the same words a message type is composed of. The general approach of these methods is to first find a suitable representation for the logs and apply some clustering algorithm to extract the message types. For example, LKE [51] considers the logs as sequences of words and applies weighted edit distance to calculate similar data groups. LogMine [58] similarly applies parallel implementation of a hierarchical clustering method where the distance is calculated based on a proposed equation. LogSig [160] considers a signature-based method to find the predefined number of cluster groups within the input log files. In contrast to the prior three methods that are *offline*, SHISO [123] and LenMa [84] are *online* parsing methods. They are building the groups by examining different properties of the logs (e.g., the number of words in the log, the number of characters in the words and similar) as they arrive. They aim to identify and append the most similar groups for each log. If no match is found new clusters are formed. Finally, the templates are updated to account for the specifics of all the logs in the group.

The *evolutionary* group considers the methods that adopt evolutionary algorithms for template extraction. MoLFI [120], as representative of this group, defines the problem of log parsing as maximizing the number of log messages matched by a single template while at the same time maximizing the number of unchanging tokens within the produced templates. To do so, it uses an evolutionary approach to find the Pareto optimal template set. It is an *offline* approach because it requires the log messages presented before producing the templates.

Log-structure heuristics methods exploit different properties that emerge from the structure of the log. The state-of-the-art algorithm, Drain [69] (in terms of accuracy), assumes that at the beginning of the logs the words do not vary too much. Relying on it, Drain creates a tree of fixed depth which can be easily modified for new groups. IPLoM [115] is another method from this group. It performs iterative partitioning of the log messages in the given log file. The partitioning is performed on three levels, token count, token po-

sition and partitioning by bijective relationships. Afterwards, the logs in similar groups are filtered based on the frequency counts. AEL [79] finds templates by a three-step procedure of anonymization (parameter masking), grouping the logs based on the number of words and a number of parameters, and template extraction. While Drain supports both, online and offline parsing, the latter two are offline parsers. The last method from this group, Spell [43], considers the problem of log parsing as the problem of finding the longest-common subsequence. It extracts templates from log messages having the longest-common subsequence match.

While there exists a plethora of approaches, we observed that there exists an alternative way towards parsing events, i.e., one that aims to mimic the operator's comprehension of the differences between the events and parameters from logs. Specifically, given the task of identifying all event templates in the logs, a possible approach to extract a template is to focus on constantly reappearing parts, while ignoring parts that change frequently within a certain context (e.g., per log message). The constantly repeating/changing parts can be determined from nearby words/tokens that form the context. Therefore, the task of log parsing can be framed as determining the variability degree of the log parts. By calculating the conditional probability of a particular token on a given position, where the conditioning is on its context it is expected that the tokens with high probability values will constitute the static text, while the tokens with low probability are the parameters. By this view, we propose a solution and complement the existing work on log parsing. The differences between the proposed methods and the related group categories are discussed in the corresponding chapter.

3.2.2 Log-based Anomaly Detection

Table 3.1: Summary of related works for log-based anomaly detection.

ID	Method	Parser	Input Type	Representation	Output Type	Semantic	Auxiliary Task	Threshold	Type
1	DT	Yes	Event Sequence Time Window	Event Count	Anomalous Event Count	NA	NA	Model Prediction	Supervised
2	LR	Yes	Event Sequence Time Window	Event Count	Anomalous Sequence	NA	NA	Model Prediction	Supervised
3	LogRobust	Yes	Event Sequence	Event Sequences	Anomalous Sequence	True	NA	Model Prediction	Supervised
4	CNN	Yes	Event Sequence	Event Sequences	Anomalous Sequence	True	NA	Model Prediction	Supervised
5	PCA	Yes	Event Sequence Event Parameters Time Window	Event Count	Anomalous Sequence	NA	NA	Automatic Threshold	Unsupervised
6	LogCluster	Yes	Event Sequence Time Window	TF-IDF Events	Anomalous Sequence	NA	NA	Automatic Threshold	Unsupervised
7	DeepLog	Yes	Event Sequence Event Parameters	Event Sequences	Anomalous Event	NA	NEP	Top_k Prediction	Unsupervised
8	LogAnomaly	Yes	Event Sequence	Event Sequences +Event Count	Anomalous Event	True	NEP	Top_k Prediction	Unsupervised

Multiple methods to the problem of log-based anomaly detection exist [154]. We identify eight properties of how to characterize the methods. Table 3.1 summarizes the related methods following our categorization. The *parser* property denotes if the method uses a specialized preprocessing technique on the raw logs to extract events, i.e., log parsing. The majority of the methods use log parsing because it reduces the variation in the data. The *input type* denotes the input format expected by the method. As input, single log events or event sequences can be given. The event sequences can be formed by external identifiers or time window grouping. Regarding the *representation* type, different approaches in the literature are considered, including event sequences, sequence events counts, single log events, or parameter values of individual events. Notably, some of the methods have multiple values for this parameter as it is possible for some of them to be composed of several components addressing different anomaly types. The *semantic* property is tightly related to the representation aspect of the logs. It denotes if general language modeling approaches are used to represent logs. Notably, some methods apply auxiliary learning objectives, e.g., supervised objectives in unsupervised model training. This can lead to ambiguities if a method should be considered supervised or unsupervised. Therefore, we find that it is important to make this property explicit. Thereby, the *auxiliary task* property denotes the usage of additional auxiliary tasks to support anomaly detection. With respect to the output, there are two important properties. The first one is *output type* which demonstrates the type of prediction, i.e., if the prediction refers to a single event or sequence of events. The second property related to the output is the approach for making a decision, i.e., how the output *threshold* is calculated. Considering the *information type* available during learning, similar to He et al. [154], we group the methods into two groups: supervised, and unsupervised. We use this property as the main discussion line to describe the methods in greater detail.

The supervised methods assume the existence of labels from the system when learning an anomaly detection model. Notably, the labels originate from the system of interest, which we refer to as the target system. In one of the earliest applications of these methods, Bodik et al. [6] consider the anomaly detection problem as a problem of binary classification. The authors apply Logistic Regression (LR) to detect anomalies from logs in data centres. Decision Trees (DT) [28] as another popular binary classification method is also used in detecting anomalous web requests from access logs. These two methods start with log parsing, i.e., extracting event templates from the raw input logs. Afterwards, they count the event templates in a fixed time interval to construct learning samples that proceeded as input to the anomaly detection method. They directly predict if the observed time window is anomalous. We refer to these two methods as traditional supervised approaches. The advances in deep learning resulted in the appearance of several supervised deep learning-based methods, i.e., LogRobust [180] and CNN [112]. LogRobust uses bidirectional Long Short Term Memory (LSTM) architecture [74] (a type of

LSTM architecture), augmented with attention [14]. These two are popular deep learning architectures often combined for modeling sequences. LogRobust, as input, receives a sequence of events, and as output, it directly predicts if the observed sequence is anomalous or not. An interesting feature of this method is that by careful construction of the sequences, i.e., with incremental growth by one element, the method can be used to predict single log line anomalies [44]. Figure 3.2 illustrates this process. An additional feature of LogRobust is the utilization of vector embeddings from general-purpose languages to represent logs (instead of template indices). Owing to the similarities of the natural language and logs [68], the authors show that semantic augmentation can slightly improve detection performance. Lu et al. [112] use the Convolutional Neural Network (CNN), another type of deep learning architecture, to learn normal and abnormal sequences based on template indices. Similar to LogRobust by careful sequence construction, CNN can be used to detect anomalies from a single log message. All of the supervised methods use parsing and the prediction of the trained model to detect anomalies. While having strong detection performance (due to the superior modeling objectives), the large dynamics of the software update and the large volume of the produced logs make the labeling process expensive. Therefore, supervised methods are often considered impractical [70].

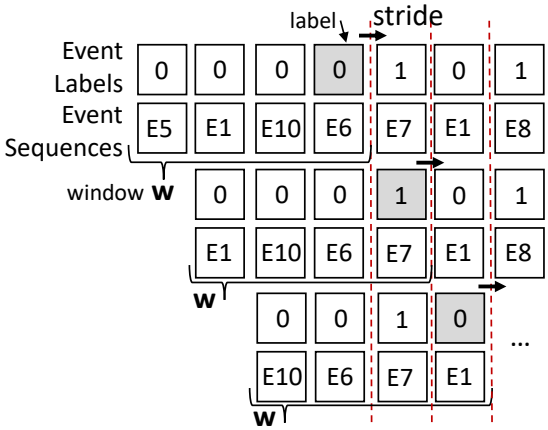


Figure 3.2: Example for how to use sequential log methods for single log line anomaly detection (window size is four, and the stride is one).

In contrast, the unsupervised methods assume an absence of labeled target system data. This assumption gives an important practical advantage because it eliminates the need for labeling, effectively making the unsupervised methods easier to adopt in practice. Many of these methods approach the problem as a one-class classification task. They model the normal system state (hence one class) and detect anomalies when significant deviations from it are detected. In one of the earliest works, Xu et al. apply the popular

unsupervised method Principle Component Analysis (PCA) [172] to learn the normal state of the log event counts. PCA projects the input data of size n into k principal components (i.e., k dimensions), such that $k < n$. The projection is done such that a large proportion of the variance of the input data is preserved (e.g., 95%). However, the remaining $(n - k)$ components describe the variation in the data that lie outside the major dimensions of variation. PCA leverages this observation. It first finds the top-k principal components, where the top-k is defined by the number of components such that 95% of data variation is preserved. By measuring the length of the projections of the anomalous space (the space formed by the $(n - k)$ components), the anomalous score is calculated. Afterwards, the new template event counts are projected in the same numerical space, and the anomalous score is calculated. The new event counts are labeled as anomalies if the anomaly scores are larger than a predefined threshold. Lin et al. [102] introduce LogCluster to model the normal state. LogCluster represents the sequences as numerical vectors of n elements, where n corresponds to the total number of events. The events are represented as a weighted average between their frequency of appearance in production as compared to their appearance in the system verification step during testing, and the frequency of appearance of an event within a single sequence (Term Frequency-Inverse Document Frequency (TFIDF)) analogue to word representations, on a sequence level. The obtained sequence representations are used to construct a knowledge base of normal/anomalous event sequences using agglomerative clustering and human domain knowledge. An automatic thresholding calculation technique is used in the construction of the knowledge base. When a new event sequence is introduced, the anomaly is detected if it is clustered into the anomalous cluster groups.

Several unsupervised methods directly model the normal sequences to learn the normal system state. Yamanishi et al. [173] introduce an unsupervised sequential method for anomaly detection that uses Hidden Markov Model (HMM) on log event sequences to model the normal state. The probability under the HMM is used as a normality score. Furthermore, there are two popular unsupervised methods of deep learning. Those are DeepLog [44] and LogAnomaly [117]. The innovative feature of these two methods is the introduction of an auxiliary task called Next Event Prediction (NEP). NEP is a supervised task that given a sequence of events, forecasts the most probable next event as a target. Notably, the targets originate from the input data itself, i.e., no labeling is performed, which makes these methods unsupervised. Any input presented at test time with an incorrect prediction for the next event is considered anomalous. The prediction is considered incorrect if the output scores do not rank the predicted next event within the top-k predictions. The latter is a hyperparameter of the method. As stated by the authors, DeepLog can be applied for sequential and single logs given as inputs. To learn the normal state, an LSTM architecture is learned on the NEP task. DeepLog also considers the problem of anomaly detection in the parameters of single events, referred to

as performance anomaly. To address the latter task, it trains an LSTM model on the parameters from a single event generated from the repetitive execution of events with numeric parameters. LogAnomaly addresses problems just within sequences. It has two additional features over the sequential part of DeepLog. These are 1) leveraging the semantics of the single log lines by applying general language embeddings, and 2) augmenting the input by event counts. The empirical results show that the improvements over DeepLog are not significant [31, 117]. Unsupervised methods are often criticized for their lower performance in comparison to the supervised ones [154]. The lower performance can lead to reporting many false alarms, leading to the phenomena of alarm fatigue [44], discouraging their wide applicability.

The given list of methods is not exhaustive. There are other methods for log-based anomaly detection in both industry and research [62, 76, 97, 98, 179, 188]. However, some of those solutions are part of production systems [97, 179], and have specific implementation challenges, and as demonstrated in Landauer et al. [88], not all provide public implementations. Therefore, due to the inability of a transparent comparison, we do not compare with them in our experiments. Instead, we give a short high-level overview of approaches with rather complementary undertakings on the task of log-based anomaly detection. OneLog [62] similar to Lu et al. [112] uses a hierarchical CNN where the words (their characters) and log sequences are learned in an end-to-end supervised manner. SwissLog [98] extracts temporal, semantic embeddings from the input sequences and trains bidirectional LSTM attention augmented network in a supervised manner to detect anomalies. UniLog [188] is a log analysis framework addressing three different tasks, alongside anomaly detection. The unique feature of this framework is that the four tasks are trained jointly and later, fine-tuned for a specific task. Despite the LSTM-based and CNN-based methods being the most popular choice for deep learning log-based anomaly detection, there exist other deep learning ideas adopted for the task. For example, Han et al. [60] use Generative Adversarial Networks (GAN) training of an LSTM for anomaly detection. Specifically, the authors propose to jointly learn a representation space of the normal source and target system data via an adversarial training procedure with hyperspherical loss. The training enables the concentration of the normal data close to a centre of a hypersphere, allowing the detection of the anomalies as points with large distances to the hypersphere centre. Zhao et al. [182] propose to use three Transformer encoder architectures with adversarial training to learn a model for anomaly detection. In contrast to the previous methods that rely on LSTM, CNN or Transformers, Wan et al. [167] adopt Graph Neural Networks (GNN) for log-based anomaly detection. The authors consider the individual log events within a session as nodes of a graph and use the data and the NEP task to learn vector embeddings for the individual nodes. The node embeddings of the session graph are aggregated within a session graph vector and used to rescale the individual node scores. If the next event is not ranked within the

top-k-ranked events, the session is considered anomalous.

Complementary to the aforenamed studies, three experience reports [31, 91, 154] give an independent empirical evaluation of the log anomaly detection methods. He et al. [154] studied three supervised and three unsupervised methods, making five observations. In one of the observations, the authors find that supervised methods are better performing than the unsupervised, however, in a second observation, the unsupervised are referred to have better practical properties. In another study, Chen et al. [31] examine four deep learning methods for log-based anomaly detection. They give four observations for the performance of the studied methods, with one of them providing a conclusion for the current state-of-the-art methods. Le et al. [91] study five deep learning methods on three benchmark datasets. The paper studies the methods through five different questions. Similarly, as in the previous cases, DeepLog, LogAnomaly and LogRobust are identified to be the most robust methods alongside all the addressed questions. In our work, we leverage the observation that an LSTM-based architecture for both unsupervised (DeepLog, LogAnomaly) and supervised (LogRobust) achieves superior performance from the deep learning methods and six other methods studied in He et al. [154].

In comparison to the related works on log-based anomaly detection, we found that there are possibilities for further improvement. We found that there exist higher-level properties of the individual log message (e.g., their semantics in the description of normal and anomalous events) or the generated log files (e.g., the existence of similar event groups) which can be used to learn anomaly detection models. Specifically, we contribute with two methods that attempt to improve the single textual representation of the log messages and their sequential characteristics when learning a model. The further specifics of the methods and their uniqueness in relation to the other works are given in the corresponding chapters.

3.2.3 Log-based Anomaly Classification

While anomaly detection is the first step toward failure resolution, there are additional steps between detection and diagnosing the root cause. The anomaly diagnosis is referred to the process of narrowing down the set of causes that led to the anomaly [70]. In general, the complexity of the modern system challenges the automation the anomaly diagnosis [170]. Nevertheless, studies on online systems (e.g., from Microsoft [102]) show that some anomalies are redundant, i.e., can occur repeatedly. Therefore, historical information can be used to speed up failure diagnosis. One approach toward anomaly diagnosis is anomaly classification. It is the task of finding the class of the anomaly from the known anomaly classes. The idea is that once the detected anomaly is correctly classified, past experiences can be applied to effectively resolve the anomaly.

A common way to diagnose the anomaly class, as in modern practices is by keyword search for finding a matching category [70] for a single log line. For example, in one of the earliest works, Oliner et al. [132] use keyword search of common words (e.g., "interface failure", "error", and similar) to identify different classes of anomalies within single logs of four different supercomputer systems. Meng et al. [118] use single log lines represented as bag-of-words, alongside the Random Forest method to classify different types of system logs. LogClass [119] proposes and uses TFILF as a representation method and supervised information for the anomaly class of the logs to categorize it.

As opposed to these approaches, there are several works addressing anomaly classification by using event sequences. The previously described method LogCluster [102] despite being used for anomaly detection, is also used for anomaly classification. The classification is achieved by assigning labels to extracted cluster groups. Whenever a new sequence is clustered to an anomalous cluster, the closest class label is assigned. Cotroneo et al. [35] propose to use a deep clustering approach to identify different failure modes of the event counts distributed tracing logging technology. Expert knowledge is then used to identify the different failure groups. In related work, Pham et al. [139] use *k-Nearest Neighbours* (kNN) with edit distance between observed anomalous sequence and the set of known prototypes anomalies for classification. Lu et al. [111] rely on domain expertise to extract five feature categories that categorize anomalies from Spark logs. By fitting linear models, they successfully categorize four different types of anomalies on a resource level (i.e., disk, CPU, memory and network). The first two methods belong to the unsupervised category, while the latter two are supervised approaches. Although some of the approaches require labels for training, while others do not, due to the nature of the task, method usage requires the involvement of human experts to categorize the anomaly. Therefore, the application of anomaly classification is bounded to scenarios where expert information even for past experiences is available.

Chapter 4

AI-enabled Dependability Framework with Log Data

Contents

This chapter describes the main challenges imposed by modern IT systems, the addressed problems, and a set of assumptions as enablers of methods design. The chapter is structured into two parts. In the first part, the challenges imposed by modern IT systems and the assumptions we considered to enable methods design, are presented. In the second part, we discuss how the proposed methods address the challenges by relying on the set of assumptions. Furthermore, the methods and ideas are conceptualized within the general AIOps platform, and a high-level overview of the proposed methods is discussed.

4.1 Challenges and Assumptions

The work presented herein aims to improve the logging code composition and anomaly detection and classification in IT systems as log analysis tasks. There exist two improvement aspects: 1) system development and 2) system operation. From the system development aspect, the improvement is achieved by improving logging instructions by

means of automatic logging code quality evaluation. From the system operation aspect, the improvement is achieved by introducing novel methods and datasets for anomaly detection that use system logs and external log-related data more effectively. The two aspects are associated with different challenges and require several assumptions, which we discuss in the remainder of this section.

4.1.1 Challenges

System Development Aspect

Automatic logging code quality evaluation is concerned with improving the logging code by detecting log instructions with insufficient quality. As such, it reduces the possibility that during development incorrect or insufficient information within the log instructions is inserted, which will be reflected in the generated logs. Therefore, the downstream tasks are indirectly improved. The automatic code improvement addresses the source code during development, leading to several challenges, given in the following:

1. *Heterogeneity of software events*: Different software systems provide different services. Naturally, their behaviour is described by diverse events. Since modern IT systems are commonly developed by many developers, their unique writing styles and cross-lingual biases lead to variations of the log instruction properties (e.g., the static text). Improving logging during development should consider diverse properties in the event descriptions (e.g., their vocabulary, writing styles, etc.).
2. *Different programming languages*: Software systems can provide similar services and functionality, but they can be written in different programming languages. The programming languages have a clearly defined but unique syntax. For example, a `for` code snippet in Python uses the `tab` character, as opposed to brackets ({}) in Java. The different syntax poses the question if 1) the developed tools for automatic logging quality evaluation should be specific for a programming language, or 2) one should limit the set of possible logging code quality assessment properties. For example, the different syntax of the neighbouring code around the log instructions from two programming languages (e.g., Java and Python) questions the applicability of log instruction placement methods that depend on the structure of the source code on arbitrary system [19]. The specifics of the programming language, thereby, challenge the design of automatic tools for improving logging code composition.
3. *Unknown empirically testable properties*: While there are universal aspects of logging code instrumentation, the system-agnostic set of properties subject to empirical evaluation is generally unknown. For example, the log level and static text syntax are common categories for the log instructions of many libraries (e.g.,

Python, and Java). However, it is not clear which syntactical elements of the static text can be evaluated empirically. Therefore, finding the set of empirically testable properties, e.g., by an empirical study, is needed.

System Operation Aspect

The system operation aspect is concerned with the efficient utilization of the generated system logs to detect and classify anomalies. The automatic approaches for anomaly detection and classification are challenged by:

1. *Diverse anomaly manifestation in logs*: The anomalies can reflect in the log data in different ways [44]. An anomaly may be explicitly recorded in one log message (e.g., connection lost), observed in a sequential change of the logs (e.g., flapping interface), or given as a parameter value of an individual event (e.g., log message reporting a long time to create a VM). Modeling one log property may lead to omitting important anomalies residing in the remaining properties. For example, for individual parameter values of a single event, an anomaly may be observed in the context of several manifestations of that event rather than a sequence of neighbouring ones. Its detection is possible within the context of the events generated from the same logging instructions. Therefore, careful consideration of the different properties exposed by the logs should be accounted for when analyzing the logs for anomaly detection and classification.
2. *Insufficient Logging Failure Coverage*: Developers may have an insufficient understanding of the complexities of the running system environment during development [92]. Inevitably not all failures can be logged. Therefore, there exist insufficient anomaly logging coverage. For example, in the previously described anomaly with type "Interface Flapping", none of the two logs has a log level with greater severity (i.e., "error" or "critical"), nor do they explicitly describe an anomaly. The anomaly can be detected just within the context of several repetitions of the specific pair of logs. Furthermore, these types of contextual anomalies occur often. For example, in Pike (version 3.12.1 of the popular cloud resource managing system OpenStack), there are more than 20% of anomalies that are not explicitly logged within a single log line [36].
3. *Complex Data Representation*: One complexity related to the input data type representation emerges when dealing with the complex nature of the logs. Logs have textual and sequential properties [117]. Therefore, the relevant information is contained within one or both. When analyzing log sequences, due to various reasons (e.g., different log preprocessing techniques, network errors, limited system throughput, and others), there is a possibility to drop or delay the reporting of certain events during monitoring. It results in log sequences having increased un-

certainty in the event order [180]. Despite the need to model sequences as complex data, the diverse log sequence appearance imposes an additional challenge.
Another challenging aspect related to the complexity of the input is concerned with the representation of a single log as a textual data format. Therefore, the supporting tools should consider their textual properties. Often the textual representation is obtained from general language models (e.g., BERT [40], word2vec [121]), where many dimensions are used to describe the textual properties – high dimensional representation. However, from an anomaly detection perspective, the detection of anomalies in high dimensional space is challenging due to the properties of such spaces [146]. Specifically, high-dimensional spaces are characterized by the property of large distances and sparsely populated regions. By definition, anomalies have large distances from the normal samples, therefore, it is difficult to disentangle the anomalies from normal data in high-dimensional sparse spaces.

4. *Software Evolution*: Agile software development leads to the insertion, modification or removal of logs at fast rates [70]. For example, a study on logging practices shows that 20-35% of the log code changes through the lifecycle of the software systems [83]. From the system operation perspective, this results in novel events and log sequences. Therefore, the data generation process describing the system behaviour changes. The log analysis should account for the novel, predominantly normal patterns and reduce their impact on the detection performance of the operational methods.
5. *Labeling*: A direct impact of software evolution is the increased cost of labeling. Since logs evolve at high rates, labeling new log sequences and log messages requires constant identification of novel normal or anomalous individual logs or log sequences from human experts. By further considering that logs not only evolve but are also generated in large volumes, labeling becomes an expensive task [154]. In addition, there is a possibility of deprecation of labels. Therefore, the reusing of past information is limited.
6. *Low Detection Performance*: A large number of logs and different operational frequencies (e.g., working-hours-related user-request patterns) make the detection and classification of anomalies similar to finding a "needle in a haystack" [70]. The operator's constructed rules and intelligent methods often result in a high alarm rate challenging their applicability to production [183], i.e., *have low detection performance*. The latter also extends to log parsing. If the alerts occur too frequently, operators second guess, skim, or even ignore incoming alerts. This problem is known as *alarm fatigue* [92] and increases the chances of important anomalies being missed. This can lead to severe implications for the system and its environment. Therefore, automatic methods are challenged by achieving an effective trade-off between high-alarm reporting and not-missing crucial events.

4.1.2 Assumptions

In the following, we discuss the assumptions enabling the development of intelligent methods to support the logging code quality evaluation and log-based anomaly detection and classification.

1. *Existence of open source code of systems with sufficient quality logging properties*: To examine the quality logging properties of a system, we assume that there exist software systems with sufficient logging quality. Following general literature on logging practices [24, 26, 68], the systems that a) are serving a vast set of industries, b) have many contributors, and c) are developed for a long period should have logging instructions of sufficient quality. The rationale is that given their vast application, the logging instructions have fulfilled their purpose for monitoring, debugging and troubleshooting. Therefore, the log instructions of the systems can be considered instructions with sufficient quality. They serve as a basis for quality evaluation. In addition, open-source code availability is important to enabling access to the source code of software systems with sufficient quality in the logging instrumentation. This assumption is important from the developer's aspect.
2. *Normality*: For the anomaly detection methods we assume that the majority of the logs originate from normal system operation. Since most of the time systems operate normally (as we observe in our service dependability study in Appendix A), and anomalies are rare events, the entirety of logs obtained during normal operation can be assumed as normal [183]. Further, following the availability of a large amount of data, the existence of a sufficient amount of normal data is assumed. Unless otherwise stated, anomaly labels are not available. These assumptions for the anomaly detection methods have important practical value as they demand the unsupervised design of the anomaly detection methods.
3. *Anomaly Detectability*: As described earlier in the text, the anomalies can be reflected differently in the log data. We do not impose constraints on how the anomalies are reflected in the log data beyond the three log properties. However, we consider the class of detectable anomalies. An anomaly is said to be detectable if it is reflected in at least one of the aforenamed log properties.
4. *Open-source Severity Level Data*: Considering that the severity/log levels are used to identify anomalies [102], we can consider that they have informative properties for anomaly detection. Furthermore, there are many publicly open-source codes with thousands of log instructions with available severity levels. Therefore, there exists a dataset of log instructions we refer to as *severity level data*. Although by leveraging this observation we can create potentially useful data it is not clear if there is merit in using it. Nevertheless, we can assume that we can access log instructions (i.e., their static text and severity level) from open-source systems.

5. *Recurrence*: Following previous studies [102, 119], for software systems, particularly online and large-scale ones, the anomalies are characterized by the *recurrent* property. It means that the same anomaly can occur more than one time. Several reasons are suggested for the validity of this assumption, including a) when a service fails, a common practice is to restore the service availability, typically by identifying a workaround solution (such as restarting a server). Given this, it is expected that similar log patterns leading to the observed issue will re-occur before the root cause is fixed; b) Online services are usually modular with diverse components running in different computing environments. Therefore, an issue occurring in one environment may repeat itself in other environments as well; c) Due to weaknesses in hardware and software, similar events (e.g., machine down, switch failure, or network disconnection [102, 119]) occasionally occur. They can lead to similar repetitive patterns in logs. Since the recurrent anomalies are previously addressed, there can be a lot of redundant effort in diagnosing them. Thereby, the recurrence assumption enables reusing past information to speed up the overall anomaly resolution. Notably, this assumption implies that generalization besides the set of already seen problems is not possible, thereby, limiting the generalization requirement to novel anomaly classes. Furthermore, labels from a domain expert for the anomaly types are also assumed. As such the automatic classification of anomalies is enabled.

4.2 Conceptual Overview

To address the challenges, in this thesis, we propose intelligent data-driven methods and ideas that support log-related activities during system development and operation. The proposed methods support the whole logging cycle during system log instrumentation and the analysis of the generated logs. They are basic components of a broader platform, referred to as an AIOps platform. Figure 4.1 gives an overview of a reference architecture based on the target *system under development and operation*. The term *system under development and operation* refers to an individual component of a larger system, or in certain scenarios, the whole system itself that is subject to analysis. In the following text, we give an overview of the presented methods and ideas, and their integration into the AIOps platform.

Since the presented methods and ideas support the two phases of the system life-cycle, there exist two main modules, i.e., 1) *automatic log code improvement*, and 2) *log analysis* modules. The contributions from the *automatic log code improvement* module are contained within the component *log instruction quality evaluation*. This component implements methods to evaluate the logging code composition quality. It analyzes the source code, extracts the log instructions, evaluates their quality and reports them to

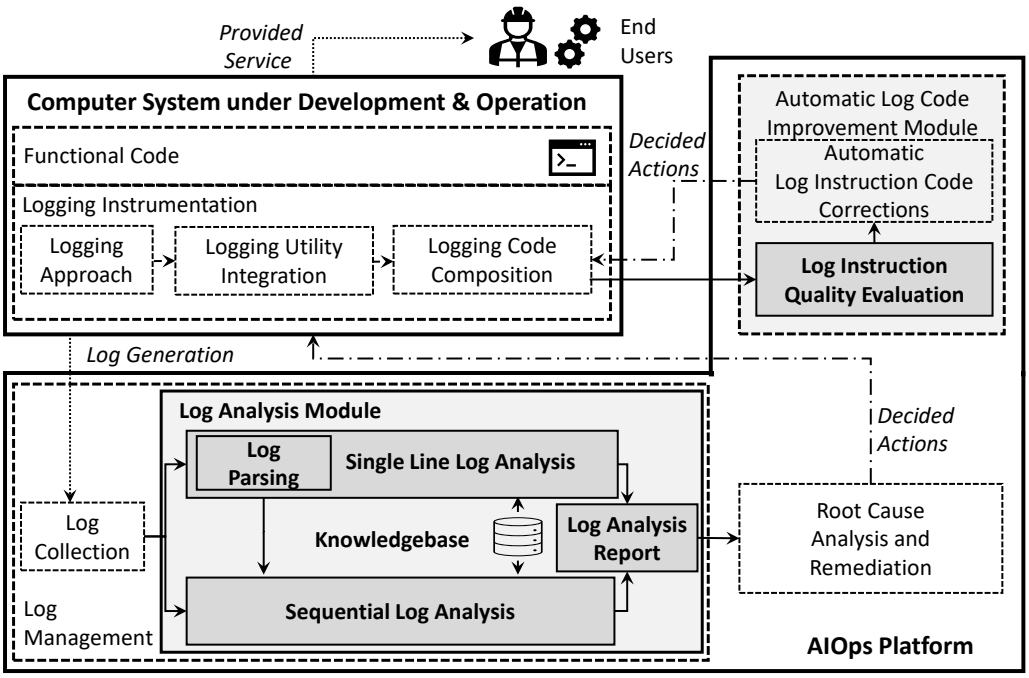


Figure 4.1: Overview of the thesis contributions within the overall AIOps framework on log data.

other components of the module. The other components act upon the suggestions to improve the logging code. In the AIOps platform, the automatic improvement components require additional decision-making processes on logging improvement and are outside the scope of the thesis.[1]

The *log analysis* module receives the generated logs from the running system. The main goal of the *log analysis* module is to correctly detect anomalies, potentially identify their class and report them. Therefore, it proposes novel methods for log processing (parsing) and anomaly detection. The greatest strength of the proposed methods is the improvement of the log representation by learning features suitable for anomaly detection with deep learning methods. As the anomalies reflect in different log properties, the log analysis is split into two components performing 2.1) *single log line* and 2.2) *sequential* log analysis. The two components follow a sequential chain of first detecting and then identifying the class of the anomaly. The reported output is given to external components of this module. These components involve an additional decision-making process to decide on the most suitable action for eliminating the anomaly and are outside the scope of the thesis. Although the two modules are parts of the AIOps platform, their output

[1]Dashed lines illustrate the components that are outside the scope of the contributions, however, they are important to describe the concepts and define the boundaries of the contributions.

can be reported directly to the operators. Note that although debuggers are a primary source for error reporting during development, the log analysis methods may be used by developers during debugging. We discuss the two main modules in greater detail in the remainder of this section.

4.2.1 Automatic Log Instruction Code Improvement

Log Instruction Code Quality Evaluation

Part of the development of a software system includes writing log instructions in the source code. The log instruction quality assessment submodule analyses the source code of the system to evaluate the log instruction quality. From the perspective of logging, we can separate the code instructions into functional (non-logging) and logging instructions. The functional code implements the functions the system/component is serving. Once developers choose the logging approach and logging utility, they instrument the source code with logging instructions, most often in an afterthought process [23]. The examination of the log instructions, therefore, requires access to the system's *source code*.

Once the logging instructions are written in the source code, their quality can be evaluated. Specifically, under the term log quality evaluation, we understand the correctness of the logging properties (e.g., correct log level concerning the static text, sufficiently rich static text and similar) compared to software systems with assumed sufficient-quality logging properties. There are two central questions with respect to this. The first question refers to which software systems are assumed to have sufficient logging quality. We address this question by referring to the assumption of the *existence of open source code of systems with sufficient quality logging properties*. The second question attempts to answer which quality properties can be automatically evaluated, given the different real-world complexities, i.e., the challenges of *different programming languages*, *heterogeneity of software events*, and *unknown empirically testable properties*. To address the second question, we conduct an empirical study to identify empirically testable properties.

Figure 4.2 depicts a detailed overview of the log instruction quality submodule. As input, it receives source code snippets (set of instructions) with logging instructions. It consists of four components, i.e., 1) *quality knowledge base*, 2) *log instructions extraction and preprocessing* component, 3) *quality evaluation*, and 4) *quality reporting* component. The *quality knowledge base* is composed of code snippets with logging instructions from systems with sufficient quality logging. It is used to create the training data for quality properties evaluation of the *system under development and operation*. The *log instruction extraction and preprocessing* component extracts the log instructions from the source files of the *system under development and operation* and proceeds it as input to the quality evaluation component. The *quality evaluation* component consists of a set of individual

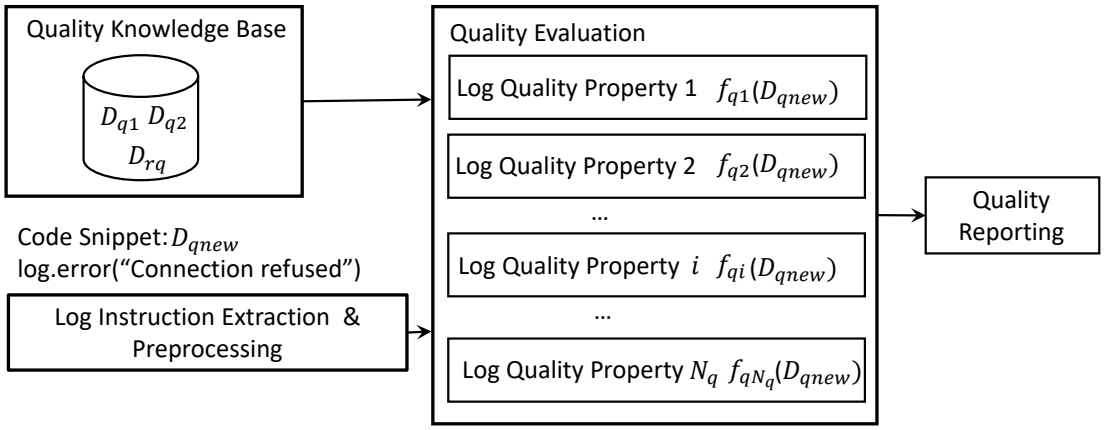


Figure 4.2: Log instructions quality evaluation component.

subcomponents, each of which evaluates a single quality feataure. The evaluation results from the quality component proceed to the *quality reporting* component. The *quality reporting* component reports the output results to external entities for improvement. An example of an external entity is an autonomous tool of the AIOps platform that decides on the suggestions (e.g., replacing the wrong log level with the log level given by the model, enriching the static text, and similar).

Without loss of generality, we can formally define the log instruction quality evaluation as follows. Let $D_q = \{D_{q1}, D_{q2}, \ldots, D_{qj}, \ldots D_{qr_q}\}$ is a set of r_q snippets from the source code of software systems with assumed *sufficiently* good logging quality having one and only one log instruction, $\mathbb{N}_q$ denotes the set of empirically testable properties expressed with a categorical data type (C_i) with $|C_i|$ categories each, and $F_q = \{f_{qi} | f_{qi} : D_q \mapsto C_i, i \in \mathbb{N}_q\}$ is a set of empirically quality testable properties, such that D_{qj} is a structured string defined by the syntax of the programming language P_q it originates from, and f_{qi} is a function of the quality feature i. The quality vector of a novel code snippet D_{qnew} from a target system is then given as $\hat{Q}_q(D_{qnew}) = (I(C_1, \hat{C}_1), \ldots, I(C_i, \hat{C}_i), \ldots I(C_{|\mathbb{N}_q|}, \hat{C}_{|\mathbb{N}_q|}))$, where I is an indicator function having value 1 when $C_i = \hat{C}_i$, and 0 otherwise. The C_i denotes the observed manifestation of the quality in the codding snippet D_{qnew}, while $\hat{C}_i$ its prediction obtained when the elements of the set F_q are applied on D_{qnew}.

The key benefit of automatic log instruction quality evaluation is that by detecting inconsistencies within the logging instructions (e.g., writing static text with a minimal linguistic structure to preserve comprehensiveness) and resolving them during development, the downstream tasks can be improved. For example, a frequent practice when debugging/detecting anomalies is to search for log levels with the value "error" [102]. If the log level is lower (e.g., "info"), the time for locating the failure may increase if

the above strategy is adopted for failure resolution. By including an independent automatic evaluation step, the chances that such inconsistencies are present in the generated logs can be minimized. Therefore, the automatic control for quality indirectly aids the subsequent steps of log analysis.

4.2.2 Log Analysis: Log-based Anomaly Detection and Classification

Once the system is instrumented and it is running, logs are generated. The users interact with the system with different rates conditioned on the working hours, service types and similar, defining system working frequencies. The generated logs are collected and passed for analysis. We assume that the logs are aggregated on the component level. Furthermore, we assume that the collected logs have at least two attributes, i.e., timestamp and event description (the log message). These are realistic expectations as most of the log collection procedures consider this information as important and retrieve it. Some of the proposed methods (i.e., the semantic anomaly detection method) within this thesis can work with just the log message but others require at least the information on time. In addition, logs can have external identifiers (e.g., taskIDs, Process Identifiers (PIDs)) that can be used, e.g., for constructing log sequences. Note that we implicitly assume that each log is a function of time, as the event must have happened at one point in time.

The collected logs are given as input to the log analysis module. The log analysis module is composed of three components. These are: 1) *single log line analysis*, 2) *sequential log analysis* and 3) *log analysis reporting*. Since the anomalies can be reflected in individual logs (including log parameters) or log sequences (the challenge of *diverse anomaly manifestation in logs*), we split the log analysis into two components, i.e., *single line* and *sequential* log analysis. The collected logs first are proceeded to the *single log* line analysis module and are parsed such that the static text and parameters are extracted. Log parsing should be performed with as little information loss as possible as the parsing errors compound, and affect the remaining tasks. As parsing is performed on a single log event it is part of the *single log analysis* module. Despite the log parsing, the remaining part of the single and sequential log analysis components are conceptually equivalent. Within both, there are two subcomponents a) *anomaly detection* and b) *anomaly classification*. The *single line* subcomponents exploit the single log line property. The *sequential* subcomponents exploit the co-occurrence of the events, i.e., the sequential (contextual) log property. By referring to the assumption of *anomaly detectability* we assume that the anomalies that are within the scope of the detectable anomalies can be detected. As the software logs are generated in large volumes and the system most of the time operates normally (as seen by the failure study in Appendix A) we consider the *normality* assumption as valid. Once the anomalies are detected, independent of the sequential or single log analysis component, they proceed towards the classification subcomponents for

the two properties separately. The classification subcomponents use past information to classify the type of detected anomalies. Notably, we refer to the *recurrence* assumption when anomaly classification is performed for both the single and sequential log analysis. The logs classification as a precondition requires knowledge about the unique properties of the detected anomalies, i.e., their classes. Due to the *labeling* challenge, the classification cannot be performed on each of the detected anomalies because it requires external information about the anomaly classes. The results from the anomaly detection and classification proceed towards the *log analysis reporting* component. From there, they are presented towards other modules for automatic remediation of the anomaly or to the entities performing the system operation/development. The log analysis methods aim to address the challenges during system operation (i.e., *software evolution, complex data type representation, low detection performance*) by considering different approaches for improving the representation of the logs. Another important aspect during the development of methods is associated with the *labeling* challenge. Owning to the expense of obtaining labels, we opt for the unsupervised design of the anomaly detectors. In the following, we discuss the components in more detail.

Single Log Line Analysis

Log Parsing; Since each system has a finite number of logging instructions, there is a finite number of possible log events in the generated logs. The parsing aims to extract the set of events from the log instructions. One important aspect of a good log parser is that it should produce robust performance over a diverse set of software systems (e.g., from mobile applications up to cloud systems). Literature reports suggest that incorrect parsing can drop the performance for anomaly detection up to 60% [91] (i.e., the challenge of *low performance* of related methods). Thereby, the correctness of parsing affects the remaining log analysis tasks stemming from it. To address the problem, we propose a novel log parsing method. The raw logs are given as input to the log parsing. As output, the log parsing proceeds with the raw logs, augmented with two columns for the event's template and parameter list.

Formally the problem of log parsing is defined as follows. Let $\mathbb{L}_p = \{l_1, l_2 \dots l_i \dots l_n\}$ be a set of n time-ordered logs from the computer system under development and operation, and $T_p = \{(s_{p1}, p_{p1}), \dots (s_{pj}, p_{pj}) \dots, (s_{pt}, p_{pt})\} = S_p \times P_p$ is the set of log instructions in the source code, such that l_i is a generated log message, s_{pj} is a correct static text of the j-th log instruction, p_{pj} is the parameter set associated with the log instruction with index j, S_p and P_p are the sets of static texts and parameters. The goal of log parsing is finding $f_p : \mathbb{L}_p \mapsto S_p$. The elements of the set S_p are sequences of characters. The elements of P_p are lists of variables, where the variables can have different data types (e.g., strings, floats, or other objects (considered as strings)).

The parsed logs proceed as inputs for the parametric anomaly detection and the sequential log analysis and reporting components. The proposed log parsing method is discussed jointly within the single log analysis part. We discuss the remaining log analysis components in the following text.

Anomaly Detection and Classification; To detect anomalies during system development and operation, single log lines are often used [102]. For example, developers/operators search the logs with different keywords like "failure", "error", "unable", "dropped" and similar. Therefore, analyzing single logs is useful for system operational activities. Figure 4.3 depicts the single line component for log analysis. Despite the log parsing, there are two parts, i.e., 1) *single log line anomaly detection* and 2) *single log line anomaly classification*. Individual logs expose two categories of information, i.e., semantics and parameter values. Accordingly, the single log detection consists of two subcomponents a) *semantic anomaly detection* and b) *performance/parameter anomaly detection*. The anomaly detectors receive the raw logs and the extracted events and parameters as input. Notably, the semantic anomaly detector presented herein with suitable preprocessing of the logs can be applied to the raw log messages eliminating the potential inaccuracies from log parsing. The performance anomaly detector compares the events generated from single log instruction through time. Any significant deviations in the parameters are reported as performance anomalies. To train single line anomaly detection models, we refer to the *normality* assumption, i.e., we assume that there exist representative normal data from the system. Furthermore, the proposed semantic model refers to the assumption of the availability of the *open-source severity level data*. Since the severity level data describes diverse normal and anomalous events from many different systems, it encodes discriminative properties between the normal and anomalous event types. The proposed method combines the system and the open-source data to learn good features for the semantic anomaly detection model.

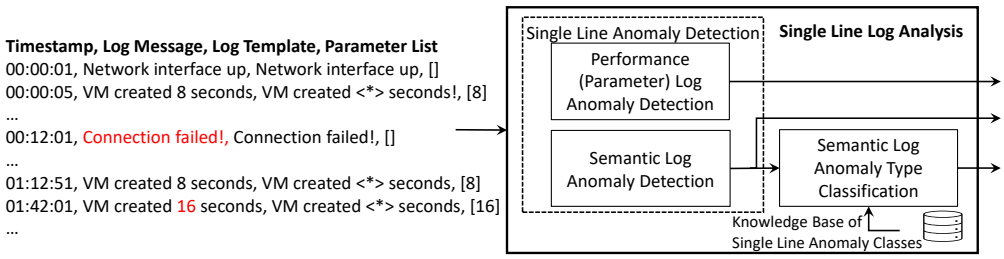


Figure 4.3: Single log line analysis module overview.

Once the semantic anomaly is detected in the log, it proceeds to the anomaly classification subcomponent. The anomaly classification sub-component assumes the existence of

a knowledge base which stores prior observed log events with their class label. Therefore, the detected anomaly can be classified. Note that the anomaly detectors can detect anomalies that the classification components cannot classify (recall that the *recurrence* assumption is valid just for the classification sub-component). The classification of performance anomalies is redundant as the parameters directly identify the anomaly class type. In case no class is identified, the log message is assigned to an unknown class. In the following, we formalize the semantic and performance anomaly detection. Afterwards, we define the task of single log line classification as well.

Semantic Anomaly Detection (semAD); Let $\mathbb{L} = \{l_1, l_2 \dots l_i \dots l_n\}$ be a set of n time-ordered logs from the system under development and operation, and there exist a function $p^+_{sem}(.)$ denoting the normality score of the individual log lines of the system $p^+_{sem}(\phi_{sem}(l_i)) : \mathbb{R}^d \mapsto \mathbb{R}$, where $\phi_{sem} : \mathbb{L} \mapsto \mathbb{R}^d$ is the representation function of the log l_i into d-dimensional numerical vector space. The task of single line anomaly detection is defined as finding the set $A_{sem} = \{l_i | a_1 > p^+_{sem}(\phi_{sem}(l_i)) || p^+_{sem}(\phi_{sem}(l_i)) > a_2, l_i \in \mathbb{L}\}$, where a_1, a_2 are constants such that $a_1 < a_2$, and $a_1, a_2 \in \mathbb{R}^+$.

Performance Anomaly Detection (perAD); Let $\mathbb{L} = \{l_1, l_2 \dots l_r \dots l_n\}$ be a set of n time-ordered logs from the system under development and operation, $\mathbb{E} = \{e_1, \dots e_i \dots e_v\}$ is the set of v possible events, such that $l_r(t_j) = (e_i, p_{ik}(t_j))$, where e_i is the static text of the event, while $p_{ik}(t_j)$ is the k-th parameter for the event e_i at time t_j. Let there exist set of function $P^+ = \{p^+_{ik}(p_{ik}(t_j); e_i) | p^+_{ik}(p_{ik}(t_j); e_i) : \mathbb{P}_{ik} \mapsto \mathbb{R}, e_i \in \mathbb{E}\}$, where $p^+_{ik}(p_{ik}(t_j); e_i)$ denotes the normality score of the k-th parameter of the event e_i, and $\mathbb{P}_{ik}$ is the set of allowed values for the parameter p_{ik}. The goal of performance anomaly detection is finding the logs l_i with deviating parameter values, i.e., $A_{per} = \{l_r | a_{ik1} > p^+_{ik}(p_{ik}(t_j); e_i) || p^+_{ik}(p_{ik}(t_j); e_i) > a_{ik2}, l_r = (e_i, p_{ik}(t_j)), p^+_{ik} \in \mathbb{P}^+, l_r \in \mathbb{L}\}$. The thresholds a_{ik1}, a_{ik2} are calculated for each parameter p_{ik} separately and $a_{ik1} < a_{ik2}$, $a_{ik1}, a_{ik2} \in \mathbb{R}^+$. Notably, in the definition, each event e_i is associated with at most k-parameters.

Semantic Anomaly Type Classification (semATC); Given a set of detected single line anomalies A_{sem} and the set of anomaly types classes $T_{sem} = \{t_1, t_2 \dots t_w\}$, where w denotes the number of unique anomaly type classes, the task of single anomaly type classification is finding a function $f_{sem}(\phi_{sem}(l_i)) : A_{sem} \mapsto T_{sem}$.

The detected anomalies and their classes proceed toward the logging analysis report. They augment the input data with three columns. The first column indicates the presence of a semantic anomaly for the corresponding event. The second column has a JSON structure with keys being the event parameters and their values information for performance anomaly. The third column shows the class of the semantic anomaly.

Sequential Log Analysis

While single log line analysis is useful for detecting and classifying anomalies, it is not sufficient for comprehensive log analysis. Recognizing that anomalies reflect differently in the log data (i.e., the challenge of *diverse anomaly manifestation in logs*) and that there is *insufficient logging failure coverage*, it is clear that not all events denoting anomalies are written in the source code. Correspondingly, some anomalies cannot be detected in a single log line. Nevertheless, a set of anomalies in these cases can still be detected, e.g., by comparing the events' cooccurrence (the assumption of *anomaly detectability*) in form of sequences or counts. One challenge in this context is related to how to represent the *complex sequences*, in the circumstances of parsing inaccuracies, dropping or delaying event reporting. These circumstances cause diverse normal sequences, which are referred to as *unstable* sequences [180].

The sequential log analysis component works in parallel with the single line component. Figure 4.4 illustrates the inner structure of the sequential log analysis component. It has three subcomponents, 1) *sequence creation*, 2) *sequential anomaly detection* and 3) *sequential anomaly classification*. As input, it receives the parsed log events and the other meta information for the events (e.g., timestamps). They are given as input to the *sequence creation* subcomponent. If the log identifiers are available, they are used to create the log sequences (e.g., task identifiers (taskIDs)). In the cases where no identifiers are available, the sequences are created based on predefined time windows. A time window is defined as a hyperparameter of the sequential component (e.g., 60 seconds). It groups all the logs within the specific window to create sequences. Similar approaches for log sequence creation are existent in related works [91, 154]. The sequences are given as input to the *sequential anomaly detector* and *anomaly classification* subcomponents. The anomaly detector implements a method that improves the representation of the log sequences by representing them as a sequence of event groups. By improving the representation, the detection and classification performance can also be improved. The detected anomalies proceed toward the sequential classification subcomponent. The latter is related to a knowledge base of past sequential anomalies. As output, it provides a class label for the detected sequential anomaly and its class. These two proceed towards the *log analysis reporting* component.

In the following, we formally define the two tasks addressed by the *sequential log analysis* component.

Sequential Anomaly Detection (seqAD); Let $L = \{l_1, l_2 \dots l_i \dots l_n\}$ be a set of n time-ordered logs from the system under development and operation, and there exist an index set $\mathbb{J} \in \mathbb{N}$ capturing dependency relation between the logs, i.e., $s_j = (l_{ji} \in L | j \in \mathbb{J})$, where l_{ji} denotes individual log of the sequence s_j. Further, we assume that there exist

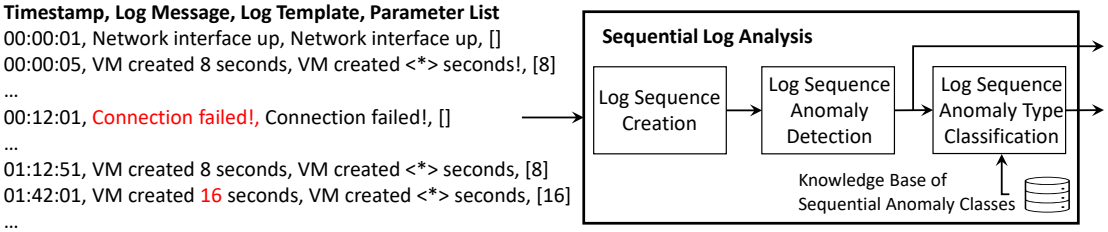


Figure 4.4: Sequential log analysis module overview.

a function $p^{+}_{seq}(.)$ denoting the normality score of the sequence $p^{+}_{seq}(\phi(s_j)) : \mathbb{R}^d \mapsto \mathbb{R}$, where $\phi_{seq}(.) : \mathbb{S} \mapsto \mathbb{R}^d$ is the representation function of sequence s_j into d-dimensional numerical vector space, and $\mathbb{S}$ is the available sequence set. The task of sequential anomaly detection is defined as finding the set $A_{seq} = \{s_j \in \mathbb{S} | a_1 > p^{+}_{seq}(s_j) || p^{+}_{seq}(s_j) > a_2, j \in \mathbb{J}\}$, where a_1, a_2 are constants such that $a_1 < a_2$ and $a_1, a_2 \in \mathbb{R}^{+}$. Although the individual logs l_i in the sequence s_j can describe normal events, the overall sequence can be anomalous. The index set $\mathbb{J}$ in the context of logs can represent task ID or workload ID. It can be given apriori or reconstructed by an additional sequence creation procedure.

Sequential Anomaly Type Classification (seqATC); Given a set of detected anomalous sequences A_{seq} and the set of anomaly types identifiers $T_{seq} = \{t_1, t_2 \dots t_w\}$, where w denotes the number of unique anomaly type identifiers, the task of sequential anomaly type classification is finding a function $f_{seq}(\phi_{seq}(s_i)) : A_{seq} \mapsto T_{seq}$.

Notably, the single log line and sequential analysis components have different working frequencies. The sequential method has a minimal working frequency defined on the task identifier or the defined time window. In contrast, the single log line can report results after each input log.

Log Analysis Reporting

The log analysis reporting component serves as a connector of the log analysis toward the modules of the AIOps platform. Figure 4.5 shows how the input logs are transformed by the log analysis module and proceeded as output from the log reporting component. The log reporting component receives the information for the detected anomalies/alarms (both single line and sequential) and potentially their classes. Also, the corresponding anomalous time intervals are reported. In the context of the AIOps system, the report proceeds toward the root cause analysis, remediation and recovery components to realize the fully autonomous system operation. These components are outside of the scope of the thesis and are not discussed further. In addition, the reported information can proceed to the operators or the end-user for further analysis or to enrich the knowledge base.

Timestamp	Log Message
00:00:01	Network interface up
00:00:05	VM created 8 seconds
...	
00:12:01	Connection failed
...	
01:12:51	VM created 8 seconds
01:42:01	VM created 16 seconds
...	

(a) Collected logs.

Timestamp	Log Message	Log Template	Parameter List	Semantic Anomaly	Semantic Anomaly Class	Performance Anomaly
00:00:01	Network interface up	Network interface up	[]	0	0	{}
00:00:05	VM created 8 seconds	VM created <*> seconds	[8]	0	0	{"param1": 0}
...						
00:12:01	Connection failed	Connection failed	[]	1	Connection Failure	{}
...						
01:12:51	VM created 8 seconds	VM created <*> seconds	[8]	0	0	{"param1": 0}
01:42:01	VM created 16 seconds	VM created <*> seconds	[16]	0	0	{"param1": 1}
...						

Sequential Anomaly Summary:
Anomaly Detection: Sequential anomaly was observed in the period, 00:12:00– 00:13:00, Time Window: # 12
Anomaly Type Class: Network Failure

(b) Output from the log reporting component (log analysis module).

Figure 4.5: Illustrative example of the log analysis reporting component.

In the following three chapters, we delineate the specifics of each of the introduced methods. We first discuss the contributions to system development, and afterwards, the contributions to system operation.

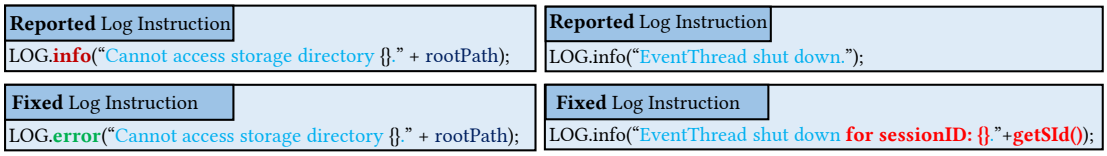
Chapter 5

Automatic Logging Code Composition Quality Assessment

Contents

Developers are adopting diverse logging utilities and specify guidance on the *quality* requirements when writing log instructions [19]. 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 [23]. The quality guidelines enable writing log instructions with good quality [19]. However, as discussed in Chapter 3, the studies on industrial [19] and open-source software systems [23] suggest that developers make recurrent log-related commits

during development. This implies that writing quality log instructions is often followed by additional improvements even with given quality guidelines [24].



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

(b) Jira issue ZOOKEEPER-2126: Insufficient information hurts event understanding.

Figure 5.1: Examples of issues related to log instructions quality.

In modern system development, the decisions about the log instructions are most commonly human-centric, which sometimes can result in log instructions with insufficient quality (e.g., wrong log level assignment or insufficient linguistic structure) [1, 94]. For example, in the Jira issue HDFS-4048[1] (depicted in Figure 5.1a), 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[2] (depicted in Figure 5.1b), the log instruction has insufficient information about which EventThread is terminated. As reported by the developers in the aforenamed issue, 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 [23, 94] suggest that it is surprisingly common for the log levels to be over/under-estimated or the logs to have missing or excessive information. Although the human-centric approach in log quality assessment is the golden standard, the existing issues imply the need for an automatic approach to access the quality of the logging code.

The automation of log instruction quality is challenged by the *heterogeneity of software events*, *unknown empirically testable properties*, and the *different programming languages*. To address the challenges we assume the *existence of open source code of systems with sufficient quality properties* as a minimal prerequisite for automation. This chapter presents the contributions towards aiding the development of software systems in improving their

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

logging code composition as follows[3]:

1. We observe that some of the properties (i.e., log level assignment and linguistic evaluation) depend on and can be assessed from the content of log instructions. Therefore, they can be evaluated irrespective of the structure of the source code and the remaining logging code properties. To verify our observation, we study the log instructions from nine open-source systems, with assumed good logging practices (similar to [23, 100, 154]) that form an initial quality knowledge base.
2. We formalize the problem of assessing the log instruction quality.
3. By leveraging our observations and the textual nature of the log instructions, we propose a deep learning method for model-driven log quality assessment as an intelligent tool to aid the writing of log instructions.
4. We adopt an approach from explainable machine learning to provide augmented feedback for possible places for log instructions quality improvement.
5. The experimental results conducted on logging instructions from open-source systems with varying quality demonstrate the usability of the proposed approach.

The remainder of this chapter describes our approach toward automating the logging code quality assessment. Section 5.1 describes the system-agnostic testable log instructions quality properties. Section 5.2 introduces QuLog as a method for automatically assessing log instructions quality. Section 5.3 provides the evaluation of the individual parts of the proposed method. Section 5.4 summarizes the chapter.

5.1 Logging Code Composition Quality Properties

To assess the quality of the log instructions, we examined literature studies on logging practices [23, 26, 181]. We identified two views on logging code quality: *explicit* (or system development), and *implicit* (or system operation). The explicit view is related to (a.1) correct *log level assignment*, (a.2) *comprehensive content of the static text and parameters*, (a.3) *correct instruction placement* and (a.4) *supporting logging code placement* [64]. The implicit view is related to the system operation expectations for the quality of the logs. For the implicit view, there are four properties, given as follows: (b.1) *trustworthiness* - refers to the valid meta information of the log (e.g., correct log level), (b.2) the *semantics* of the log - relates the word choice in the verbose event expression, (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 [1].

Some of the aforenamed properties (i.e., relevant variable selection, log instructions placement, safeness, and completeness) depend on the different programming languages, design patterns, and other source code structures [1, 100]. These properties are challenging to

[3]Parts of this chapter are published in [9].

assess across different systems and programming languages because of the heterogeneity of software systems and the ways programming languages organize the source code, i.e., their syntax. For example, *safeness* requires reasoning across a complex chain of method invocations. In the issue CVE-2021-44228[4] the bug allows execution of any Java method through the log instruction from an LDAP server decreasing the safeness. Identifying safeness in this example requires a deep understanding of potential method invocation chains, which do not even require the method's presence within the source code. The latter is against our effort in automatic log quality assessment. However, by considering various studies on logging practices [23, 24, 26] and Jira issues, we observe that some of the properties (i.e., correct log level assignment and linguistic evaluation) depend on and can be assessed just from the content of log instructions. Therefore, they can be evaluated independently to the structure of the source code and the remaining logging code. To examine the observation, we conducted an empirical study of the logging instructions from nine open-source systems (considered in Li et al. [100]), with presumably good logging practices (similarly assumptions exist in related works [64, 100]). Table 5.1 enlists the properties of the studied systems. These systems form the *initial* (quality) knowledge base.

Table 5.1: Overview of the studied systems for log quality assessment. These systems form the *initial* quality knowledge base.

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.

5.1.1 Log Level Assessment

In the conducted study, we observed that the static text of the log instructions may have relevant features for log level assessment. Intuitively, when describing an event with the "error" log level, the static text commonly contains words like "error", "failure",

[4]https://nvd.nist.gov/vuln/detail/CVE-2021-44228

"unexpected exit", and similar. Whenever these words occur within the static text, it is more likely that the level is "error" than "info". Similar to Hassani et al. [64], we considered an approach from information theory that defines the amount of uncertainty of information in a message [37] to investigate the relation between the static text and the log levels. We extend the experiments to nine systems, as opposed to the two studied in the corresponding study. We analyze the relation of word groups (n-grams, $n = \{3, 4, 5\}$) from the static text in relation to the log level. N-grams are simple, yet effective representations for texts that can successfully model relationships between words based on their co-occurrence [158]. For all the n-gram groups, we try to identify the log level using n-grams from the given static text of the log instructions. 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 Shanon's entropy [63]. This approach enables to study of the uniqueness of the n-grams in relation to the different log levels. 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 with "error", and once with "info". By calculating the entropy we obtain a low number (0.05), which reflects low uncertainty about the n-gram word association with a level other than "error". To limit the influence of the rear n-grams, similar to He et al. [68], we further analysed the n-grams that appear at least three times.

Table 5.2: Empirical study: Log level assignment.

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

Table 5.2 summarizes the n-grams entropy distribution. It is seen that the majority of the static texts of the log instructions have low entropy with respect to the log levels. Specifically, more than 50% (the median) of the population of static text n-grams have zero entropy. The zero entropy means that most of the n-grams are associated with a single log level. By using the static text as input, one can distinguish among the different log levels across the different systems. Thereby, *the static text has relevant features useful to discriminate the log levels* across the studied systems.

5.1.2 Linguistic Quality Assessment

A quality log instruction should describe the event concisely and verbosely [23]. From a general language perspective, complete and concise short texts should have a minimal

linguistic structure (e.g., usage of nouns, verbs, prepositions, adjectives) [46]. 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 log instruction from the Jira issue ZOOKEEPER-2126 (depicted in Figure 5.1b), the static text "EventThread shut down." linguistically is composed of "noun verb particle". Since the general English language and language used in log instructions share similar properties, we point out that an informative event description may also have a minimal linguistic structure. The following example explains our intuition. In the aforenamed Jira issue (Zookeeper-2126), 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, static text with *insufficient* linguistic structure is transformed into static text with *sufficient* structure, improving the event description and potentially its comprehensibility.

To examine the extent of validity of our observation, we perform 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. It allows extraction of the linguistic structure of the static text by linking the words to linguistic concepts from an ontology of the English language (OntoNote5 [145]). We use the pretrained POS tagging model introduced in Honnibal et al [75] because it has high performance on the POS tagging task (>97% accuracy score). It is also part of a popular language modeling library Spacy [75]. Second, we group the extracted linguistic structures such that the static texts 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 question: "Does the static text from the examined linguistic group contain minimal information required to comprehend the described event?". This question hints at our intuition that the quality and self-sustained static text has a minimal linguistic structure aligned with expert intuition for a comprehensible event description.

Table 5.3: Empirical study: Linguistic quality assessment.

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 5.3 gives the top-5 frequent linguistic groups alongside representative examples. In total, we found more than 5K linguistic groups from the studied systems. As the number of obtained linguistic groups is high, to make the analysis feasible we subsampled the groups, following similar practices on studies on logging code properties [100]. Specifically, we randomly sample 361 groups based on a 95% confidence interval and a 5% confidence level [168]. These values are commonly used to make the large input feasible for manual analysis [100]. Two human experts examined the samples and identified 24 linguistic groups with insufficient linguistic structure. The agreement between the two experts assessed by Cohen's Kappa score is sustainable (>0.7) [156]. 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* can be useful in representing a minimal informative description of the log instruction.

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.

5.2 QuLog: Automatic Method for Logging Code Composition Quality Assessment

Inspired by our findings in the empirical 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 a deep learning method to learn static text properties concerning the correct log level and sufficient linguistic structure. Although other design choices for the methods are possible (e.g., n-gram [158]), we considered deep learning methods, as they are demonstrated to exploit the rich textual dependencies effectively, achieving high-performance results across different tasks involving text [17]. By training the models on systems with high quality logging properties, the models 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. Following our formal definition of the problem of automatic log quality in Chapter 4, the number of quality properties is $\mathbb{N}_q = 2$. The log level assessment has $|C_1| = 5$ classes ("debug", "trace", "info", "warning", "error"), while the sufficient linguistic structure has $|C_2| = 2$ classes ("sufficient" and "insufficient").

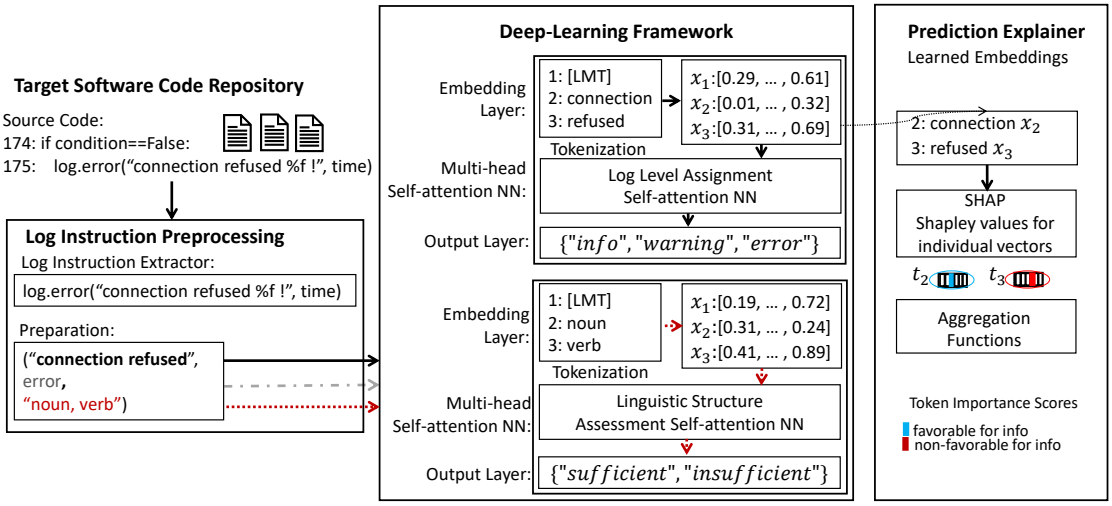


Figure 5.2: Internal architectural design of QuLog.

Figure 5.2 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 code 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. As a deep neural network method we choose the encoder of the Transformer architecture. As Transformer-based architectures are considered state-of-the-art in many text-related tasks [17], due to their ability to encode the characteristics of the textual data better, we consider them as most suitable for the problem of log instruction quality. 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 as it incorporates log instructions from different systems, 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.

5.2.1 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 instructions extractor* and (2) *log instructions preparation*, described in the following.

Log Instructions 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, and developers writing styles 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 the development process with writing correct log levels, we consider three levels ("info", "warning", and "error"). Related work reports that the three log levels are practically useful when different stakeholders (other than developers) examine logs [23, 100]. For example, operators care more for the high severity levels (i.e., "info", "warning", "error") [100]. Formally, from the initial five, we analyze a total of $|C_1| = 3$ classes for the log level.

Log Instructions 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 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 texts of the log instructions, we apply Spacy [75] for preprocessing. We split the words using space. We preprocess the static text by following text preprocessing techniques, including removing all ASCII special characters and applying lowercase transformation of the words [30]. Once processed, we give the static text as input to a pre-trained POS tagging model from Spacy, as it is reported to achieve high scores on the POS

tagging task [75]. 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 log level.

5.2.2 Deep Learning Framework

Overall Architecture

QuLog has two independent neural networks to assess the two quality properties. They share the same architectural design and are composed of an embedding layer, encoder network from Transformer architecture [166] and output layer. For a 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), which we referred to as tokens. 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 output of the encoder's layer.

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, as each token receives one embedding vector. Figure 5.2 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 adjusted to preserve the log properties (e.g., frequently co-occurring words for a certain log level).

Neural Network Encoder

We model the dependencies between the tokens and the two quality properties with non-linear parametric functions represented as neural networks. As a suitable architecture, we identified the encoder of the Transformer [166] architecture. [5] It provides state-of-the-art results in many NLP tasks [17]. By pointing to the similarities between the static text of logs and natural language [68], 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. 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, as a common learning algorithm for neural networks [103]. 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 for 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.

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 structure 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 a class (i.e., a number between 0-1 indicating class relevance). 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.

5.2.3 Prediction Explainer

The prediction explainer is used to explain the prediction of the models. The main idea is that by explaining the decisions to the developers, the most relevant tokens that contribute to the decision can be examined in the potential of misalignments between the model prediction and the written log level. The prediction explainer uses SHAP [114]

[5] The details of the architecture are discussed in Chapter 2.

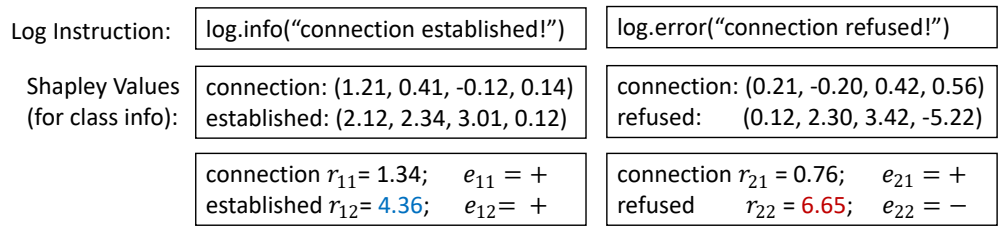


Figure 5.3: Prediction explainer working procedure example.

(Shapley additive explanations) – an approach from explainable artificial intelligence, to provide recommendations for improvement. We selected SHAP as one of the most widely popular tools for an explanation of a decision of a machine learning model due to its capability to provide good explanations for a wide range of problems [18].

SHAP - Shapley Additive Explanations; SHAP calculates feature importance scores (how relevant is a feature for the prediction) by defining the problem as a coalitional game between the features [114]. The goal is to find the 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 the feature of the prediction. The signs of the Shapley values show the feature's favorableness (or lack of it in the case of a negative sign) for the model prediction. Therefore, each Shapley has intensity and sign score.

Implementation; We used the original SHAP implementation with the default values for its parameters [113]. One required parameter of SHAP is a differentiable learning model (a model with gradients calculated for each network layer). To apply SHAP, we used a distilled trained encoder network as input [73]. While the explanation procedure is applicable for both quality properties (log level and linguistic quality), we implemented just the log level prediction explainer because of the intuitive meaning of the importance scores concerning the predictions.

Token Importance Scores; The Shapley values are calculated for each neuron of the input vector embeddings. Figure 5.3 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 a vector of Shapley values (with size $d = 4$ in the example). 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 prediction). 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. 5.1 and Eq. 5.2.

$$r(t_{ji}) = ||S_{ji}(t_{ji})||_2^2 \tag{5.1}$$

$$e(t_{ji}) = sign(\max_k |S_{jik}(t_{ji})|) \tag{5.2}$$

where $r(t_{ji})$ is the token's t_i importance score intensity, $e(t_{ji})$ is the token importance sign for the log instruction l_j. S_{jik} 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 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, alongside the predictions, QuLog automatically give suggestions for improving quality on a word level. The output of the explanation module is an ordered list of tokens, ordered by their intensity (from highest to lowest).

5.3 Evaluation

We start the description of the evaluation by first explaining the data collection procedure. Afterwards, we give the evaluation of the three parts of QuLog. First, we evaluate the log level assessment. Second, we evaluate the sufficient linguistic quality assessment part. Finally, we give the evaluation of the prediction explainer.

Code Repositories Collection; Alongside the studied software systems (that form the initial quality knowledge base), we collected log instructions from other systems from GitHub to construct a diverse system-agnostic log instruction dataset. To collect this data, we crawled GitHub and searched for systems from the following topics: Java, Python, Angular, Ruby, and PHP, selecting 7039 source code repositories. Additionally, we collected the number of GitHub stars for each system. Similar to previous works [68],

we consider the number of stars as an indicator of the quality of logging and include code repositories with more than 100 stars to build the extended quality knowledge base. Notably, as this data covers different programming languages and different systems we can apply QuLog in a system-agnostic manner.

5.3.1 Log Level Assessment

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 competing methods. Second, we evaluate QuLog's performance on a few instances of the problem of log level assignment. The details of the second experiment are given in Appendix B. The motivation for this evaluation type on one side is to examine the performance of QuLog against competing methods, and on the other side, to identify problem instances where the lower performance of data-driven approaches will not overwhelm developers with many incorrect predictions. The latter is relevant for QuLog's practical usability.

Comparison Against Competing Methods

Experiment Design; In the evaluation of QuLog against competing methods, we considered two QuLog models using the two different quality knowledge bases. The models have the same architectural design but differ in the input data used to train them. The first model, we referred to as QuLog-8 [6], is trained on data from eight software systems listed in Table 5.1. The remaining system is used for evaluation. The procedure is repeated for each system in a leave-one-out system manner. Since these systems are characterized by good logging practices, we consider that the majority of the log levels are correctly assigned, similar assumptions are made in previous studies [100]. 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 GitHub systems from the extended knowledge base. While QuLog* does not account very rigorously for the instruction quality, 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 5.1, such that the instructions from the evaluation dataset are never seen during the training of the model.

We compare QuLog against two competing methods: DeepLV [100] and Support Vector Machines (SVM) [34]. DeepLV trains bi-LSTM – a deep-learning architecture as ordinal regression, on the logging code-related data. DeepLV outperforms other log assignment

[6]The 8th in QuLog-8 refers to the number of systems used to train a model.

methods, therefore we do not evaluate against other works [100]. In addition, we consider SVM as a popular multi-class classification method trained on the vector representation of the static text from general-purpose language models [121] previously used for log level assignment [59]. The hyperparameters of the competing methods are set to the recommended values by the authors. QuLog-8, DeepLV and SVM were trained in a leave-one-system-out manner. As evaluation criteria, we used accuracy and auc following related works [100].

Regarding the considered hyper-parameters for QuLog's log level method, we considered the following ranges: *model size* $\{16, 32, 64, 128\}$, *layers number* $\{2, 4, 6\}$, and *heads number* $\{2, 4, 6, 8\}$. The maximal number of tokens to $max\_len = 16$ as most of the log instructions have fewer than 16 words. As optimizer we used Adam [87] with learning rate 10^{-4} and hyperparameters $\beta_1 = 0.8, \beta_2 = 0.95$ as frequently used in training encoder architectures with Transformers [40]. The experiments were conducted on a Linux server with Intel Xeon(R) 2.40GHz CPU running with Python 3.6 and PyTorch 1.5.0.

Results and discussion; Table 5.4 gives the results of the evaluation of QuLog against competing methods. We first compare the QuLog-8 model against the two competing methods. To evaluate the correctness of the log level assignments, we discussed the results on accuracy. On average the better-performing method is QuLog-8. It has an average 0.62 accuracy, outperforming DeepLV by 0.03 and SVM by 0.06. Comparing QuLog-8 against DeepLV it is seen that it is outperforming it in 7/9 systems while failing to do so in 1/9 systems (Elasticsearch), and ties in 1/9 (Zookeeper). As QuLog-8 uses the encoder of a Transformer, it directly learns log-language-specific embeddings in comparison to DeepLV which adapts to these specific harder, experiencing a drop in performance. This can be attributed to the greater efficiency of the Transformer to learn the specifics of the log level language. DeepLV outperforms QuLog-8 just on Elasticsearch. Elasticsearch has 50% of the logs on level "warning", while for the other datasets, the "info" is predominantly the most common class. As across the training data, the most common class is the "info", QuLog-8 learns the properties of this class better, which explains the better performance over DeepLV for the majority of the datasets. Comparing QuLog-8 against SVM shows that QuLog performs better on all the datasets. QuLog learns the specific language characteristics of the words used in the logging that appear in different contexts associated with the relevant level. SVM on the other side uses general language which is trained on large language corpora from general literature where the different context of the words appears (e.g., the word *bank* used in different contexts of memory and financial banks). As QuLog-8 is a trained log-specific language it can generalise better in comparison to SVM.

Next, we compare QuLog-8 against QuLog*. The results on accuracy show that QuLog* outperforms QuLog-8 by 0.04 on average. These results indicate the existence of shared

Table 5.4: Log level quality assessment evaluation on accuracy.

	Accuracy			
Systems	QuLog-8	DeepLV	SVM	QuLog*
Cassandra	0.66	0.63	0.64	0.68
Elasticsearch	0.55	0.60	0.52	0.60
Flink	0.63	0.59	0.58	0.69
HBase	0.65	0.58	0.64	0.67
JMeter	0.61	0.56	0.52	0.61
Kafka	0.60	0.59	0.46	0.68
Karaf	0.63	0.58	0.55	0.60
Wicket	0.67	0.63	0.56	0.62
Zookeeper	0.59	0.59	0.54	0.62
Average	0.62	0.59	0.56	0.64

system-agnostic properties of the static text and the log levels (within the extended knowledge base), independent of the software systems examined in the empirical study. The instructions originate from different programming languages and publicly accessible software systems from public repositories, 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 outperforming over the related methods shows that this is a useful property of QuLog. Furthermore, the cross-system training of QuLog* makes it useful as a cross-system evaluator of log level assignments. The good performance across different systems and the system-agnostic training of QuLog* suggest that QuLog* is more suitable for an automatic assessment of the quality of the log instructions, represented by their correct log level assignment.

In addition, we evaluate the methods on the AUC score, following related works [94]. AUC evaluates the overall goodness of the model in discriminating between the individual class pairs, e.g., how good is a model when predicting "info" instead of "warning". Table 5.5 shows the results. We compare QuLog's log level prediction models against SVM. Note that we do not compare against DeepLV as we are using the sklearn library (v 0.24.1) implementation of the AUC score, which supports the AUC score for the binary, multiclass and multi-label settings, but to the best of our knowledge does not support AUC score calculation for ordinal regression. When comparing the results on QuLog-8 and SVM it can be seen that QuLog-8 outperforms SVM by margins of 0.2 on average. QuLog-8 learns the specific differences between the static texts in relation to the log levels, in contrast to SVM which uses static general language embeddings representations. The learning flexibility of QuLog-8 enables to adjustment of the specifics of the static

text with the log levels. QuLog* further uses the rich vocabulary from the different software systems to improve the AUC score by 0.02 in comparison to QuLog-8. The AUC score is evaluating the overall discriminative power of the model for all possible decisions, not for a specific one. In practice, the log level model is expected to give a single prediction, making the AUC not very informative from a practical perspective. Therefore, for practical usability of the learned models, metrics that evaluate the final model decision (e.g., accuracy, or F_1 score) are preferable.

Table 5.5: Log level quality assessment evaluation on AUC.

	AUC		
Systems	QuLog-8	SVM	QuLog*
Cassandra	0.81	0.64	0.84
Elasticsearch	0.71	0.55	0.78
Flink	0.79	0.62	0.82
HBase	0.80	0.64	0.86
JMeter	0.79	0.53	0.82
Kafka	0.76	0.51	0.80
Karaf	0.80	0.58	0.77
Wicket	0.78	0.59	0.76
Zookeeper	0.77	0.57	0.79
Average	0.78	0.58	0.80

The main difference between QuLog's log level prediction method with the related works of DeepLV and SVM is that it utilizes log instructions from many systems to learn the specifics of the static text. To do so, it uses the encoder of Transformer architecture trained on a large database of log instructions (the extended knowledge base) and learns log instruction-specific language properties with relation to the expressed logging intention (given by the log level). This feature is one difference in our work, allowing QuLog to generalise better on different systems. The leveraging of the learning model itself is also beneficial, as seen by the improvement of the accuracy over the competing methods. Another difference is a use case, where although one can consider the prediction of the log level as a recommendation for the development process, the main intent as considered here is to evaluate the overall log level correctness of the source code of a given system and present it to the entity executing the development process. This favours our approach as it is trained on many projects from the extended knowledge base. Such information can be beneficial to reduce the chance of propagating errors in the instrumentation process and improve the trustworthiness of the produced logs.

5.3.2 Linguistic Quality Assessment

Experimental Design; To evaluate the sufficiency of the linguistic structure of the static text, we used the data from the empirical study as given in Section 5.1.2. We trained QuLog on the linguistic representations from the eight systems and evaluated the remaining one. Notably, owing to the high quality of the datasets, we found log instructions with insufficient linguistics in four systems (Cassandra, HBase, Kafka and Zookeeper), for which we present the evaluation. As baselines, we considered two popular binary text classification methods, i.e., SVM and Random Forest (RF) [16], trained on the general-purpose representation of the linguistic categories [40]. We train QuLog's linguistic quality assessment part with the same values of the hyperparameters as for the log level quality assessment. As evaluation criteria, we used F_1 and specificity. F_1 evaluates the correctness of the sufficient, while specificity evaluates the correctness of the insufficient class. In unison, they allow discussing the goodness of the models on prediction of the sufficient and the insufficient class. The scores are reported, similar in the case of the log level, in a leave-one-out manner.

Results and discussion; Table 5.6 enlists the evaluation results. It is seen that all the methods achieve a high average F_1 score and the differences are negligible. This is due to the strong signal in the data that comes from the POS tag representation of the static text. Therefore, as seen by the results the classifiers that are adjoint to this representation can all perform well. QuLog's linguistic part learns separate representations for different POS tags and their combination. As seen by the results, this degree of flexibility is beneficial for marginal improvement, which may also be attributed to the given sample of data. 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 "sufficient" and "insufficient" classes. From a comprehension perspective, it encodes verbose information on the type of *failed parsing*. The richer linguistic structure is associated with a better-described and more comprehensible event. By validating the linguistic structure QuLog detects which instructions have insufficient linguistic structure and will be reported for additional examination.

The results on *specificity* are high for both QuLog and SVM while being a bit lower for RF. Since specificity evaluates the methods' performance in the correct prediction for the

insufficient class (true negative class), the results show that QuLog and the two other methods can correctly identify the instructions with an insufficient linguistic structure. Similar observations for specificity on the individual comparisons between the methods can be made. By combing these results with the high performance on F_1 (as a trade-off between incorrect and correct sufficient predictions), we conclude that the linguistically insufficient instructions can be detected, without compromising the performance on the sufficient class for the reasons discussed above. Interestingly, the representation of the static text we considered, enables more lightweight methods, e.g., Decision Trees or Logistic Regression to be applied to the task.

Table 5.6: Sufficient linguistic structure quality assessment evaluation on systems from the initial log quality database.

	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

To evaluate the applicability of QuLog's linguistic quality module on systems other than the initial quality base, we further show the results of QuLog against SVM on systems from the extended quality database. We use the same experimental setting as in the previous case with QuLog being trained on the data from the nine systems. Table 5.7 enlists the results of the three systems. Similarly, as in the previous case, the results for the three systems are also high, predominantly due to the considered representation of the data.

Table 5.7: Sufficient linguistic quality additional evaluation on systems from the extended log quality database.

	F1		Specificity	
System	QuLog	SVM	QuLog	SVM
Openwhisk	0.99	0.98	0.99	0.96
Log4j	0.98	0.99	0.98	0.98
Tomcat	0.99	0.99	0.99	0.96
Average	0.99	0.99	0.99	0.97

QuLog's sufficient linguistic quality part draws closer connections with the literature on "how-to-log" [23, 65]. Specifically, by examining the different log-related issues (including Jira issues), we observed that the presence of sufficient linguistic components has a relation to the comprehensibility of the events. We used the observation to create a dataset which identifies linguistic structures related to sufficient and insufficient understanding of an event description. QuLog's linguistic quality further has a tangential relation to the works on static text generation [42, 68, 107]. While the related works focus on how to generate static texts, QuLog focuses on evaluating the sufficient properties of the static texts of the log instructions. Therefore, a potential use case for this method is to be part of automatically evaluating the correctness of the static texts (or jointly with the log level prediction part) that have sufficient linguistic structure (or correct level) as modeled by the QuLog models. QuLog further shares similarities to the log checker introduced in Hassani et al. [64] in the part of evaluating log levels. However, QuLog further considers the linguistic properties, beyond spelling mistakes checking, as well as evaluates the quality against many systems from different programming languages as opposed to this related work which studies and evaluates two Java systems.

5.3.3 Prediction Explainer

Experiment design; To evaluate the prediction explainer, we construct a 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 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. 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 (QuLog* model trained on the two classes Info-Error, given in Appendix B), due to its high performance and the three-class log level assignment IWE (QuLog* model trained on the two classes Info-Warning-Error, given in Appendix B). As a baseline, we consider suggesting a randomly chosen token as the most relevant.

Results and discussion; Figure 5.4 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. This can be attributed to the SHAP explainer accounting for certain properties of the input data as opposed to the baseline which gives random words. The observed discrepancy between the prediction explanations of the IE and IWE models is due to the better performance of the IE model (average F_1 score of 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 have a lower error, with IW and IWE having sharp decreases, achieving 0.05 and 0.23 on error@2 correspondingly. The low value of the performance criteria shows that QuLog's log level prediction explainer explains the predictions in alignment with human intuition on which word mostly contributes towards the log level class. Therefore, the prediction explainer gives reasonable suggestions on static text updates to improve either the words in the static text or the log level and ultimately improve the logging quality.

Alongside the quantitative evaluation, we also show qualitative evaluation, i.e., examples where the QuLog's prediction explainer module performs well and poor. When running QuLog, the developer is shown a similar screen as in Figure 5.5a-5.5c. The considered static texts are denoting successful and unsuccessful events when a connection is being established ("info" and "error" levels, correspondingly). It can be seen that QuLog, recognizes the token "refused" to have a negative contribution for the class "info", compared to the word "established". If the developer has labeled this log instruction as "info", QuLog can recognize it and directly identify the token that is the most likely suspect. In contrast, Figure 5.5b-5.5d depict one case where the QuLog prediction explainer fails. While for a human may be intuitive that the static text "Able to determine the

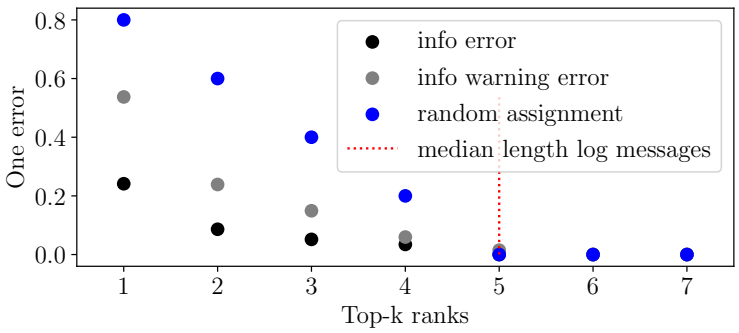
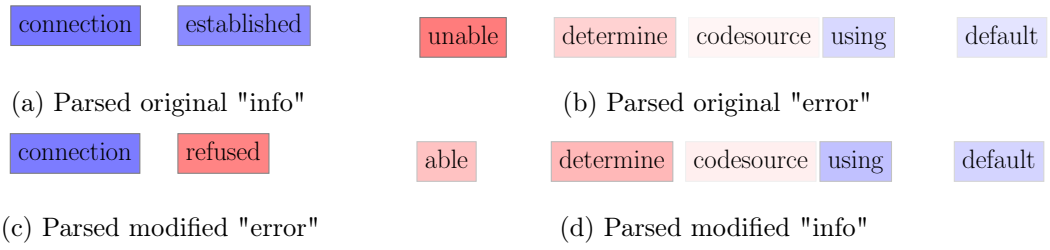


Figure 5.4: Quantiative evaluation of the explanation module.



(a) Parsed original "info"

(b) Parsed original "error"

(c) Parsed modified "error"

(d) Parsed modified "info"

Figure 5.5: Qualitative analysis of QuLog's explanation module. Blue/Red (positive/negative proclivity for "info").

code source using defaults" refers to info, the large corpora of data confuse the model to predict "error". Although the token "able" has a significant decrease in the score for the token in the first position, it does not have a token importance score of sufficient intensity to change the decision. This behaviour of the prediction explainer can be attributed to the accumulation of various correlations present in the diverse log instructions data used to train the model which correlation does not necessarily relate to human intuition or logging intention.

5.4 Chapter Summary

In this chapter, we addressed the problem of automating log quality assessment. We first did an empirical 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 instructions. Based on our observations, we proposed a deep learning-based approach for automatic log instruction quality assessment on the source code from a given target system. Our approach uses static text and its linguistic structure representation to evaluate the two properties. In addition, we adopted an approach from explainable AI to explain the model predictions and give suggestions for potential improvements of the instructions. Our approach shows high performance for both the log level and the sufficient linguistic qualities. The good performance on the log level is due to incorporating information from events and the log levels of many systems. Notably, the data used to train QuLog level prediction includes different programming languages and different systems enabling QuLog application in a system-agnostic manner. The good performance on the linguistic quality is predominantly due to the considered representation of the static text with POS tags. The suggestions from QuLog can be used by external entities to improve the logging code. Therefore, QuLog aids system development by evaluating the logging code, giving suggestions for its improvement and with that aiding the system development process.

As QuLog is system-agnostic it addresses shared properties among different systems and programming languages. As discussed in Section 5.1 the logging quality covers additional aspects where other quality properties can be analysed. Notably, they involve the nearby source code features, which may be very different among the systems because of the different programming language syntax. Due to the system-agnostic nature of the method, QuLog does not consider the contextual features where the parameters or the locations for logging (where-to-log) are being analyzed. Therefore, we do not discuss the differences between QuLog and these methods. Irrespective of this, by accounting for the quality properties of the log instructions, QuLog aims to reduce the chances that logging codes of insufficient quality will be released in production.

Chapter 6

Single Line Log-based Anomaly Detection and Classification

Contents

During the operational and development processes it is common to analyze single log lines to detect and classify anomalies [102]. As a single log line is often composed of two parts, i.e., static text and parameters, the anomalies can be reflected in both of them, independently or simultaneously [44]. The static text expresses the logging intent, i.e., the semantics of the log. The variable parameters show dynamic information for the event. For example, the log instruction log.info("VM took %f seconds to spawn.", createSeconds) from the static text perspective expresses positive intent. However, if the time for creation is larger than usual, it can indicate an anomaly. Therefore, achieving greater anomaly detection coverage is possible by considering the two parts of the log [44].

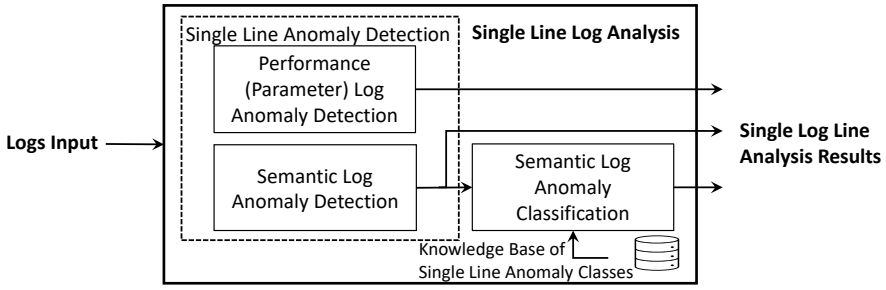


Figure 6.1: Overview of the single log line analysis.

This chapter focuses on automating anomaly detection and classification from single log lines. Figure 6.1 illustrates its overview. It is composed of 1) semantic log anomaly detection and classification and 2) parameter anomaly detection. The semantic log anomaly is concerned with the effective representation of the sentiment of the log. Log levels can be particularly useful in this context. For example, the higher log levels such as "error", are associated with *abnormal* states or state transitions, e.g., "Machine failure", useful for anomaly detection [102]. As log levels are part of the source code, we found that there exists a large set of unlabeled and unstructured anomalous data accessible from the source code of public code-sharing sites, e.g., Github. which may be useful for improving the generalization in log-based anomaly detection. In the context of parameter/performance anomaly detection of vital importance is the correct extraction of the parameters from the logs [172]. The incorrect template processing can result in missing vital parameters and potential anomalies reflected within. Therefore, good and robust log parsing is a precondition for parameter anomaly detection for different systems.

The single log line analysis is challenged by *complex data type representation*, *insufficient logging failure coverage*, *software evolution*, *labeling* and *low detection performance* of related methods. To train single line anomaly detection models, we refer to the *normality* and *detectability* assumptions. Furthermore, to design our method, we refer to the availability of the *open-source severity level data* assumption. Finally, to enable anomaly classification, we refer to the *recurrence* assumption.

We present the contribution for single log line analysis in the following [1]:

1. We found that the unlabeled public code repositories contain log-anomaly-related information that can be used as a learning signal for log-based anomaly detection.
2. We propose a novel method, named ADLILog, for semantic log-based anomaly detection that uses the target system (system subject to analysis) data, alongside the data from the public code repositories as external auxiliary data related to

[1]Parts of this chapter are published in [11, 127, 128, 129, 169].

anomalies to learn the anomaly detection model.
3. For performance anomaly detection, a prerequisite is to separate the constant and variable parts such that the parameter data can be correctly extracted. To that end, we propose a novel method for log parsing, named NuLog, to accurately extract the parameters on which performance anomaly detection models are learned.
4. The evaluation over several dimensions of the proposed methods demonstrates their applicability in single log line analysis.

This chapter is further structured as follows. In Section 6.1 we introduce the semantic-based anomaly detection method ADLILog, which uses the anomaly-related information from public repositories, alongside the target system data to learn an anomaly detection model. The single log line anomaly classification method is discussed as well. Section 6.2 introduces the performance anomaly detection approach with the novel log parsing method as one of its main components. Section 6.3 presents the experiments where we evaluate the performance of the proposed methods. Section 6.4 summarizes the chapter.

6.1 Semantic Log Analysis

The general literature on anomaly detection shows that for many different domains if even a small set of labels denoting related concepts (i.e., auxiliary labels) with the modeled phenomena exist, the performance of anomaly detection can be improved [146]. The pointed explanations for the observed improvements are that the auxiliary samples can contrast the modeled phenomena in a manner that is beneficial for learning specific representations of normal concepts [71]. In the context of log-based anomaly detection, we observed that such kind of rich auxiliary data may exist within the log instructions from the source code of public systems. Intuitively, the log levels of the log instructions show different severity levels of the contained events [102]. For example, the log level "info" is commonly associated with normal system state or state transitions. In contrast, the log levels like "error", "fatal", and "critical" are commonly used for events related to abnormal system execution. If such information indeed can be extracted from the source code, one can create rich auxiliary data with both "normal" and "abnormal" events from diverse systems, that can be used for learning an anomaly detection model. As this data incorporates information from many systems it may further enable better generalization. Guided by this intuition, we conducted a study to examine the potential of the log instructions to aid anomaly detection.

In the following, we describe the study for log instruction usage in anomaly detection. Afterwards, based on the results from the study, we propose a method that uses the log instructions from the public repositories as auxiliary data to learn an anomaly detection model in an unsupervised manner.

6.1.1 Log Instructions Usage for Anomaly Detection

We consider that we can group the log instructions, based on their log levels into two severity groups, i.e., "normal" ("info") and "abnormal" ("fatal", "critical", and "error"). Following the usage of the log levels for anomaly detection, we consider that the static texts of the instructions have complementary properties concerning the two severity level groups, i.e., they preserve anomaly-related information. To study the extent of the differences between the two groups, we analyze two language properties of the word combinations (n-grams) in the log instructions static texts concerning the two groups. N-grams are one way how to represent textual data [158]. As they are count based, they have an intuitive interpretation, which is why we select them for the analysis.

Specifically, we study the n-gram uniqueness and n-gram sentiment across the two groups. By studying the n-gram uniqueness among the groups, we examine the differences in the vocabulary used to describe normal/abnormal events. If there are considerable differences between the used words in the static text with regard to the two groups, it is expected that the two groups have different properties. By relating the n-grams with the expressed intent (e.g., positive intent relates to normal system state), we examine the semantic diversity between the groups, i.e., if the n-grams express positive (normal state transition) or negative (abnormal state transition) intents. In the following we describe the 1) *log instruction collection procedure* and then present the 2) *uniqueness* and the 3) *sentiment* analyses of the log instructions static texts.

Log Instructions Collection and Processing

For the starting point of the analysis, we created a representative log instruction dataset by collecting log instructions from the source code of many public code projects from GitHub. We included a wide spectrum of domains and programming languages (Python, Java, C++), covering different log instruction types. The heterogeneity enables us to examine the vocabulary diversity and semantic properties used in describing normal and abnormal events across systems. That way, we consider diverse logging styles and a wide range of events, with complementary severity levels. To account for the reliability of the log level assignment, we selected projects with more than 100 stars. This quality control criterion is used in other studies that analyse source codes [68]. The collection procedure resulted in more than 100.000 log instructions.

Afterwards, we process the log instructions by extracting the log levels and the static texts to represent the event descriptions and their severity levels. The diverse programming languages use different names for the log levels. Therefore, as a first step, we unify all the log levels. We preprocess the static texts by applying several preprocessing techniques, similar to related works [30, 68], including lower-case word transformation, splitting the

static texts on whitespace, removing placeholders, removing ASCII special characters and stopwords from the Spacy English dictionary [75]. We refer to this data as **Severity Level (SL)** or auxiliary data. It is a set of tuples from two elements – (1) the static text of log instruction, and (2) the severity group based on the aforenamed log level to severity group mapping (e.g., ("machine error", "abnormal")). We used the SL data to conduct the log instruction examination study. Similar to related log instruction analysis studies [68], we extracted the n-grams from the static text by varying the value for the n parameter in the range $n = \{3, 4, 5\}$. An n-gram analysis shows that many n-grams appear once. To eliminate the impact of the rare n-grams on the analysis, we considered the n-grams that appear more than three times, similar to He et al. [68].

Log Instructions Static Texts Uniqueness Analysis

Intuitively, when describing abnormal events, the static text typically contains n-grams like "failure" or "error connection", as opposed to normal events, where n-grams like "successful" and "accepted" are more likely to appear. Therefore, we consider that the log instructions static texts of the two severity level groups share different, partially overlapping vocabularies. To verify this, we adopted an approach from information theory that defines the amount of information uncertainty in a message [37]. In our case, we analyze the relation of the n-grams with the two severity groups. At first, given an n-gram (e.g., "machine failure"), there is high uncertainty for the assigned severity group. As we receive more information about the n-gram (e.g., new logs with the n-gram "machine failure"), its uncertainty concerning the associated severity group is reduced. For example, if the n-gram "machine failure" is associated five times with the "abnormal" and one time with the "normal" severity group, we have low uncertainty. In contrast, if another n-gram, e.g., "verifying connection" is associated three times with the "abnormal" and three times with the "normal" group, the n-gram uncertainty is high. To measure the uncertainty, we use Normalized Shannon entropy [63]. We calculate the entropy for each n-gram on a random sample of the SL data and reported the key statistics of the n-grams entropy distribution.

Table 6.1: Log instructions static texts uniqueness analysis results.

	Min	1st Qu.	Median	3rd Qu.	Max
Average Entropy	0.00	0.00	0.00	0.27	0.51

Table 6.1 summarizes the key properties of the n-gram entropy distribution. It is seen that the median of the distribution is 0. This means that at least half of the n-grams are associated with only one of the two severity groups. Thereby, *the two severity groups are characterized by a rather unique vocabulary.*

While this analysis gives information about the uniqueness of the vocabularies, it does not account for the type of intent expressed with the n-grams. To investigate the expressed event intent, we made an n-gram sentiment analysis (where the sentiment is used to quantify the intent type, i.e., positive or negative), given in the following.

Log Instructions Static Texts Sentiment Analysis

To evaluate the n-gram sentiment concerning the two severity groups, we considered a pretrained sentiment analysis model from Spacy [75]. We consider the applicability of the sentiment model as a good design choice by pointing to the observed similarities between general language and logs [68]. Since the sentiment model is trained on diverse language texts, it has learned notions of positive, neutral or negative intent. We run the n-grams through the model to obtain the sentiment score. We used the sentiment score to categorize the n-grams into three categories, i.e., positive, negative and neutral. We relate the events from the "normal" severity group with positive intent considering that they describe successful states or state transitions. Similarly, we relate the "abnormal" group with a negative intent considering that they describe unsuccessful system states or state transitions. The third category contains n-grams with neutral intent, i.e., events without strongly expressed intent.

Table 6.2: Log instructions static texts sentiment analysis results.

Sentiment	Positive			Negative			Neutral		
Severity Group	Normal	Abnormal	Shared	Normal	Abnormal	Shared	Normal	Abnormal	Shared
N-gram Coverage [%]	66.94%	28.13%	4.93%	23.13%	69.75%	7.12%	46.98%	43.43%	9.59%

Table 6.2 summarize the results of the n-gram sentiment analysis. For each of the three sentiment categories, we show the percentages of the n-grams concerning the two severity groups. In the positive intent category 66.94% of the n-grams are associated with the normal severity group, and 28.13% are related to the abnormal severity group. In contrast, from the n-grams associated with negative intent, 69.75% are associated with the abnormal group, 23.13% are associated with the normal severity group, and 7.12% are shared between the two. These two observations show that there exists a relationship between the normal group and positive intent, and the abnormal group and the negative intent. Therefore, the proposed severity log level grouping aligns with human intuition when expressing positive and negative sentiments. *Structuring the static text of the log instructions by their log levels in a proposed way can extract intuitively relatable anomalous information.*

Combining this observation with the uniqueness in the vocabularies between the two severity groups demonstrates that SL data has the potential to preserve rich anomaly-

related properties. Therefore, it can potentially be used as auxiliary data and serve as a foundation for anomaly detection.

6.1.2 ADLILog: Semantic Anomaly Detection with Log Instructions

Following the affirmative observations about anomaly-related information encoded in the SL data as auxiliary data, in this section, we introduce ADLILog as an unsupervised log-based anomaly detection method. Figure 6.2 illustrates the overview of the approach. Logically, it is composed of (1) *log preprocessing*, (2) *deep learning framework* and (3) *anomaly detector*. The role of the *log preprocessing* is to process the raw logs by carefully selecting preprocessing transformations that expose rich information for the deep learning framework. The *deep learning framework*'s goal is to learn and output useful log representations for the target-system logs. It does so by training a deep neural network model with a sequential two-phase learning process (pretraining and finetuning), during which data from the target-system logs and the SL data are used. The *anomaly detector* detects if the input target-system logs are normal or anomalous. In the following, we describe the three components of ADLILog.

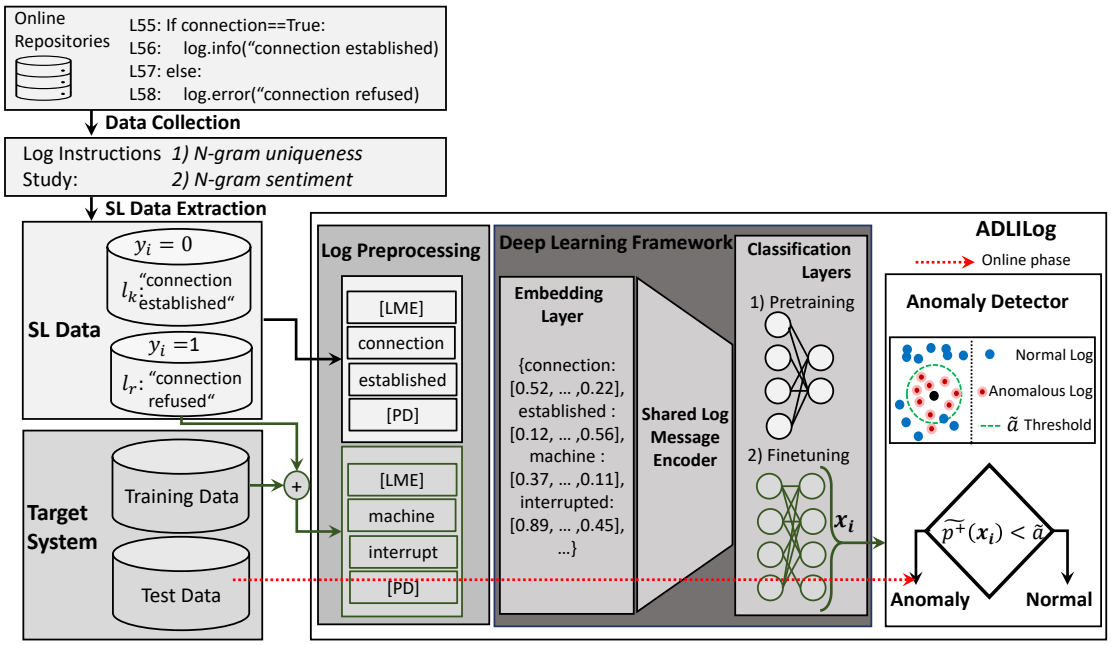


Figure 6.2: ADLILog: Detailed design of the single log line anomaly detection method.

ADLILog has two operational phases: offline and online. During the offline phase, we use a two-stage training procedure to learn the parameters of the neural network and

the anomaly detector. The two stages of the training procedure are pretraining and fine-tuning. This training procedure, on the one side, allows learning of general discriminative features between normal and anomalous events from the SL data (pretraining). On the other side, it enables the learning of good log representations of the normal target system data by combining the training with anomalous labels from the SL data used as auxiliary data (during fine-tuning). The learned models are stored. In the online phase, the test logs from the target system are given as ADLILog's input. ADLILog processes the input logs and proceeds them towards the loaded models to obtain log representations. The log representations proceed towards the anomaly detector which detects and reports the anomalous logs. In the following, we delineate ADLILog's implementation details.

Log Preprocessing

The raw target-system logs are characterized by high noise due to the parameter values generated during system runtime (e.g., IP address, endpoints, numerical parameters). The log noise can significantly affect the anomaly detection performance [187]. Therefore, log preprocessing aims to reduce the noise by applying a set of preprocessing steps. To that end, we start by removing all path endpoints (e.g., /home/spelce1/HPCCIBM/bin/) and split the static text using whitespaces into singleton items we call tokens. The tokens with numeric values most often denote variable parameters that are not relevant to the semantics of the logs. We consider them as noise and remove them. Similar to the preprocessing for the SL data, we apply Spacy and remove all ASCII special characters (e.g., $), the stopwords (e.g., `is` and `the`) [75] and transform each character into lower case, following related work [30]. Notably, as previously described, the SL data is already preprocessed by a similar set of operations making the preprocessing uniform. In addition, each log is prepended with a dedicated Log Message Embedding ([LME]) token. The [LME] token is an important design detail because we use it to extract a numerical representation of the log from the neural network, further given as input to the anomaly detector. Finally, different logs can have a variable number of tokens while the neural network requires fixed-size input. Therefore, we specify a hyperparameter max_len to unify the lengths. The shorter logs are appended with a special pad token ([PD]), while longer ones are truncated.

Deep Learning Framework

The deep learning framework consists of three components: 1) embedding layer, 2) encoder network from Transformer architecture and 3) classification layers. Given the preprocessed and tokenized logs at the input, the embedding layer transforms the input tokens into numerical vector representations, which we refer to as token embeddings. The token embeddings are numerical features represented in a suitable format for the

neural network. We then use the encoder network to learn relationships between the vector embeddings from the embedding layer and the appropriate target. The output from the encoder layer is the vector embedding of the input log/(static text), i.e., the [LME] vector. Depending on the training phase (pretraining or finetuning), the [LME] vector proceeds towards one of the two classification layers. The output from the classification layers is used as input in the appropriate loss function. After finetuning, the output from the second set of classification layers is the final vector embedding of the input log, which proceeds towards the anomaly detector.

Embedding Layer; The *embedding layer* receives the preprocessed logs as input. It serves as an interface between the textual and numerical token representation format. Specifically, each token is assigned a single index corresponding to a token embedding vector. The embeddings are learned during pretraining and are adjusted to learn the properties of the normal and abnormal events. The embeddings are learned jointly with the parameters of the neural network. There are other possible ways how to extract embeddings (e.g., pretrained language models [17], word2vec [121]). However, as they are pretrained on general language data, their values are influenced by modeling different contexts (e.g., the embeddings of the word "bank" is influenced by the financial and memory as part of the training data). In contrast, the logs are shorter texts with specific words appearing in them [68]. As we further have access to rich data, we proceeded to directly learn the embeddings on the log-specific data.

Log Message Encoder; As a suitable architecture for the log message encoder, we identified the encoder of the Transformer architecture. The encoder implements a multi-head self-attention mechanism that exploits the relations between tokens within the log instructions' static texts. This property enables learning discriminative features between the words and the different contexts they appear in (e.g., diverse vocabularies, intent). The embedding vectors and the encoder parameters are updated via the backpropagation algorithm during pretraining [103]. At the output of the encoder, we provide the vector embedding of the [LME] token. Due to the architectural design, the vector of the [LME] token attends over all the other token vectors during training. We considered this implementation architectural design detail because it allows learning the most relevant information from the input concerning normal and abnormal events. The *model size*, *number of heads* in the encoder, and the *number of encoder layers* are three hyperparameters of the log message encoder.

Classification Layers; The *classification layers* as input accept [LME] tokens from the encoder. It is composed of two sets of linear neural layers. As depicted in Figure 6.2, the first layer set (*Set 1*) has two linear layers, with parameters $\theta^{'}$. It is trained jointly with the log message encoder during the pretraining procedure. The size of the first linear layer (from the first set of linear layers) is equal to the *model size* of the encoder

layer, while the second layer (from the first set of linear layers) has two neurons that correspond to the "normal" and "abnormal" severity groups from the SL data. The output of the first set of classification layers proceeds to the binary cross-entropy loss used as a pretraining loss function.

The second set of classification layers, with parameters $\theta^{"}$, has two linear layers (*Set 2* in Figure 6.2). The two layers have the same number of neurons equal to the *model size*. The output of the second set of linear layers is given as input for the loss function during finetuning. Additionally, the output of this layer is used as the final log representation and is proceeded as input to the anomaly detector.

Learning Process. The learning process is split into two sequential phases: pretraining and finetuning. During the *pretraining* phase, we update the parameters of the embedding layer, the log message encoder and the first set of classification layers. We perform the pretraining with the SL data, using the binary cross-entropy as a commonly used loss for binary classification [56]. After pretraining, the parameters of the encoder are learned and they preserve anomaly-related information.

For the *finetuning* phase, we pair the pretrained model with the second set of linear layers. Notably, the training data in this phase consists of the target-system data and the "abnormal" severity group from the SL data as auxiliary data. Since by definition of the anomaly detection task, the majority of the target-system data is assumed to describe normal system behaviour (i.e., class 0), considering the "abnormal" class of the SL data as anomalous (i.e., class 1), the finetuning can be addressed as a binary classification problem. As our study shows, the "abnormal" class of the SL data is available, thereby, ADLILog does not need manually labeled target-system data, i.e., its unsupervised method. In the fine-tuning phase, we learn the parameters of the second set of linear layers ($\theta^{"}$). The finetuning enables learning the specifics of the target-system data while relying on the anomaly-related information from the SL data as auxiliary data. In addition, since the normal target-system data and the normal events from the SL data can differ, the finetuning adjusts the log representation embeddings to these differences.

Another important aspect of the finetuning phase is the choice of the finetuning loss. It determines the shape of the final learned log vector embeddings. Since the finetuning is defined as a binary classification problem multiple loss choices are possible (e.g., binary cross-entropy [56], or hyperspherical loss [106]). The binary-cross entropy is a formidable choice if the anomalous labels originate from target-system data because it allows learning of the exact discriminative properties between the classes [56]. However, in the case of logs, the expensive labeling process makes this assumption hard. In contrast, hyperspherical loss concentrates the normal class around a single point, e.g., the centre of the

hypersphere. At the same time, it is scattering the anomalous logs further apart. This is known as the *concentration* property [147]. The general literature on anomaly detection [146] suggests that preserving this property frequently results in improved anomaly detection performance. Consequently, the hyperspherical loss has more desirable properties for anomaly detection, and we use it as finetuning loss. Eq. 6.1 gives its definition for a single log l_i:

$$L_{ad}^{i} = (1 - y_i)||g(\mathbf{x_i}; \theta, \theta^{"})||^2 - y_i log(1 - exp(-||g(\mathbf{x_i}; \theta, \theta^{"})||^2)) \tag{6.1}$$

where $\mathbf{x_i}$ is the log representation as output from the second classification layers set, $y_i \in \{0, 1\}$ is a label for the normal target-system data and the "abnormal" SL severity class, θ and $\theta^{"}$ are parameters of the encoder and the second set of linear layers, and $g(\mathbf{x_i}; \theta, \theta^{"})$ is the function learned by the network.

Anomaly Detector

The goal of the anomaly detector is to highlight the anomalous target-system logs represented as log vector embeddings ($\mathbf{x_i}$). It has two components, i.e., 1) an assumed target-system normality function $\tilde{p}_{sin}^{+}$, and 2) an anomaly decision rule. The normality function is an assumed model of the normal target-system logs. It is a positive function, having small values for the anomalous and large values for the normal target-system logs. The form of the function depends on the type of finetuning loss. Since the chosen hyperspherical loss learns a model that places the normal logs (class 0) close to the centre of the hypersphere, the smaller distances correspond to normal system behaviour. Following the definition of the normality function, we use the reciprocal value of the Euclidean distance between the learned log representation $\mathbf{x_i}$ and the hypersphere centre (set to the origin), given by Eq. 6.2. The large distances of the vector representation from the centre of the hypersphere will result in small values for the normality score (denoting anomalies) and vice versa (as seen in Figure 6.2).

$$\tilde{p}_{sin}^{+}(\mathbf{x_i}) = \frac{1}{||\mathbf{x_i} - \mathbf{c}||^2}, \qquad \mathbf{c} = \mathbf{0} \tag{6.2}$$

Finally, to detect anomalies, we apply a decision rule on top of the normality function score values of the input logs. The decision rule involves setting a decision threshold $\tilde{a}$ over the scores, such that the logs with lower normality scores are reported as anomalous. This concludes the discussion of the semantic anomaly detection method. In the following, we describe the semantic anomaly classification.

6.1.3 Semantic Anomaly Classification

Once the anomaly is detected, we use the event template (extracted with log parsing) and compare it against the known set of templates using string matching. Under the recurrence assumption, it is assumed that the templates are occasionally examined by an operator that assigns a label for the unknown template, enriching the knowledge base of templates. As soon as a full template match is found the detected anomaly inherits the class of the matched template. Otherwise, the class $\langle UNK \rangle$ is assigned. To enable the detection of anomalies without a known class, we further consider training a multi-class classification model. To train a model we considered natural text representation of the logs obtained from pretrained language models [75]. The log representation is given as input to Random Forest as a good method for multi-class problems [165]. This model suggests a class when the matched template is unknown. As the prediction between the template matching and the model can differ, whenever the prediction between the matching and the model is different the prediction obtained by the matching is considered as correct.

6.2 Performance Log Analysis

The anomalies in the single log lines can further be reflected in the parameters. To achieve greater anomaly coverage, alongside the semantic anomaly, performance anomaly detection is important. In the following, we describe a method for performance analysis. As the parameters are intertwined with the static text, the performance analysis method highly depends on the accuracy of the correct parameter extraction, i.e., on the log parsing procedure. Therefore, the performance log-based anomaly detection method is composed of two parts, i.e., 1) log parser and 2) performance anomaly detector. Note that, as the parser extracts events from a single log line at a time, it is part of a single log analysis module. Figure 6.3 depicts the architecture of the performance anomaly detector. The extracted events can be used from other analysis components needing information on the event level. Once the logs are parsed, they proceed toward the performance anomaly detector. The performance anomaly detector detects abnormal values for the parameters and reports log messages with anomalous parameter values.

The performance anomaly detection model has two operation modes - *offline* and *online*. During the offline phase, log messages are used to tune all model parameters of the log parser and the parameters of the anomaly detector. During the online phase, every log message is passed forward through the model. This enables the log detection of parametric anomalies. In the following, we describe the two parts.

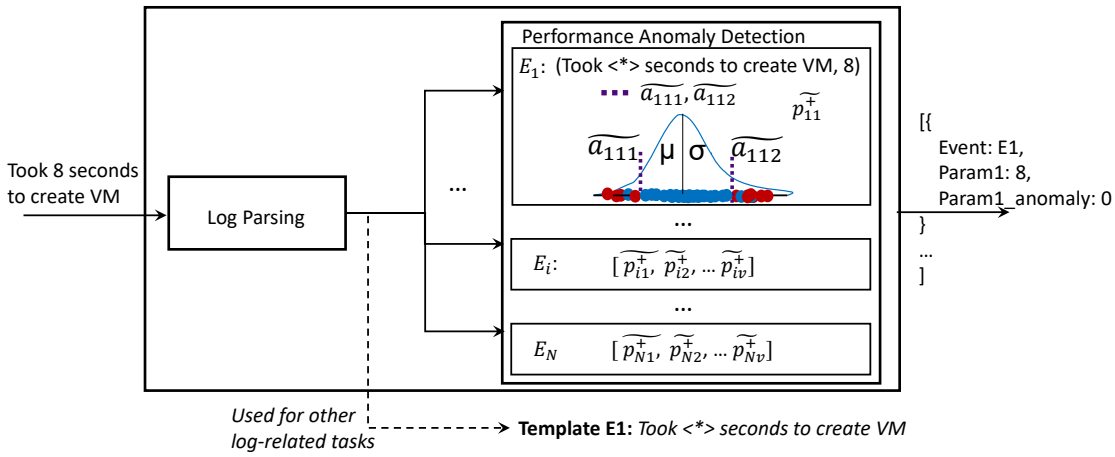


Figure 6.3: Performance anomaly detection architecture details.

6.2.1 NuLog: Self-Attentive Log Parsing

The proposed log parsing method attempts to mimic an operator's comprehension of the differences between the events and parameters from logs. Specifically, we observe that given the task of identifying all event templates in the logs, a possible approach to extract a template is to focus on constantly reappearing parts, while ignoring parts that change frequently within a certain context (e.g., per log message). The constantly repeating/changing parts are determined from nearby words/tokens that form the *context*. From a modeling perspective determining the variability degree of certain parts can be seen as calculating the conditional probability of a particular token on a given position, where the conditioning is on its context. The tokens with high probability values will constitute the static text, while the tokens with low probability are the parameters. Based on this observation we propose a parsing method, which we named NuLog.

Figure 6.4 depicts the overall architecture of NuLog. It is composed of three parts, i.e., 1) preprocessing, 2) neural network model, and 3) template extraction. The log preprocessing part prepares the generated logs for model training. The model implements a neural network architecture to learn the conditional dependencies for a given token. Once the neural network is trained, the template extraction thresholds the output probabilities to extract the event templates. In the following, we discuss the three parts in more detail.

Log Preprocessing

The log preprocessing transforms the log messages into a suitable format for the learning model. It is composed of two main parts: tokenization and masking.

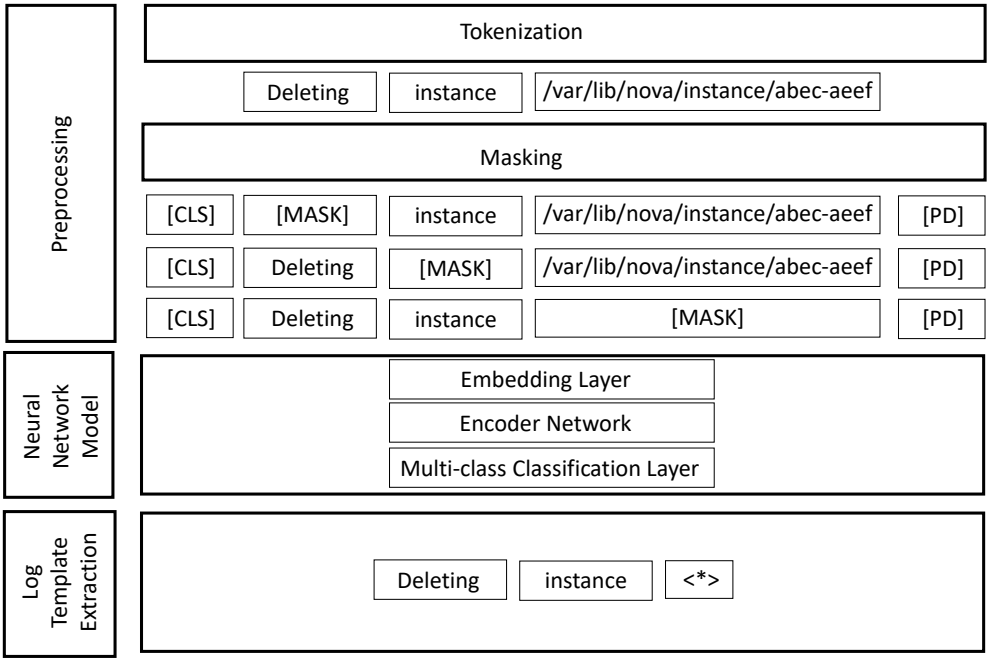


Figure 6.4: An instance of parsing a single log message.

Tokenization; The tokenization part transforms each log message into a sequence of tokens. For NuLog, we utilize a simple filter-based splitting criterion to perform a string split operation. We keep these filters short and simple to reduce the effort for log parsing utilization. Figure 6.4 illustrates the tokenization of the log message "*Deleting instance /var/lib/nova/instances/abec-aeef*". If a splitting criterion matches white spaces, then the log message is tokenized as a list of three tokens ["*Deleting*", "*instance*", "*/var/lib/nova/instances/abec-aeef*"]. In contrast to several related approaches that use additional hand-crafted regular expressions (*regex*) to parse parameters like IP addresses, numbers, and URLs, we do not parse any parameters with a regex expression[187]. Such approaches require manual adjustments in different systems and updates within the same system. In contrast, NuLog utilizes the fact that these are parameters that given a context should change.

Masking; The intuition behind the proposed parsing method is to learn a general representation of the logs by analyzing occurrences of tokens within their context. To achieve this, we apply a general task from NLP research called Masked Language Modeling (MLM). It is originally introduced in Taylor et al. [161] and successfully applied in other NLP tasks [17]. The masking module takes the output of the tokenization step as input, which is a token sequence of a log message. A percentage of tokens from the sequence is randomly chosen and replaced with the special $[MASK]$ token. If the percentage

suggests replacing two tokens with masks, the masking module will create two samples where each of the words will be masked once. The masked token sequence (the context) is used as input for the model, while the masked token acts as the prediction target. To denote the start and end of a log message, we prepend a special $[CLS]$ and apply padding with $\langle PD \rangle$ tokens. The padding ensures the training of the model as all inputs are of a unified max_len size. This is a hyperparameter of the parser.

Neural Network Model

The neural network is composed of three parts, i.e., the embedding layer, encoder network and linear output layer. The embedding and encoder layers are similar as in the case of the semantic log anomaly detection, and we do not describe them in detail. However, the output layer is different. Specifically, the output layer addresses a multi-class classification problem and predicts the most relevant token from the set of possible tokens, addressing the MLM task. The output from the encoder proceeds towards a linear layer with a softmax activation function [56]. The activation function scales the values of the output between 0 and 1, generating an ordered list for the most relevant masked token target. For a single log with a single masked token, there is an ordered token class-score list for the masked position. These scores are used by the template extraction part to decide if the masked token is a parameter or a constant part.

Log Template Extraction

The extraction of all log templates within a log dataset is executed after the model training. Therefore, we pass each log message as input and configure the masking module in a way that every token is masked consecutively, one at a time. We measure the model's ability to predict each token, and thus, decide whether the token is a constant part of the template or a variable. High confidence scores in the prediction of a specific token indicate a constant part of the template, while small confidence is interpreted as a variable. To decide if the token is a parameter or a constant part, we employ the following procedure. If the prediction of a particular token is in the top ϵ predictions and doesn't contain numbers, we consider it as a constant part of the template, otherwise, it is considered to be a variable/parameter. If the model can correctly learn the constant parts, the relevant positions are expected to always be ranked high, in the top ϵ predictions. For all variables, an indicator $\langle * \rangle$ is placed on its position within the log, forming a tuple of the log template and its parameters. ϵ is a hyperparameter of the method. The output of the log parser is the parsed log templates with a parameter list. These two proceeded towards the performance anomaly detector. In the following, we discuss the performance anomaly detector.

6.2.2 Performance Anomaly Detection

A single log may have one, several or no parameters. As a log denotes the execution of an instruction at one and only one point in time, we consider that the logs generated at different time intervals from that instruction are independent. Thereby, we consider the problem of performance anomaly detection as contextual point anomaly detection. The contextual part originates from the template of the log, while the template further gives the meaning of the parameters. Therefore, intermixing the parameters between different log instructions is discouraged. There are two important aspects of performance to consider, i.e., 1) parameter data type, and 2) a large number of events.

Regarding the parameter's data types, they can be numeric or categorical. The numeric data types are typically describing some important runtime information measuring the performance (e.g., the time needed to create a virtual machine). They come as a single predictor encouraging the adaptation of models with greater parsimony. Notably, the numeric values are directly interpretable as they are associated with dynamic variables, which further can be considered as their class. Similarly, the categorical parameters share the interpretability property. They are usually associated with indicating relationships between logs in form of IP addresses, endpoints, component names and similar. The categorical parameters are further associated with exact value meaning (e.g., HTPP error code 400), requiring expert knowledge. Thereby, we do not consider them.

An IT system can have many logs with parameters. The univariate nature of the parameters is particularly useful in this context as it enables the usage of parsimonious anomaly detection models, e.g., the σ rule [140]. In the case of numerical parameters as normality function $\tilde{p}^{+}_{ik}$ we use the Euclidean distance of the observed parameter value and the average value of the parameter in normal system state $\tilde{p}^{+}_{ik}(\mathbf{x_i}) = (\frac{1}{|\mathbb{N}_{val}|}\sum_{x_j}^{|\mathbb{N}_{val}|} t(\mathbf{x_j}) - t(\mathbf{x_i}))^2$, where t is the identity function, while $\mathbf{x_i}$ is the value of the parameter k of the log instruction i. The thresholds $\tilde{a}_{ik1/2} = \frac{1}{|\mathbb{N}_{val}|}\sum_{x_j}^{|\mathbb{N}_{val}|} t(\mathbf{x_j}) \pm r\sigma^2$ are calculated on a separate validation set for each parameter, where r denotes a confidence factor, σ^2 denotes the standard deviation of the parameter values of the normal validation set. Note that this predictor is simple to update (it involves just summation and additions of single variables) and efficient for calculation, fulfilling the requirement for an efficient model that is easy to scale to a large number of events.

6.3 Evaluation

We perform several experiments to evaluate the single log line analysis module. We split the evaluation into two parts. First, we discuss the evaluation of semantic log analysis. We compare ADLILog against two unsupervised and four supervised methods with publicly available implementation, on two real-world HPC systems. Afterwards, we

discuss the performance log analysis results with a particular focus on the log parsing evaluation.

6.3.1 Semantic Log Analysis

In this part, we present the experimental evaluation of ADLILog. The performance evaluation is performed on two public real-world data from HPC systems: BGL and SPIRIT [132]. An important characteristic they share is that the methods have novel log messages appearing through time, i.e., they evolve. This enables the evaluation of the methods with appearing novel logs – methods generalization. We set the focus on evaluating the detection performance, as the precise detection of anomalies is an important quality indicator of the method. To estimate the ADLILog's deployment complexity, we further analyze two hyperparameters of the method and the learning procedure. These experiments evaluate the practical properties of ADLILog.

Table 6.3: Datasets properties.

System	Vendor	#Processors	Days	#Messages	# Anomalies	# Anomalies (5m)
Blue Gene/L	IBM	131072	215	4713493	348460	348460
SPIRIT	DELL	1028	558	272298969	172816564	3403341

Datasets; We select BGL and SPIRIT as two real-world log data from HPC systems. They have been used for log-based anomaly detection by the research community [91, 127]. Table 6.3 shows the key properties of the datasets. As SPIRIT is a large dataset, similar to related works [91], we consider five million chronologically ordered logs. The BGL dataset is utilized fully, as it has less than five million logs. It is important to note that the two datasets are labeled on a single log message enabling the evaluation on a single log line anomaly detection. The column *#Anomalies* shows the number of labeled anomalies within the data.

To evaluate the methods' robustness, we perform the experiments on different train-test splits, similar to Nedelkoski et al. [127]. To ensure that the test data always have novel logs, we split the data chronologically on train-test splits. The train-test splits we considered are: 5%-95%, 80%-20%. These splits allow the evaluation of the performance of the method with fewer training instances and novel test logs, and vice-versa. We further show relevant statistics on the input data. Table 6.4 gives the logs distributions among the two splits. It can be seen that in each split there are many new logs in the test, that do not appear in the training dataset. This enables testing the generalization performance of the methods, i.e., the performance on unseen logs.

ADLILog Experimental Setup Implementation; We perform the experiments using three

Table 6.4: Difference between the train-test splits for the two datasets.

System	# Unique log messages in train that are not part of the test		Total Template Number
	5%	80%	
Blue Gene/L	251	97	360
SPIRIT	463	51	1088

different values for the *model size* $\{16, 64, 256\}$. The max_len parameter is set to 32 because this length covers the majority of the log lengths. To prevent overfitting, we use the dropout regularization technique with a probability rate of 0.05. In the pretraining phase, we use Adam [87] optimizer with a learning rate 10^{-4} and values for β_1 and β_2 set to 0.9 and 0.99, as commonly used optimizer for training encoder [40]. The finetuning is performed for a maximum of 20 epochs with the same values for the parameters of the optimizer. The experiments are conducted on a machine using Ubuntu 18.04, with CPU Intel(R) i5-9600K, RAM 128 GB, and GPU RTX 2080.

Experimental Results and Discussion

Comparison against unsupervised methods; We evaluate the anomaly detection performance in two separate experiments, against (1) unsupervised and (2) supervised methods, to examine the potential strengths and shortcomings of our method with respect to the two method groups. As evaluation criteria, we considered F_1, precision and recall as commonly used to access the performance of the anomaly detectors [91].

Experimental Setup; From the unsupervised methods, we considered DeepLog [44], and PCA [172]. We found these methods to be competitors of ADLILog as our method is unsupervised from the perspective of the target system. We set the hyperparameters of the methods using their defaults from their respective implementations, enabling equal methods comparisons. Recent study by Van-Hoang et al. [91] identifies LogAnomaly [117] to outperform DeepLog by margins of 0.03-0.05 on the F_1 score. As we are not aware of any public implementation of it, and we do not consider it in our evaluation. We considered the publicly available implementation of the two methods from the deep-logalizer[2] and logalizer[3] libraries.

Results and Discussions; Figure 6.5 shows the results of ADLILog compared against the unsupervised baselines. Overall, ADLILog achieves the best-performing results in three out of the four tested cases. Specifically, it outperforms the methods on F_1 score for

[2]https://github.com/logpai/deep-loglizer
[3]https://github.com/logpai/loglizer

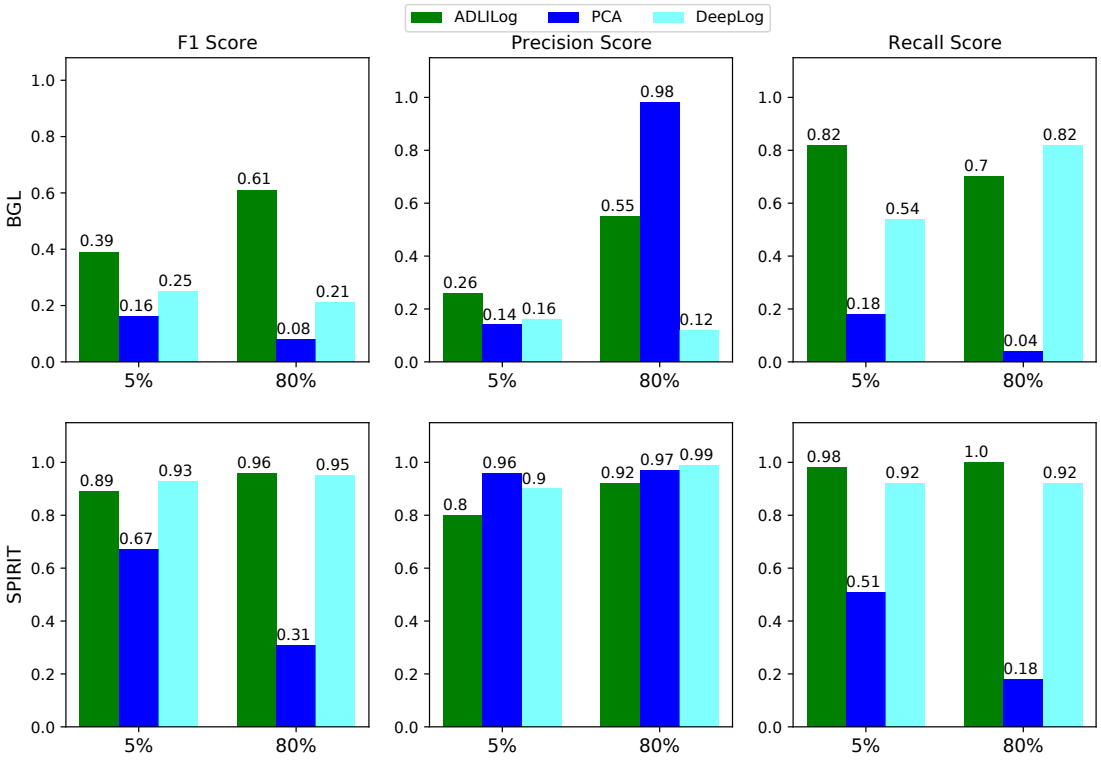


Figure 6.5: Comparison of ADLILog against unsupervised methods.

the two splits in BGL, and by a slight margin of 0.01 on the 80% split for SPIRIT. ADLILog has a strong performance on recall, for both of the datasets. This means that the method can correctly detect the true anomalies. This is attributed to the shared properties of the "abnormal" class of the SL data as auxiliary data and the anomalous target system class. As the SL data contain rich semantics of anomalous events, it helps ADLILog to make more correct predictions on the novel logs. This is particularly emphasised for the two splits of BGL. As can be seen from Table 6.4, this dataset has a significant number of novel events in the test set, dozens of which have words like "failed", "interrupted", "error" and similar. By leveraging the "abnormal" class of the SL data (which shares similar vocabulary), ADLILog can exploit the shared properties between the target system anomalies and the "abnormal" class. Therefore, the SL is useful as auxiliary data in improving anomaly detection performance. It enables learning of representations of the target-system logs by emphasising the differences between the normal and anomalous logs, which ultimately helps anomaly detection.

DeepLog, in most cases, has a stronger performance on recall, as it learns the normal state well. In some cases, as for BGL, leveraging just the template indices may not be

sufficient, as the precision drops. In the cases where novel normal log events are asked to be predicted, the model confuses them with an anomalous input, which increases the false positive rate affecting the precision. As ADLILog solely utilizes individual logs, it aims to learn the distinctive features of what constitutes a normal sample from the target system, by relying on the SL data as auxiliary data. The PCA method tends to have a better performance on precision, as opposed to recall. Notably, PCA has a drop in performance for both of the datasets on the 80% split (when more training data is available). As PCA aims to represent the normal state by relying on the template count, when many normal samples are available (increase variance in the data), the produced representation space will have an increased number of components. In this case, PCA faces the problem of the sparsity of the high dimensional spaces where the normal and anomalous points are both further apart. This results in many anomalous points being detected as normal (dropping the recall). As ADLILog and DeepLog learn the data characteristics, they better exploit the availability of the input data, and in general, do not have reduced performance when more data is available.

Comparison against supervised methods; As observed from the statistics of the data, there exist many labeled logs. As supervised methods are commonly shown to be

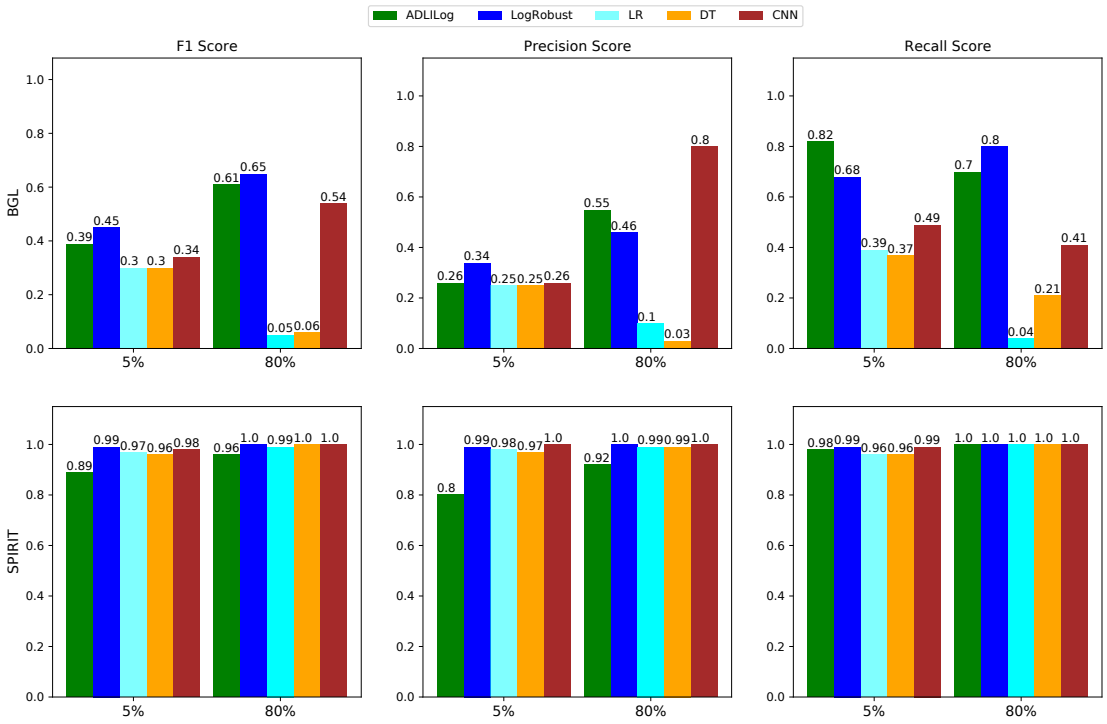


Figure 6.6: Comparison of ADLILog against supervised methods.

best performing for log-based anomaly detection, we evaluate ADLILog against them [91]. As the SL data used by ADLILog shares similar, but not the same properties with the ground truth labels, this comparison further enables us to glean more insights into the strengths and limitations of using the SL data as auxiliary data.

Experimental Setup; We compare ADLILog with four supervised methods. We considered two deep learning-based methods (LogRobust [180], and CNN [112]), and two traditional based approaches Decision Tree (DT) [28] and Logistic Regression (LR) [6]. LogRobust is currently regarded as a state-of-the-art method, according to a recent survey [91]. Therefore, we do not compare ADLILog against any further methods from the literature. We consider a similar evaluation scenario as in the unsupervised case.

Results and Discussions; Figure 6.6 shows the results from the comparison. On average the best performing method is LogRobust with an average F_1 score of 0.55 on BGL, as compared to 0.5 as obtained by ADLILog. On SPIRIT, ADLILog is outperformed by both CNN and LogRobust on the F_1 score. They both use the correct labels from the target system. The existence of discrepancy can be attributed to the SL data that does not model the exact vocabulary of the target system, i.e., the SL data as auxiliary data does not preserve the correct target distribution, but similar related properties instead. A further implication of this can be seen in the lack of the best-performing scores on the SPIRIT dataset on the 80% split. While all the methods achieve the best score, ADLILog suffers a drop by 0.04 on the F_1 score. Nevertheless, it is important to note that ADLILog in the case of BGL can use a small number of samples and can provide slightly better scores than CNN. In comparison to the supervised methods, the strongest aspect of ADLILog is the good performance on recall, where it outperforms all the methods on the 5% BGL split, compares second on the 80% BGL split, and is comparable for the two splits on SPIRIT. In addition, when considering the BGL dataset, ADLILog outperforms the traditional supervised methods. As these methods use fixed vectors as representation, ADLILog benefits from the finetuning process to adjust its rich vocabulary to the vocabulary of the logs in the target system. Therefore, it can perform better than the traditional methods. The deep learning-based anomaly detection methods outperform the traditional ones due to learning the feature vectors on the data. In general, as more samples for training become available, the performance of the methods on F_1 and precision improves. As the traditional supervised methods on the 80% BGL split are trained on many data, they overfit and do not generalize well, as seen by the drop in precision and recall. The good performance of the 80% SPIRIT split for all the methods can be attributed to the low number of novel classes in the test split.

While in general ADLILog is outperformed by the deep learning-based supervised methods, it still performs comparably to them. This is particularly important as ADLILog does not need labeled data, as opposed to supervised methods. Due to the fast rate of

system evolution and the depreciation of labels for logs, which come in large volumes, the availability of the SL data as auxiliary data is important as it allows an unsupervised method training with lower, but comparable performance to the supervised methods. Therefore, ADLILog has a competing edge over the supervised methods from this perspective. Furthermore, ADLILog is out-competing the unsupervised methods, addressing the challenge of the *low detection performance* of the competing methods. Therefore ADLILog generalizes better. The utilization of the SL data obtained from public code repositories is one unique property of the proposed approach in comparison to other existing methods. In contrast to the related deep-learning methods that adopt word vector representations for the input (e.g., LogRobust), ADLILog leverages publicly accessible data from public repositories to learn log-specific properties of normal and abnormal events among the different computer systems. The SL data utilization as auxiliary data is specific for two aspects, 1) it enables learning of log-specific model to discern normal and abnormal system events (after pretraining) and 2) enables the discrimination of the anomalous data from the target system enabling anomaly detection (after fine-tuning). Notably, the SL data is easily and publicly available, making its utilization practically possible. Similar to other domains, as discussed in Ruff et al. [146], the results show that the auxiliary data is beneficial for log-based anomaly detection.

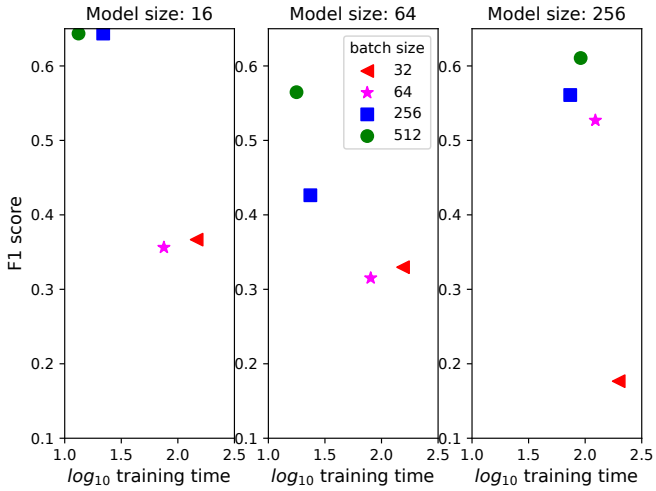


Figure 6.7: Sensitivity analysis of the influence of batch and model size over the predictive and runtime performances.

ADLILog Hyper-parameter setting evaluation; The correct hyper-parameter setting influences the needed effort for the method training, and the quality of the detection. To evaluate the impact of the hyperparameters on the detection performance and efficiency, we examined two hyperparameters, model and batch sizes. The model size is

related to memory utilization while the batch size is related to the time needed to update the model.

Experimental Setup; We considered the 80% split of the BGL dataset to study the effects of the hyperparameters. We varied the model size in the range $\{16, 64, 256\}$ and batch size in the range $\{32, 64, 256, 512\}$, at a fixed number of iterations for the pretraining phase. The F_1 score as a measure of the detection performance is reported.

Results and Discussions; Figure 6.7 shows the results. It can be seen that the larger batch size and smaller model size provide better detection performances while being faster for updating. The smaller model sizes imply smaller memory utilization. The prediction time per batch size of 512 is 17 ms (∼30000 logs per second) on the aforenamed configuration. Together with the small model size, these experiments imply that ADLILog can produce many predictions relatively fast. This is important for modern software systems that produce many logs (e.g., He et al. [70] report up to 50GB/h). Observing the time it can be seen that the times for model updates are relatively low. These parameters show that the method has good practical properties. In the following, we discuss the performance on the single log line anomaly classification task.

Semantic Anomaly Classification

Experimental Setup; To evaluate the semantic anomaly classification method, we used the BGL dataset as it has an anomaly class of individual logs. In total there are 41 classes. The top-3 most frequent classes are *KERNDTLB* which denotes *"data TLB error interrupt"* (and it is the most frequent one with (152,734/348,460) log messages), *KERNSTOR* which shows *"data storage interrupt"*, and *APPSEV* which describes *"ciod: Error reading message prefix after LOGIN MESSAGE on CioStream"*. We compare our approach with LogClass [119] implemented with TFIDF, which was trained with the default values for its parameters. We evaluate the methods of several splits from BGL. The splits are created by incrementally splitting the data into intervals of 10% between 10-90%. We used the last 20% of the split to create the test dataset. For example, if there are 100 log messages, for the 10% split we will use just the first 10 logs. Out of them, eight will be used for training the model, and the remaining two for evaluation. This evaluates the methods progressively through time as more knowledge for anomaly classes becomes available (e.g., given by an operator).

Results and Discussions; Table 6.5 shows the results. Note that both methods achieve comparable and peak performance on the task of semantic log-based anomaly classification, under the *recurrence* assumption. Specifically, as seen by the target class statistics, whenever there are no new log classes in the test set (the row unique test samples) the performance of the two methods is optimal. Once there are new log templates (e.g.,

the split 0.5), the performance drops as the models are asked to predict unseen classes. The drop in performance is the most severe for the split of 0.7, where there are 24 new classes in the test set. However, by directly matching the templates, one is ensured that the known anomalies are correctly classified. The unclassified logs are stored and can be given to operators for labeling. As seen by the next split, i.e., 0.8, all the classes are correctly detected after labels are obtained. Therefore, the single-log-line anomaly classification method augments the reported output beyond the detected anomaly if knowledge of anomalous classes is available.

6.3.2 Performance Log Analysis

We split the evaluation of the performance log analysis into two parts. We begin by discussing the evaluation setup for log parsing. Next, we present the evaluation results against twelve related log parsing methods. Finally, we describe the experimental scenario we developed alongside the experimental results for performance anomaly detection evaluation.

Log Parsing Evaluation

Experimental Setup; To quantify the performance of the proposed method, we perform an exhaustive evaluation of the log parsing task on a set of 10 benchmark datasets and compare the results with 12 other log template parsing methods. The comparison with other systems allows us to examine the robustness of the parser across logs generated from different systems. This is of particular importance for the performance anomaly detector which requires that the log parser can work well for many different systems. The datasets with the implementation of the other parsers were obtained from the log

Table 6.5: Semantic anomaly classification results on BGL.

	Split	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Target Class Statistic	Train Classes	4.0	4.0	4.0	4.0	4.0	4.0	5.0	29.0	30.0
	Test Classes	1.0	1.0	1.0	1.0	5.0	5.0	28.0	6.0	26.0
	Unique Test Classes	0.0	0.0	0.0	0.0	1.0	1.0	24.0	0.0	6.0
Ours	F1	1.0	1.0	1.0	1.0	0.6	0.6	0.06	1.0	0.69
	Precision	1.0	1.0	1.0	1.0	0.8	0.8	0.14	1.0	0.7
	Recall	1.0	1.0	1.0	1.0	0.6	0.6	0.05	1.0	0.7
LogClass	F1	1.0	1.0	1.0	1.0	0.6	0.76	0.06	1.0	0.7
	Precision	1.0	1.0	1.0	1.0	0.8	0.8	0.14	1.0	0.74
	Recall	1.0	1.0	1.0	1.0	0.6	0.73	0.05	1.0	0.71

benchmark introduced in Zhu et al. [187]. As evaluation criteria, we used parsing accuracy and edit distance.

Table 6.6: Datasets and NuLog hyperparameter settings.

System	#T	Tokenization filter	#epochs	ϵ
BGL	120	([|:|\(|\)|=|,])|(core.)|(\.2,)	3	50
Android	166	([|:|\(|\)|=|,|"|||@|$|\[|\]|||;])	5	25
OpenStack	43	([|:|\(|\)|"|||@|$|\[|\]|||;])	6	5
HDFS	14	(\s+blk\–)|(:)|(\s)	5	15
Apache	6	([ ])	5	12
HPC	46	([|=])	3	10
Windows	50	([ ])	5	95
HealthApp	75	([ ])	5	100
Mac	341	([ ])|([\w-]+\.)2,[\w-]+	10	300
Spark	36	([ ])|(\d+\sB)|(\d+\sKB)|(\d+\.)3\d+	3	50

Datasets; The log datasets employed in our experiments are summarized in Table 6.6. These real-world log data range from supercomputer logs (BGL and HPC), and distributed system logs (HDFS, OpenStack, Spark), to standalone software logs (Apache, Windows, Mac, Android). To perform the experiments, we follow the guidelines from Zhu et al. [187] and utilize a random sample of 2000 log messages from each dataset, where the ground truth templates are available.

The BGL dataset is collected by Lawrence Livermore National Labs (LLNL) from the BlueGene/L supercomputer system. HPC logs are collected from a high-performance cluster, consisting of 49 nodes with 6,152 cores. HDFS is a log data set collected from the Hadoop distributed file system deployed on a cluster of 203 nodes within the Amazon EC2 platform. OpenStack is a result of a conducted anomaly experiment within CloudLab with one control node, one network node and eight compute nodes. Spark is an aggregation of logs from the Spark system deployed within the Chinese University of Hongkong, which comprises 32 machines. The Apache HTTP server dataset consists of access and error logs from the apache web server. Windows, Mac, and Android datasets consist of logs generated from single machines using the respectively named operating system. HealthApp contains logs from an Android health application.

Results and Discussions; In the following we discuss the log parsing PA results of NuLog on the benchmark datasets, and compare them against twelve other methods. Table 6.7 presents the results. Specifically, each row contains the datasets, while the compared methods are given the columns. Additionally, the penultimate column contains the high-

est value of the first twelve columns - referred to as the best of all - and the last column contains the results for NuLog. In the bold text, we highlight the best of the methods per dataset. HDFS and Apache datasets are most frequently parsed with maximal (i.e., 100% PA). This is because HDFS and Apache error logs have relatively unambiguous event templates that are easy to identify. For them, NuLog is also able to achieve comparable results. For the Spark, BGL and Windows datasets, the existing methods already achieve high PA values above 96% (BGL) or above 99% (Spark and Windows). Our proposed method can slightly outperform those. For the rather complex log data from OpenStack, HPC and HealthApp the baseline methods achieve a PA between 78% and 90%, which NuLog outperforms by 4-13%.

Having a good parsing model is vital for the performance anomaly as it enables the correct disentanglement of the static text and the variable parameters. This enables performance anomaly detection. Therefore, the robustness of NuLog is analyzed and compared to the related methods. Figure 6.8 shows the accuracy distribution of each log parser across the log datasets within a boxplot. From left to right in the figure, the log parsers are arranged in ascending order of the median PA. That is, LogSig has the lowest, and NuLog obtains the highest parsing accuracy on the median. Although most log parsing methods achieve high PA values of 90% for specific log datasets, they have a large variance when applied across all given log types. NuLog outperforms the other baseline methods in terms of PA robustness with a median of 0.99. This makes NuLog particularly suitable for extracting the parameters from the log messages across different systems. This is important for the performance anomaly detector whose performance is directly related to the correctly extracted parameters. Notably, the extracted events can be further used by other log analysis tasks.

Figure 6.8 gives the edit distance scores. The table structure follows one of the PA results. In bold, we highlight the best edit distance value across all tested methods per dataset. It can be seen that in terms of edit distance, NuLog outperforms existing methods on the HDFS, Windows, Android, HealthApp and Mac datasets. It performs

Table 6.7: Comparisons of log parsers on parsing accuracy.

Dataset	SLCT	AEL	LKE	LFA	LogSig	SHISHO	LogCluster	LenMa	LogMine	Spell	Drain	MoLFI	BoA	NuLog
HDFS	0.545	0.998	1.000	0.885	0.850	0.998	0.546	0.998	0.851	1.000	0.998	0.998	**1.000**	0.998
Spark	0.685	0.905	0.634	**0.994**	0.544	0.906	0.799	0.884	0.576	0.905	0.920	0.418	0.994	**1.000**
OpenStack	**0.867**	0.758	0.787	0.200	0.200	0.722	0.696	0.743	0.743	0.764	0.733	0.213	0.867	**0.990**
BGL	0.573	0.758	0.128	0.854	0.227	0.711	0.835	0.690	0.723	0.787	**0.963**	0.960	0.963	**0.980**
HPC	0.839	**0.903**	0.574	0.817	0.354	0.325	0.788	0.830	0.784	0.654	0.887	0.824	0.903	**0.945**
Windows	0.697	0.690	0.990	0.588	0.689	0.701	0.713	0.566	0.993	0.989	**0.997**	0.406	0.997	**0.998**
Mac	0.558	0.764	0.369	0.599	0.478	0.595	0.604	0.698	**0.872**	0.757	0.787	0.636	**0.872**	0.821
Android	0.882	0.682	0.909	0.616	0.548	0.585	0.798	0.880	0.504	**0.919**	0.911	0.788	**0.919**	0.827
HealthApp	0.331	0.568	0.592	0.549	0.235	0.397	0.531	0.174	0.684	0.639	**0.780**	0.440	0.780	**0.875**
Apache	0.731	1.000	1.000	1.000	1.000	1.000	0.709	1.000	1.000	1.000	1.000	1.000	**1.000**	**1.000**

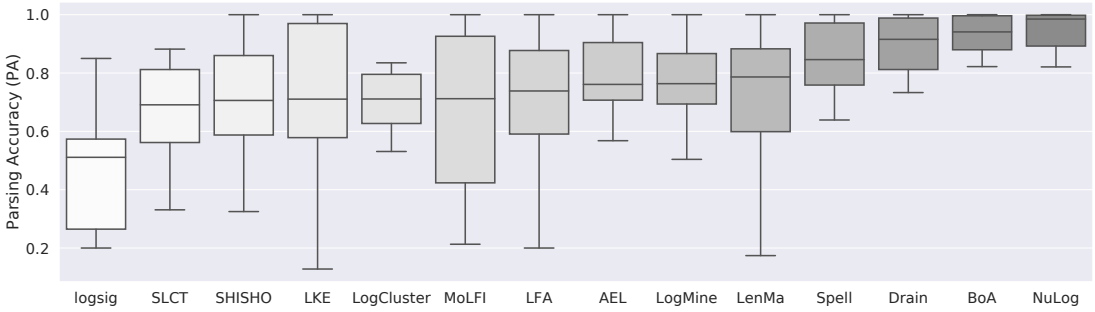


Figure 6.8: Robustness evaluation on the parsing accuracy.

comparably on the BGL, HPC, Apache and OpenStack datasets and achieves a higher edit distance on the Spark log data.

We further want to verify how consistent NuLog is performing in terms of edit distance across the different log datasets. Figure 6.9 shows a box plot that indicates the edit distance distribution of each log parser for all log datasets. From left to right in the figure, the log parsing methods are arranged in descending order of the median edit distance. Although most log parsing methods achieve minimal edit distance scores under 10, most of them have a large variance over different datasets and are therefore not generally applicable for diverse log data types. MoLFI has the highest median edit distance, while Spell and Drain perform constantly well - i.e. small median edit distance values - for multiple datasets.

Table 6.8: Comparisons of log parsers on edit distance.

Dataset	LogSig	LKE	MoLFI	SLCT	LFA	LogCluster	SHISHO	LogMine	LenMa	Spell	AEL	Drain	BoA	NuLog
HDFS	19.1595	17.9405	19.8430	13.6410	30.8190	28.3405	10.1145	16.2495	10.7620	9.2740	8.8200	8.8195	8.8195	**3.2040**
Spark	13.0615	41.9175	14.1880	6.0275	9.1785	17.0820	7.9100	16.0040	10.9450	6.1290	3.8610	**3.5325**	3.5325	12.0800
BGL	11.5420	12.5820	10.9250	9.8410	12.5240	12.9550	8.6305	19.2710	8.3730	7.9005	5.0140	**4.9295**	4.9295	5.5230
HPC	4.4475	7.6490	3.8710	2.6250	3.1825	3.5795	7.8535	3.2185	2.9055	5.1290	**1.4050**	2.0155	1.4050	2.9595
Windows	7.6645	11.8335	14.1630	7.0065	10.2385	6.9670	5.6245	6.9190	20.6615	4.4055	11.9750	6.1720	5.6245	**4.4860**
Android	16.9295	12.3505	39.2700	3.7580	9.9980	16.4175	10.1505	22.5325	3.2555	8.6680	6.6550	3.2210	3.2210	**1.1905**
HealthApp	17.1120	14.6675	21.6485	16.2365	20.2740	16.8455	24.4310	19.5045	16.5390	8.5345	19.0870	18.4965	14.6675	**6.2075**
Apache	14.4420	14.7115	18.4410	11.0260	10.3675	16.2765	12.4405	10.2655	13.5520	10.2335	10.2175	**10.2175**	10.2175	11.6915
OpenStack	21.8810	29.1730	67.8850	20.9855	28.1385	31.4860	18.5820	23.9795	18.5350	27.9840	**17.1425**	28.3855	17.1425	21.2605
Mac	27.9230	79.6790	28.7160	34.5600	41.8040	21.3275	19.8105	17.0620	19.9835	22.5930	19.5340	19.8815	17.062	**2.8920**

Performance Anomaly Detection Evaluation

Experimental Design; In the following, we present the experimental design for performance anomaly detection. The detection of performance anomalies requires knowledge about the system itself, e.g., the meaning of the parameters and the context anomalies

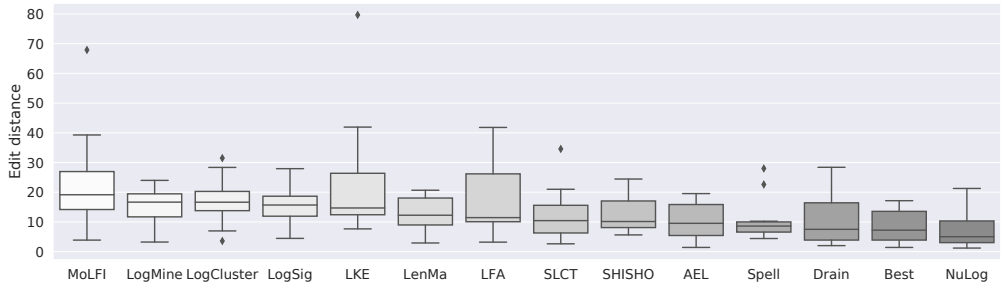
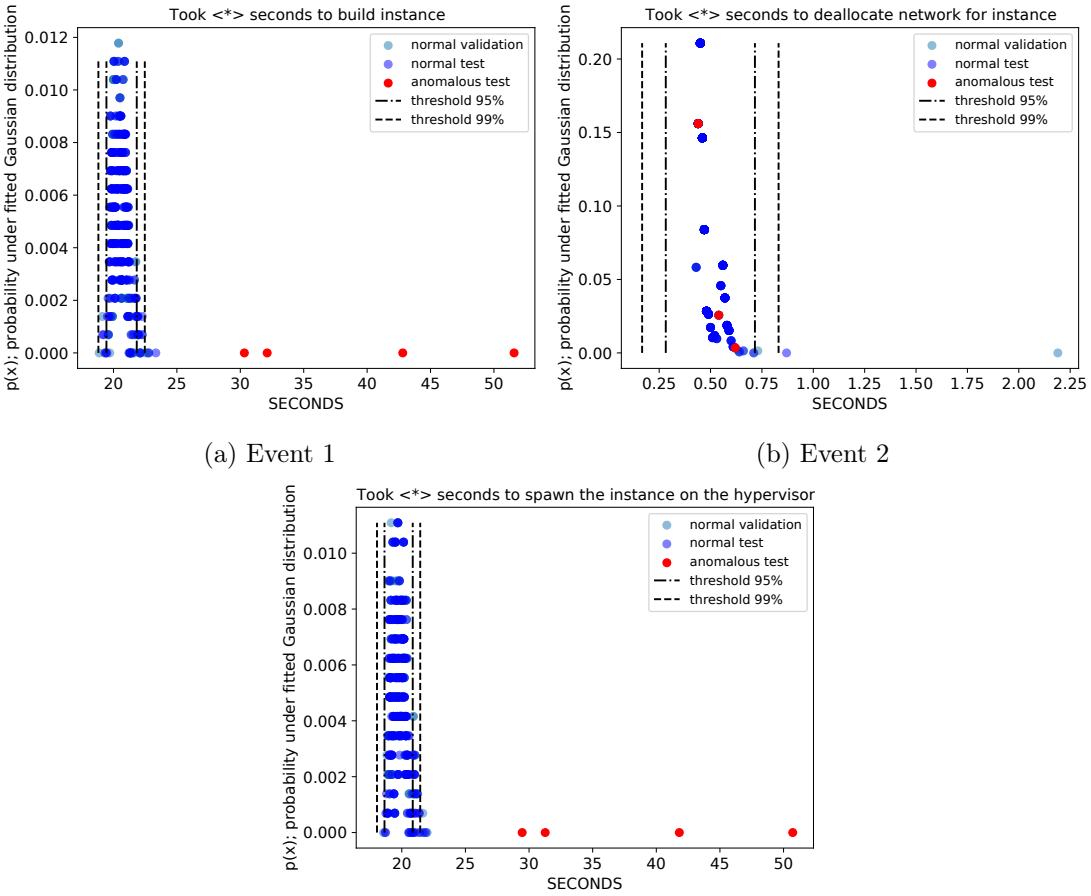


Figure 6.9: Robustness evaluation on the edit distance.

appear in. This does not limit the general applicability of the approach but enables the evaluation. Du et al. [44] propose a dataset and a procedure for generating performance anomalies we adopt. Specifically, the introduced dataset simulates how multiple users constantly request the creation and deletion of VMs. To generate performance anomalies, the authors propose throttling the network between the control and compute nodes. As copying the image from the control node to the compute node is an important operation during VM creation, the network throttling increases the time for executing the operation but does not report sequential or single log line anomalies. The events related to this parameter can be used for performance anomaly detection. We restrict to a single use case for the evaluation as we are not aware of other public resources describing the generation procedure for performance anomalies without introducing anomalies with additional properties, e.g., semantic or sequential.

To build models we considered the procedure described in Section 6.2.2. We applied first the log parser to extract the log templates. The procedure resulted in a total of 54 events. A total of 42 templates have categorical information in the name of the VM the event is associated with. The remaining templates are characterized by both categorical and numeric data types. As the network throttling slows down the internet connection, the time needed to copy the image is increased. Therefore, from the numerical data types, the parameters related to the word "seconds" are considered the most relevant and we analyze them.

Results and Discussions; Figure 6.10 summarizes the results from three different events. It can be seen that the majority of the normal data fall within the 95% and 99% confidence intervals obtained for $r = \{1.96, 3\}$ correspondingly given by the blue dots. Figure 6.10a depicts the distribution of the normal validation and test, and anomalous test times for the parameters "seconds" of the event "Took <*> seconds to build instance". The red dots denote the VMs with a creation time larger than 30 seconds when the anomaly was injected. From the semantic perspective, the event denotes a normal system state.



(a) Event 1

(b) Event 2

(c) Event 3

Figure 6.10: Performance anomaly detection results.

However, the large time to build the instance suggests the existence of some problem. The exploitation of the parameter enables the detection of the anomaly. If just semantics would have been used, this anomaly would have been probably missed as denotes a normal system event. Figure 6.10c gives the event "Took <*> seconds to spawn the instance on the hypervisor." where a similar observation can be made. However, in the case of the second event "Took <*> seconds to deallocate network for instance." (Figure 6.10b) the time is not affected as it does not require the involvement of the communication between the control and compute node. While all three events denote a normal state from a semantic perspective, utilizing the parameters of the log events enables richer visibility into system behaviour and detects anomalies that would have been otherwise missed. Therefore, the successful detection of anomalies in single log lines should encompass both, the parameters and the semantics of the logs.

6.4 Chapter Summary

This chapter addresses the task of automating single log analysis. As the anomalies affect the single log lines differently, we split the analysis into 1) semantic log-based anomaly detection and classification and 2) performance anomaly detection. To detect semantic anomalies, we introduced a novel unsupervised method for log anomaly detection, named ADLILog. The key idea of ADLILog is to use large unstructured information from the logging instructions of public code repositories. ADLILog uses this information as auxiliary data to improve the target-system log representations, which directly improves anomaly detection. We first conducted a study to examine the language properties of the log instructions, and we showed that they encode rich anomaly-related information. ADLILog combines the anomaly-related information as auxiliary data and the target-system data to learn a deep neural network model by a sequential two-phase learning procedure. The extensive experimental results on two real-world datasets from HPC systems showed that ADLILog outperforms the unsupervised, and has a competitive detection performance with the supervised methods. Furthermore, ADLILog has better generalization performance. By leveraging the *recurrence* assumption we further considered the classification of single log line anomalies.

For a complete single log line analysis we further detect the anomalies within log message variables. One challenge with respect to this is to correctly disentangle the static text from the variable parameters, and extract data for performance anomaly detection. To that end, we proposed a novel log parsing method. We observed that having words appearing at a constant position of a log record implies that their correct prediction can be directly used to produce a log message template. An incorrect prediction indicates that a token is a parameter. Based on this observation, we proposed NuLog as a log parsing method that uses deep learning to learn the constant and variable parts of the log instructions. The evaluation results on ten public benchmark datasets from diverse systems showed that NuLog achieves high robustness. This is important because it enables the usage of NuLog as a log parsing method across many different systems, and correctly extracts the parameters important for performance anomaly detection. Furthermore, by adopting a parsimonious model for anomaly detection on the parameters extracted with the log parsing procedure, we showed that we can also detect performance anomalies. Through a use case, we showed that the analysis of the two categories (semantic and performance) in unison enables greater anomaly detection coverage. Ultimately, the proposed methods contribute to the improvement of the operational and development processes.

The single log analysis is particularly useful in cases where the semantics and variables are of primary concern. As long as the anomaly is expressed within a single log, these

methods can give a hint and narrow down the potentially relevant set of logs. That way the method is complementary to the existing log levels within the source code, as it put the accent on the semantics of the logs. Due to the possibility of misjudging the log level of the events [100], the semantic anomaly detector as it is trained on many diverse normal and anomalous system events can be particularly useful when large log volumes need to be analyzed. A key feature of the presented semantic anomaly detection approach is the existence of publicly available data to extract the log instructions to learn general features of anomalous and normal events among different systems. The presented achievements indicate that reusing publicly available data can be effective in the automation of IT operation tasks such as log-based anomaly detection.

Chapter 7

Sequential Log-Based Anomaly Detection and Classification

Contents

Although the single log line anomaly detection and classification methods can achieve high performance, due to the *insufficient anomaly coverage* in the source code some anomalies may not be explicitly logged [102]. As a consequence, the single log analysis cannot deal with those cases. Nevertheless, anomalies can manifest in logs as changes in the log sequences, prompting the need for sequential analysis [44].

The focus of this chapter is the problem of sequential log-based analysis. When considering log sequences, the challenge of how to efficiently represent log sequences as *complex data* emerges. The log sequence representations should consider different dependencies within the overall sequence. One type of dependence originates from the non-conditioned sequential calls between the different functions within the source code. However, as the

program flow of the modern system involves the conditional invocation of multiple system components (e.g., caused by branching conditions), there exist many conditional changes within the log sequences, i.e., diverse log sequences. The conditional changes cause different log events to be part of different contexts (i.e., a set of events frequently co-occurring together). As the scale of the modern systems increase, the event contexts are becoming more diverse. Furthermore, due to the challenge of *software evolution*, log sequences frequently change, further enriching the reasons for novel log sequence dependencies. Another consideration with respect to the complexity of the data resides in the limited usability of the labels (i.e., the *labeling* challenge) which is a commonly referenced reason for the inability to achieve good practical properties of the log-based anomaly detection methods (i.e., the challenge of *low performance of related methods*) or it is limiting the scope of their validity [91]. Another aspect adding to the complexity of the log sequential analysis is the existence of unstable log sequences [180]. The unstable log sequences are log sequences that describe normal system behaviour. However, network errors, limited throughput, or storage issues can cause some events to repeat or be dropped which slightly changes the normal sequence. Therefore, the resulting log sequences are normal, but there are certain events that are missing or duplicated. Notably, the instability makes the log sequences similar to the anomalous sequences (e.g., shortened lengths or contexts differ in a single event), making it harder to distinguish them from one another.

To address the challenges for sequential log anomaly detection we adhere to the *normality* and *anomaly detectability* assumptions. These assumptions enable us to model the normal system state and adopt models with higher modeling capacity, like deep learning-based methods. The *anomaly detectability* enables the detection of the anomalies reflected within the log sequences. To enable anomaly classification, we further refer to the *recurrence* assumption.

The contributions presented within this chapter aid the system operation and development in detecting and classifying sequential anomalies and they are presented as follows [1]:

1. We find that by representing the log sequences as sequences of contextually similar event groups (e.g., groups of neighbouring events), the impurity in the input log sequences is reduced.
2. We propose a novel method, named PLog, that uses deep learning and clustering techniques to extract context-aware event groups.
3. We show that by representing the log sequences as sequences of event groups by PLog, properties of the sequential anomaly detection and classification methods can be improved.

[1] Parts of this chapter are published in [10, 12, 135] and patented in [20].

The remainder of this chapter describes our approach to sequential log analysis. Section 7.1 analyzes the benefits of representing the log sequences as sequences of event groups instead of sequences of log events. Section 7.2 introduces our method, PLog, for extracting event groups. It further shows its application in the overall method named CLog for unsupervised anomaly detection and classification. Section 5.3 presents and discusses the evaluation results. Section 5.4 summarizes the chapter.

7.1 Log Sequence Representation with Event Groups

The log generation within the log file is conditioned on the control flow and the different internal and external inputs and conditions that cause certain behaviours. Intuitively, during normal system operation, the logs that are serving and referring to the execution of a certain function should appear consecutively. Additional logs that are not referring to the function can potentially be intertwined in between. Considering this, if there is a successful normal realization of the function, the same logs are expected to co-occur together within a certain time interval irrespective of their ordering within the interval (i.e., they form an event context). For example, Table 7.1 shows a context of events associated with creating a VM in OpenStack (an open-source cloud computing infrastructure software project) obtained from log dataset introduced in Cotroneo et al. [36]. The VM is successfully created, and the log messages on lines 200, 224, 225, 226, and 240 all refer to the initial and later phases of the successful creation of a VM. The events are appearing within short time intervals, potentially separated by other logs. Note that, the additional logs are also part of the context (line 210 in the example) of a given time window irrespective that they are not referring to a specific function.

Considering this, one can observe a whole log file as composed of different time windows, where each time window refers to a set of different contexts composed of co-occurring logs and additional logs intertwined in between. As a single system is expected to execute similar tasks, some event groups share the same events and experience similar properties, but occur in different time windows. We refer to these groups as event groups (clusters) identified by a single identifier. Note that the identifier does not necessarily have an explicit meaning in terms of executing certain subprocesses. The event groups are naturally appearing one after the other forming the sequence of event groups. By learning the normal similar sequence of event groups one expects that whenever anomalous logs appear within given event groups, the sequence of event groups is disturbed, showing useful information for operation tasks (e.g., detection of anomalies).

Motivated by this intuition, we analysed a log file with logs from OpenStack, introduced by Cotroneo et al. [36] to find if there are benefits of this view on the log sequences. The data is generated by repeating a single workload (common operations associated with

VM, e.g., creating, deleting) many times. For this analysis, we consider just the normal event sequences. We represent the log sequence as a string of characters where each character is a single event. We found 518 events that describe the different runs of the workload. Following our discussion above, we first split the event sequences into event groups on different window sizes (60s, 120s, 180s, 240s and 300s). Therefore, we obtain two representations of a log sequence with a certain task ID: 1) time window event split of a log sequence, alongside its 2) original sequence representation.

Table 7.1: Example of event groups forming the context of VM creation event in OpenStack.

Log Line	Timestamp	Log Message
200	00:49:18.456	Claim successful on node localhost.localdomain
210	00:49:26.362	GET 10.0.20.46:35357
224	00:49:33.667	Took 12.51 seconds to spawn the instance on the hypervisor
225	00:49:33.896	During sync_power_state the instance has a pending task (spawning). Skip.
226	00:49:33.900	VM Created (Lifecycle Event).
240	00:49:34.041	Took 15.67 seconds to build instance.

To compare the impact on the event sequence representation over the log sequences, we used entropy [63]. Entropy enables quantifying the differences in the uncertainty of the different representations as it measures the impurity of the input sequences. From a modeling perspective, higher entropy relates to more challenging input data as the relevant information important for the operational task is closely intertwined with additional less relevant data. The entropy on the *time window event split* representation for a single task ID is calculated by first calculating the entropy on the individual event windows, and afterwards averaging the entropies over all the windows of the log sequence with the same task ID. The entropy for the original sequence representation is calculated directly over the original log sequence. For both representations, the entropy is averaged over all the log sequences with a task ID in the data. That way, we can evaluate the differences in the impurity in different representations. Figure 7.1 illustrate the entropies with the two types of representations (black rectangles denote the input represented by the *time window event split*, while the diamond is the entropy over the individual sequences represented by their original length). It can be seen that as the time window increases so does the entropy. The entropy peaks when the sequences are represented by the overall event sequence associated with a task ID. Notably, for smaller time windows the entropy also decreases. Therefore, by aggregating the windows, we expect that the impurity of the modeling sequences will decrease. As this reduces the impurity of the modeling input data, the latter representation may expose richer information for modeling, potentially suitable for sequential log analysis.

In addition, as we expect that similar log sequences exist among the different time windows, we identify similar event groups and assigned an individual identifier for each of them. [2] Therefore, the log sequences are represented as sequences of group identifiers. By calculating the average entropy of the log sequences associated with a single task ID on this representation, it can be seen that the entropy is further reduced. The representation with event group identifiers further reduces the impurity in the log sequence. Inspired by this observation, we consider that it is beneficial to learn these context-aware groups, to reduce the uncertainty within the overall log sequences. Therefore, an important goal of the proposed method is to learn these event sequence groups while preserving their characteristics (e.g., preserving contextual event information).

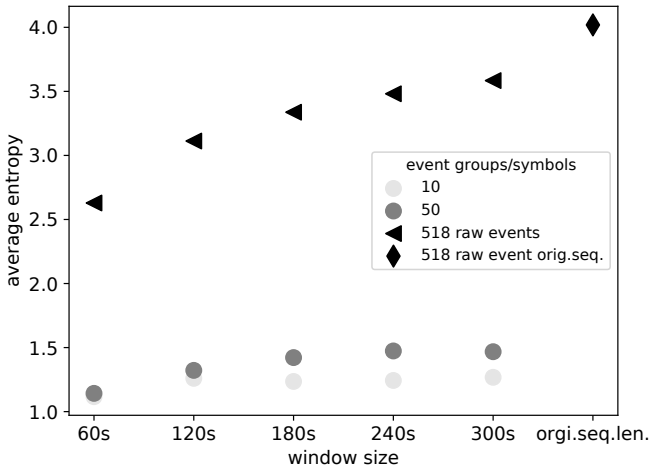


Figure 7.1: Impact of the log sequence representations on entropy.

7.2 CLog: Method for Sequential Log-based Anomaly Detection and Classification with Event Groups

To address the problems of sequential log-based anomaly detection and classification, we propose CLog. Figure 7.2 gives an overview of the method. It has three parts 1) log parsing, 2) context-aware event group extraction, and 3) anomaly identification [3]. Log parsing, as a general preprocessing procedure in log analysis [187], extracts the event templates from the incoming raw log events, transforming the raw log message sequences into sequences of log event templates. The event template sequences are processed by the context-aware event group extraction part, converting them into sequences of event

[2] We explain the process of similar event group extraction in the following parts of the chapter.

[3] We use the term anomaly identification to refer to the two problems of anomaly detection and classification jointly.

groups. This part leverages the observation that by representing the log sequences on a level of event groups, the entropy of the representation sequence is smaller. Finally, the processed log sequences are given as input into the anomaly identification part. The latter is composed of two modules (a) anomaly detector and (b) anomaly classification. The goal of the anomaly detector is to detect the anomalous sequences of event groups. The anomaly classification module leverages the recurrence assumption to further identify the anomaly class. The anomaly classes are stored in a knowledge base of instances with sequential anomalous classes. In case when this knowledge base is not available, the classification part is not used. CLog has two modes of operation: offline and online. During the offline phase, the parameters of the context-aware event group extraction part, and the anomaly identification parts are learned, and the learned models are stored. In the online phase, the stored models are loaded and used to identify anomalies. Note, that the method expects parsed log events as input, therefore, we keep the log parsing part in the figure to emphasise this need. As we discussed the log parsing part in Chapter 6 we do not discuss it here. In the following, we describe the internal mechanisms of the second and third parts of CLog in detail.

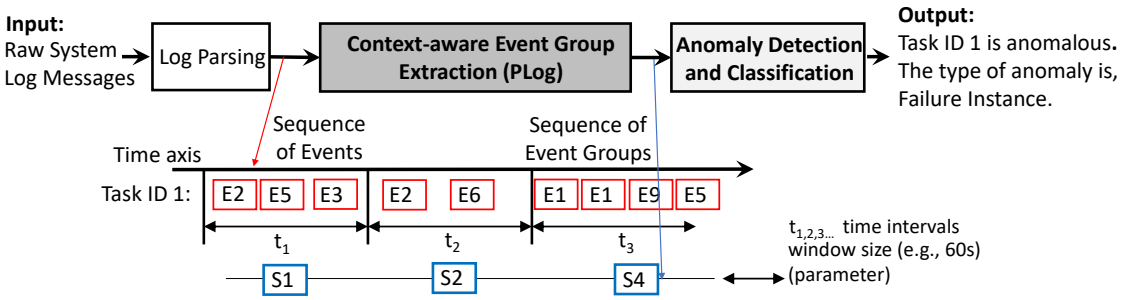


Figure 7.2: CLog architecture overview.

7.2.1 PLog: Context-aware Event Group Extraction

The context-aware event group extraction is the central part of the method. Its goal is the extraction of event groups from the parsed log event sequences. By representing an execution workload on a higher-level granularity, i.e., by event groups, we reduce the entropy in the input learning samples. The context-aware event group extraction combines context-aware neural network and clustering methods to learn explicit relationships between the events within the event sequences preserving their local properties. For example, in the Table 7.1, the log message "Took 15.67 seconds to build instance.", although, appears after the log message "Took 12.51 seconds to spawn the instance on the hypervisor", it still refers to the successful event of VM creation. Considering this information from the context is beneficial for the learning of the representation model

as it is additional information the model can learn from. We choose a neural network-based approach to learn the sequential embeddings due to the neural network's ability to extract complex dependencies from the input event sequences. The availability of many instances to learn from, and the sequential nature of the data are generally considered domains where neural networks can use their high modeling capacities and are preferred method choices [56]. We choose a clustering approach towards extracting the event groups on a given dataset, as it is unknown which event sequences are similar. As the clustering process groups the similar event sequences based on how similar the learned representations from the neural network are, it allows considering the cluster prototype as an event group identifier.

Figure 7.3 depicts the overall design of the context-aware event group extraction part with a running example. Conceptually, it is composed of three submodules – (1) preprocessing submodule that transforms the input sequences into a format suitable for learning, (2) a neural network learning module which is combined with a batched k-means method to learn event groups in an unsupervised manner, (3) event group extraction module that assigns a unique event group identifier to the input event sequence. In the following, we describe the submodules.

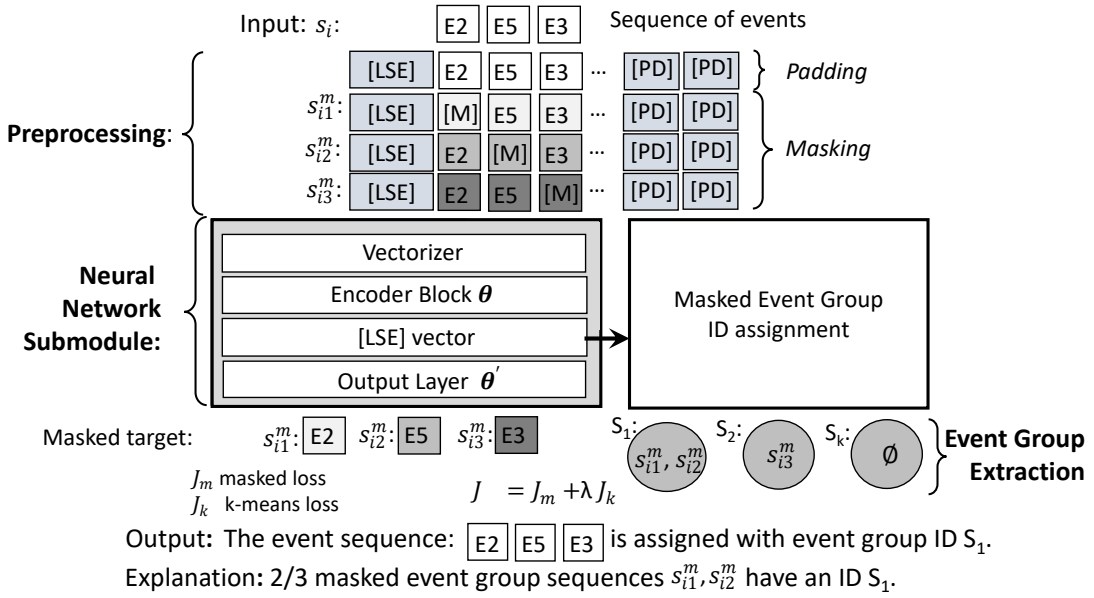


Figure 7.3: Internal design of the context-aware event group extraction part (with a detailed explanation of a running example).

Preprocessing Submodule

The goal of the preprocessing submodule is to preprocess the input log sequences in a unified format for the neural network. It has two components: padding and masking. *Padding*; The padding component receives the sequences of log events as input, with each event represented by a unique symbol (e.g., integer). We refer to it as a token. The sequences of events in a given time interval are different in length. However, the neural network requires a fixed-size representation of the input. The padding component specifies a hyperparameter max_length and appends each of the shorter log sequences with a dedicated token $[PD]$ up to max_length to enforce fixed-size representation. The longer sequences are truncated. Notably, we add a dedicated token $[LSE]$ (Log Sequence Embedding) at the beginning of each sequence. During learning, we enforce the sequence token representations to propagate through the upper layers in the network via the $[LSE]$ token. Thereby, $[LSE]$ attends over all the tokens from the sequence and summarizes the relevant context during learning. The $[LSE]$ token serves as a sequence vector representation used to group contexts and identify event groups. The output of this module is the prepended and padded event sequence.

Masking. To learn context-aware groups, we consider a general self-supervised learning task from NLP research called Masked Language Modeling (MLM) [40]. To apply the MLM task, the masking component is processing the prepended and padded log event sequences in a suitable format. More specifically, as input, it receives the prepended and padded log event sequences and outputs a set of pairs of masked log event sequences and original masked events. Masked log event sequences are sequences of log events created by replacing all of the events from an original log sequence with a special $[M]$ (masked) token. For example, for the input sequence (E_2, E_5, E_3), one masked sequence is $(E_2, [M], E_3)$, with E_5 being the original masked event. There are three masked event sequences for this example. $[LSE]$ and $[PD]$ tokens are not affected by the masking procedure. During training, a masked sequence is given as input to the neural network, while the original masked token is used as the prediction target. By predicting the mask from the co-occurring tokens, the method learns the most important events from the surrounding context, extracting context-aware representations. The capability of the MLM task to consider the learning of local contexts is the main reason for its selection. Note that by this procedure single input sequence is multiplied several times. We keep track of the origin (the input event sequence) of each masked sequence and use it to extract its corresponding event group identifier.

Neural Network Submodule

The neural network submodule learns context-aware groups of masked log sequences. To do so it implements a neural network following the design of a self-attention encoder of the

Transformer [40] architecture. The advantage given by this architectural choice resides in its capability to learn the contextual information between the input events. When learning the parameters of the network, guided by a carefully designed cost function, the model learns local relationships between the events based on their co-occurrence extracting useful contextual features. The neural network submodule has four components: vectorizer, encoder block, output layers and masked event group id assignment.

The **vectorizer** transforms the masked input event sequences of tokens into numerical vector sequences. These vectors are called *event embeddings* and are part of the training procedure. At the beginning of the training procedure, the event vectors are randomly initialized and updated during training. This way, they learn contextual information about the events.

The **encoder block** is composed of a self-attention encoder layer. The self-attention extracts co-occurring information by weighting the input vector embeddings by their similarity to all the other embeddings in the given context. Combining the self-attention with the MLM learning task modifies the parameters of the network to learn the context of the original masked event and extract sequential properties. The hyperparameters of the encoder are the model size (denoted by d), the number of encoder layers, and the number of heads. Particularly interesting is the embedding of the $[LSE]$ token. Since $[LSE]$ serves as an embedding of the input event sequence group, it learns the contextual properties of the masked input sequences. The output from the encoder is the vector embedding of the $[LSE]$ token for each of the masked sequences, proceeding towards the output layer.

The **output layer** is composed of two layers with nonlinear activation (RELU is used). The purpose of it is to map the masked sequence embedding vector $[LSE]$ of size d, to a vector with a size corresponding to the total number of events/tokens C. The RELU activation function is a common activation function used to introduce nonlinearities and improve the learning process [56]. The output of this layer is used to calculate the loss. As an optimization loss function, we use categorical cross-entropy. This loss is a common choice for multi-class classification problems as addressed by MLM task [56]. Notably, during the execution of a workload, some events occur only once (e.g., "notification of successful creation of a VM") while others occur with greater frequency (e.g., HTTP or RPC calls). When using the original loss formulation on the MLM task, the less frequent events can be averaged out, resulting in missing important information. To account for the imbalances of the distribution of the events, we use weighted categorical cross-entropy given in Eq. 7.1 as follows:

$$J_m(\psi(\mathbf{s^m_{n,c}};\theta,\theta^{'}),y^m_{n,c};\mathbf{w}) = \frac{1}{|C|}\sum_{c=1}^{C} -w_c y^m_{n,c} log \frac{exp(\psi(\mathbf{s^m_{n,c}};\theta,\theta^{'}))}{\sum_{i=1}^{C} exp(\psi(\mathbf{s^m_{n,i}};\theta,\theta'))} \tag{7.1}$$

where ψ denotes the function modeled by the neural network, θ and θ' are the parameters of the encoder, and the output layer accordingly, $\mathbf{s^m_{n,c}}$ is a masked sequence obtained from the n-th input sequence $\mathbf{s_n}$, $y^m_{n,c}$ is the original masked event/token, C denotes the total token numbers and w_c represents the weight of an individual token. The weights ($\mathbf{w}$ – a weights vector) are assigned such that the less frequent events have weight values closer to 1, as opposed to the frequent ones that have values closer to 0. Therefore, we optimize for preserving the correct predictions on the infrequent events.

Masked Event Group ID assignment. The masked event group ID assignment receives the vector embedding of the $[LSE]$ token as input. Its goal is to identify similar context-aware masked sequences and group them. It applies the mini batched k-means algorithm [174] to group the embeddings of the masked event sequences into a predetermined number of $\mathbf{k}$ event groups/centroids identifiers. While the goal of the encoder block is to learn context-aware representations, the mini-batched k-means complements it by extracting similar context groups, enabling the extraction of event groups. The k-means algorithm is a commonly used method for identifying similar instance groups in an unsupervised way [66]. We used its k-means mini-batch version because it allows per batch update of both the network (θ and $\theta^{'}$) and clustering parameters ($\mathbf{M}$) as opposed to the classical k-means method. To group the contexts, k-means optimizes the loss given in Eq. 7.2 by altering between two steps: 1) updating a centroid m_k as the average of the embeddings currently assigned to it, and 2) reassignment of the embeddings to the nearest newly calculated centroid.

$$J_k(\phi(\mathbf{s^m_n},\mathbf{r_n};\theta),\mathbf{M}) = ||\phi(\mathbf{s^m_n};\theta) - \mathbf{r_n M}||_2 \tag{7.2}$$

where $\mathbf{M} \in \mathbb{R}^{kxd}$ represent the matrix of event groups (interchangeably referred to as centroids), while $\mathbf{r_n}$ is an indicator vector of discrete values (0's and 1's) with just one element set to one, corresponding to the membership of the masked sequence $\mathbf{s^m_n}$ to a certain centroid m_k. The number of event group identifiers $\mathbf{k}$ is a hyperparameter.

Finally, we add the two optimization losses as $J = J_m + \lambda J_k$ to obtain the final loss subject to optimization. By combined optimization of the two losses, the parameters of the context-aware event group extraction learn local contexts and local-context groups based on their similarity. The role of the hyperparameter λ is to ensure the learning of correct contexts and correct context-embedding groups by trading off the impact of the two losses. We further discuss the optimization procedure.

Optimization. The optimization is done in two phases: 1) pretraining and 2) joint training. The reason for this is explained in the following. We first describe the *pretraining phase*. Since at the beginning everything is initialized at random, we pre-train the neural network parameters (θ and $\theta^{'}$) by the weighted cross-entropy loss (Eq. 7.1). That way, the model learns good initial parameters for the encoder while extracting context-aware features for the masked sequences. The pretraining is terminated after observing a lack of improvement in the loss on several epochs [55]. At the end of the pretraining, the $[LSE]$ vectors are valid representations of the masked input sequences. Afterwards, the event group prototypes ($\mathbf{M}$) are initialized by k-means using $[LSE]$ masked sequence embeddings of the training data.

$$m_k \leftarrow m_k - \frac{1}{c_k}(\phi(\mathbf{s_n^m}; \theta) - m_k)\mathbf{r_n} \tag{7.3}$$

Joint training (phase 2). The joint optimization function has a discrete variable ($\mathbf{r_n}$), making the parameter updates non-trivial. To address this issue, we adopt the Alternating Stochastic Gradient Descent (ASGD) [174] algorithm. ASGD alters the updates of the network parameters and centroids such that, when the network parameters are updated, the centroids are fixed and vice versa. Therefore, the optimization problem does not depend on the discrete variable, enabling the parameter updates. The training of the network parameters and the centroids is done in batches. Eq. 7.3 is used for centroids update. At each batch, the centroids with newly assigned embeddings are slightly updated based on their distance from the newly calculated centroids.

Event Group Extraction

Once the masked event groups are clustered, the event group identifier (ID) is executed. The procedure we consider is presented in the following. Given an original input event sequence and the event group ID assignments of its masked subsequences, we count the number of occurrences of the event group IDs and divide the counts by the length of the original input event sequence. The event group ID with the highest score value is assigned as an event group ID for the input event sequence. Intuitively, if the majority of the masked subsequences are assigned with a single event group ID, the event group ID with the maximal score value is the most relevant for the input event sequence. Figure 7.3 depicts an example of extracting the event group S_1 for the sequence (E_2, E_5, E_3).

7.2.2 Sequential Anomaly Detection

The anomaly identification part is given sequences of event groups with the same task ID as input. Figure 7.4 depicts the internal design. It is composed of two subparts 1) anomaly detector and 2) anomaly classification. The anomaly detector detects if the

input event group sequence is anomalous. When an anomaly is detected, the sequence proceeds towards the anomaly classification part. This part identifies the class of the anomaly conditioned if the knowledge base of sequential anomaly classes exists. We describe the details in the following.

As a modeling choice for the *anomaly detector*, we consider Hidden Markov Model (HMM) [173]. HMM models the sequences of event groups by assuming that the appearance of the next event groups within the sequence depends only on the currently observed event group (the Markov property). The main advantages of HMM are that it directly handles sequential data, does not require further preprocessing of the input, and is fast for both learning and inference (with a reasonably high number of hidden states). As the produced sequences are expected to be with lower entropy and lower element count, we prefer HMM over LSTM or other models. The reason behind this is the need to employ additional design choices on how to train and evaluate the anomaly detector in that case.

To produce normality score estimates for a sequence $\tilde{p}^{+}_{seq}(\mathbf{s_i})$, we used HMM probability scores ($t(\mathbf{s})$), calculated by marginalizing the probabilities over all the event groups of the sequence and the hidden states of the fitted HMM $t(\mathbf{s}) = -\log \sum_h q(h)q(\mathbf{s}|h)$, where h denotes the hidden states, and $q(\mathbf{s}|h)$ denotes the likelihood of the event groups given the hidden state. The normality score estimates for a single sequence $\mathbf{s_i}$ is given in Eq. 7.4, as follows:

$$\tilde{p}^{+}_{seq}(\mathbf{s_i}) = \left(\frac{1}{|\mathbb{V}|}\sum_{s_j}^{|\mathbb{V}|} t(\mathbf{s_j}) - t(\mathbf{s_i})\right)^2 \tag{7.4}$$

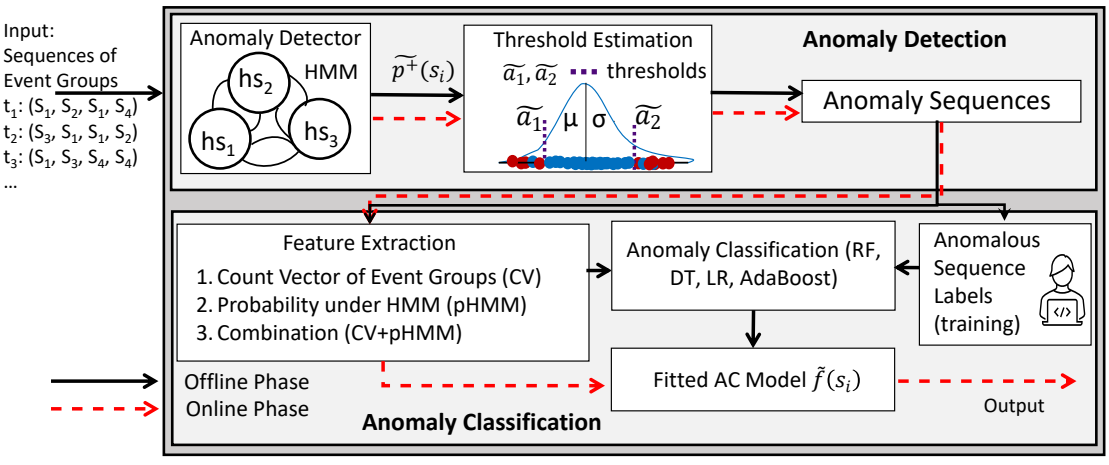


Figure 7.4: Internal architectural design of the anomaly detection and classification subcomponent.

where $\mathbb{V}$ is the set of normal sequences of event groups. The normality score estimate $\tilde{p}^{+}_{seq}(\mathbf{s_i})$ is a symmetric positive function, given as the spread of the probability of the sequence $\mathbf{s_i}$ under the HMM ($t(s_i)$) from the mean score estimates of the normal data. The parameters of the HMM are learned on the normal training data, thereby, the anomaly detector models the normal system state. We refer to the *normality* assumption and assume that the normal data is always obtainable. Considering the existence of normal data addresses the *labeling* challenge. Any sequence with significantly different values for the normality score estimate is *detected* as an anomaly. Using the symmetrical property of the normality function, we *estimate the thresholds* as $\tilde{a}_{1/2} = \mu \pm r\sigma$, where μ and σ are the mean value and standard deviation of the normal score estimates calculated by standard formulas, and r is a parameter denoting what is considered a significant deviation from the normal state. Thereby, the anomaly detector is fully unsupervised. The hidden state number is a hyperparameter of HMM.

7.2.3 Sequential Anomaly Classification

Once the anomaly is detected, the anomalous sequence proceeds towards the anomaly classification module. It has two components (1) feature extraction and (2) anomaly classification method.

The *feature extraction* processes the sequences of event groups in a format suitable for the anomaly classification learning method. Each sequence is represented by a count vector that counts the number of occurrences of the event groups within the sequence. For example, for the sequence of event groups $\mathbf{s} = (S_1, S_3, S_1)$ and total of four event groups (S_1, S_2, S_3, S_4), the count vector is given as $CV(\mathbf{s}) = (2, 0, 1, 0)$. The absence/presence of certain event groups from the sequence is a distinctive feature that discriminates among anomaly types. Therefore, the count vector is a suitable sequence representation. We also considered the normality score estimates from the anomaly detector as an additional feature (pHMM). This feature summarizes in a single number the overall sequence information. As similar anomalies may be reflected in a similar manner in the data, the summarization of the overall sequence information has certain modeling capabilities. When class labels are not available, the classification module is not active.

The extracted features are used to fit *anomaly classification* model. The anomaly classification part learns a multiclass classification model to classify the input sequences into several predefined types of anomalies. As adequate methods we considered several popular multiclass classification methods, i.e., Random Forest (RF) [16], Decision Tree (DT) [142], Logistic Regression (LR) [67] and AdaBoost [50]. They show good performance and are simple to tune [165].

Table 7.2: Datasets statistics.

Dataset Name	Tasks	Anomalous Sequences Count	Logs	Unique Events	Average number of events in a task
OpenStack	876	257	217534	518	248
Unstable Sequence Data	876 + b%	257	217534	518	248

7.3 Evaluation

In this section, we describe the experimental evaluation. We give details about the experimental design and discuss the results of four experiments analyzing different aspects of the method. Specifically, we first compare CLog against related unsupervised methods. Second, we study the impact of the different window sizes on the method performance. Third, we discuss the impact of sequence instability on the anomaly detection performance of CLog. Finally, we show the evaluation of the anomaly classification task.

Experimental Design

Datasets; To evaluate CLog, we considered a large-scale study of failures in OpenStack, introduced in Cotroneo et al. [36]. To the best of our knowledge, it is the most comprehensive publicly available dataset of anomalous log data. Its strength is the wide range of covered anomalies following the most common problem reports in the OpenStack bug repository[4]. The faults are generated by software fault-injection procedure, i.e., modifying the source code of OpenStack and running a predefined workload under fault-injected and fault-free (normal) conditions. Each workload has a rich description of the anomaly class type and its fault. Therefore, it is suitable for tackling the tasks of sequential log-based anomaly detection and classification.

The considered fault types are grouped into four groups as of following: 1) *throw exception* (method raises an exception in accordance to a predefined API list), 2) *wrong return value* (method returns an incorrect value, e.g., return null reference), 3) *wrong parameter value* (calling a method with an incorrect value for a parameter), and 4) *delay* (method returns the result after a long delay, e.g., caused by hardware failure – leading to triggering timeout mechanisms or stall). As a running workload with a unique task ID, the authors considered the creation of a new instance deployment. This workload configures a new virtual infrastructure from scratch – it creates VM instances, volumes, key pairs, and security groups, a virtual network, assigns instance floating IPs, reboots the instances, attaches the instances to volumes and deletes all resources. Importantly, this comprehensive workload invokes the three key services of OpenStack Nova, Cinder, and Neutron, causing diverse manifestations of the faults as anomalies. Due to the nature of the data generation process, and access to the source code of OpenStack, there exists

[4]https://launchpad.net/openstack

a sequential relationship between the generated log sequences for the individual subparts of the overall workload (e.g., the repetitiveness of co-occurring events). Therefore, this dataset has emphasised sequential dependencies between the logs making it suitable for sequential log analysis.

To generate ground truth labels for the anomalous state, assertion and API checks are performed at the end of the workload runs. There are three failure types: *1) failure instance*, *2) failure SSH* and *3) failure attaching volume*. While the authors provide information on a granularity of a workload with a task ID, we further examine the individual logs. More specifically, two human annotators analysed the logs and annotated individual templates to obtain a better understanding of the types of anomalies that are reflected in the data. Table 7.2 gives detailed statistics of the used data.

Unstable Sequence Data; To evaluate the robustness of our method in dealing with unstable log data, we create an additional dataset. Zhang et al. [180] describe a data generation process on how to create unstable sequences from a given dataset which we adopt. We apply the following two operations on the data to extract anomalous sequences (from the normal sequences from OpenStack), i.e., 1) random removal of log events, and 2) repetition of a randomly selected log event in the sampled log sequence. To inject unstable log event sequences, we randomly sample b-percentage of the normal training data, and we inject the aforenamed operations in random order. The generated data is joined to the normal data and later used to build a model that is evaluated on the remaining data.

In general, the access to labeled log data with good quality suitable for sequential log anomaly detection is limited [70, 88]. As discussed in the previous chapter, there exist several labeled log datasets. One dataset with emphasised sequential characteristics is HDFS [172]. Landauer et al. [88] manually analysed the log sequences from HDFS and reported that by relying on two heuristics: 1) looking for new events in the sequences and 2) shorter sequences than the normal, they can achieve more than 0.9 on F_1, and up to 0.98 on precision. Therefore, following their recommendation, we do not consider this data in the evaluation. BGL and SPIRIT [132] are two other available labeled datasets. However, those systems are from supercomputers where the different cores are constantly dumping log messages that are weakly coupled. As we lack domain expertise on the relationships between the logs (which are the underlying subprocesses), we do not know if the data generation processes produce log sequences with stronger dependencies, important for the evaluation of the sequential log analysis. Furthermore, the available labels do not specify sequential anomaly classes. Therefore, we do not use these datasets in the evaluation. To mitigate the issue of availability of quality labeled data on a sequential level, we further adhered to literature practices [180] and generated additional data for evaluation.

7.3.1 Sequential Anomaly Detection

Competing methods; We compare the anomaly detection method against two unsupervised competing methods: DeepLog [44], and Hidden Markov Model (HMM) previously used in log anomaly detection in Yamanishi et al. [173]). Notably, as shown in Cotroneo et al. [36], around 20% of the anomalies in the dataset are not explicitly logged in the logs. Therefore, the semantic-based methods, due to the challenge of *insufficient failure logging coverage* are not used in the comparison as their optimal performance is not expected to exceed more than 0.8 on anomaly detection.

Experimental Design; We conduct the experiments as follows. The optimized hyperparameters of CLog are the number of extracted event groups, the window size, and the number of hidden states of the HMM. They were selected from the range values of the sets $\{10, 20, 30, 40, 50, 100, 200\}$, $\{60s, 120s, 180s, 240s, 300s\}$ and $\{2, 4, 8, 16\}$ accordingly. Experimentally we find the following values for the learning process to be robust across the different experimentation settings: for *phase 1* the training was performed for a maximal of 200 epochs, and *phase 2* training for a maximum of 20 epochs. As an optimizer, we use SGD with a learning rate set to 0.0001. For the encoder, the model size d was set to 128, with two encoder layers and four heads. To prevent overfitting, we set the dropout rate to 0.01. Experimentally, we find that λ with a value of 0.1 leads to robust results. The hyperparameters of the competing methods for anomaly detection are tuned to produce their best F_1 scores. The anomaly detection performance was evaluated on F_1, precision and recall as common evaluation metrics, with the anomalies being labeled with a positive label. The training is done on 60% randomly sampled normal sequences. The rest of the sequences are used to report the performance scores. The experiments are repeated up to five times to reduce the influence of the samples included for learning. We report the mean and standard deviation of the results. The experiments were conducted on a Linux server with Intel Xeon(R) 2.40GHz CPU and RTX 2080 GPU running with Python 3.6 and PyTorch 1.5.0.

Table 7.3: Comparison of CLog against competing methods on sequential anomaly detection.

	F_1	Precision	Recall
CLog	0.93±0.01	0.91±0.03	0.95±0.02
HMM	0.93±0.02	0.87±0.03	0.99±0.01
DeepLog	0.91±0.01	0.84±0.02	0.99±0.01

Results and Discussion; Table 7.3 shows the results of the unsupervised method comparison. The results show that all the methods perform well, with similar F_1 scores. CLog

and HMM both use HMM to model the sequences, but they differ in the granularity of the input representation. CLog uses a sequence of event groups, while HMM uses a sequence of events. The results show that changing the input representation of the log event sequences with sequences of event groups does not impair the anomaly detection performance. Combining the results with the observation depicted in Figure 7.1) points that changing the input representation as CLog does is non-detrimental for the detection performance. Predominantly, the improvement in performance originates from precision (it is higher for 0.04 for CLog compared to the related methods). As CLog includes a broader context of related events (instead of just autoregressive information) it can reuse the information of a latter event occurrence that shares some relation with a current event (recall the example in Figure 7.1), which leads to better discrimination of the anomalous inputs. CLog makes use of all the information within the time window, as opposed to DeepLog and HMM which consider just information prior to a given event. Specifically, the learning sample construction process enables CLog through the Transformer architecture to relate the contextual properties between distant events within the time interval. Although CLog uses HMM on the learned representations, the final sequence is created by considering the learned contextual dependencies, where the context may span over many events. This additional information can help CLog learn more informative representations of the normal input. In contrast, the input of competing methods has larger entropy, which challenges the learning of the normal sequences. This leads to more of them being detected as anomalies, which further explains the improvement CLog achieves in precision over DeepLog and HMM.

While a direct comparison between CLog and HMM on anomaly detection performance shows differences within the standard deviations, CLog has an advantage with respect to learning the HMM model of normal system behaviour. As CLog uses the event group identifiers it describes the overall sequences with fewer symbols, enabling faster training of the anomaly detector. To observe this practical advantage, we further plot the learning times of the anomaly detector for the two methods, CLog and HMM. We run the anomaly detector with the two different representations as used by CLog (sequence of event groups) and HMM (sequence of events) five times for each of them. Figure 7.5 plots the average times from the runs. As the number of states in the HMM increases, so does the training time for both representations. However, this increase is more emphasised for the sequence of events, instead of the sequence of the event groups. As the latter uses all the event templates (approximately 500), CLog uses just a fraction of it, making the training of the anomaly detector faster. Note that CLog does not experience an anomaly detection drop in performance, but improves the input representation, such that another property, i.e., the training time of the anomaly detector is improved.

When comparing CLog against the related works, one property it distinguishes itself

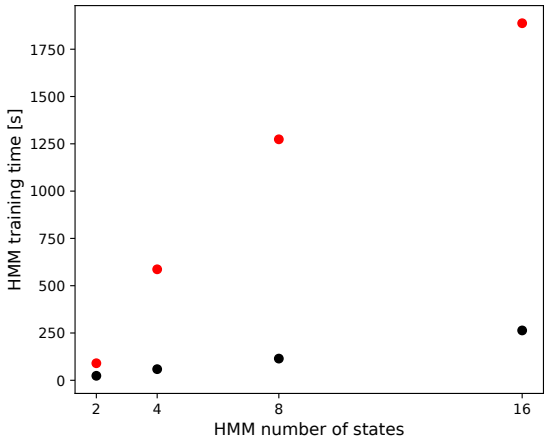


Figure 7.5: Comparison of the different representations on the training time of HMM as anomaly detector.

from the other unsupervised methods is the utilization of the full contextual information within the input time window sequence compared to DeepLog or HMM. Similar operations (e.g., creation of a VM) are expected to have a similar invocation chain of logs that co-occur within relatively shorter time intervals. As CLog considers the broader context, instead of just the previous information, it exposes richer information to the learning method. The representation of the sequences with event groups also reduces the entropy of the final event sequence. This design idea is a distinctive feature of the proposed method. Another log-based anomaly detection method that uses clustering is LogCluster. However, this method clusters the sequences and later uses heuristics of what is considered an anomalous class to label the clusters as anomalous. In contrast, we use the event group identifiers to construct a sequence on top of which we learn a sequential model for anomaly detection. As seen from the results this is beneficial as it does not reduce the performance while it improves other practical properties, for the tested dataset.

Time Window Impact over Detection Performance; The time window hyper-parameter determines the granularity level of event group extraction. Therefore, it is an important hyperparameter of the method. Following the discussion on the entropy reduction by different window sizes given in Section 7.1, we grouped the input events into time intervals of increasing window size and evaluated several models of CLog.

Results and Discussion; Figure 7.6 shows the results. It can be observed that as the window size increases, the detection performance decreases. Paring the average entropy over the sequences for different window sizes (as depicted in Figure 7.1) with the detection results reveals a negative correlation between the increased entropy and anomaly detection

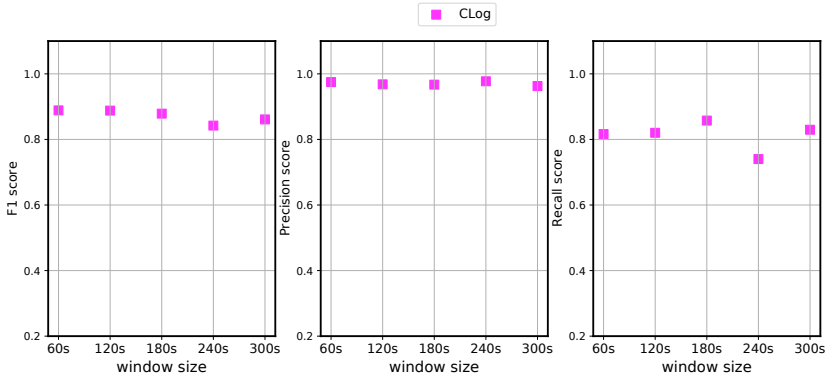


Figure 7.6: Window size impact on the anomaly detection performance.

performance. As the window size increases, CLog has a drop in performance on the F_1 score. While CLog preserves good performance on precision, the recall drops. Therefore CLog has a harder time distinguishing the correct anomalies due to the diversity in the different contexts coming with the larger window sizes.

Evaluating CLog on Unstable Log Sequences

Experimental Design; One challenge in dealing with log sequences is dealing with unstable log sequences. Therefore, we evaluate CLog on a dataset derived from the OpenStack data following literature recommendations on generating unstable sequences [180]. We created the unstable datasets by randomly injecting $b-$percentages unstable sequences using the dataset generation procedure previously described. We varied b in the ranges $b = \{5\%, 10\%, 15\%, 20\%\}$, similar to related works [180]. We used the sampled and modified normal log sequences to learn a model and use the remaining ones to evaluate the method. The anomaly detector was trained on the original data as in the case when we compared CLog against the competing methods.

Results and Discussion; Table 7.4 shows the results. It can be seen that CLog preserves a high detection performance on F_1 in the case of a small percentage of unstable log sequences. This can be attributed to the fact that their lower count in comparison to the normal data is not accounted for by the model. Importantly, as the ratio increases, CLog experiences decreased performance. Notably, the performance under a larger fraction of unstable sequences (i.e., 20%) drops by 0.06 on the F_1 score. As more unstable sequences are present in the data, the model has a harder challenge detecting the anomalies due to the similarities between the anomalous and unstable data. This is seen by the drop in the recall. It is worth mentioning that although the method drops in performance, the drop is not significant. The extracted event groups still preserve discriminative properties enabling anomaly detection.

Table 7.4: CLog anomaly detection evaluation on unstable log sequences.

injection ratio	F_1	Precision	Recall
0.05	0.86±0.001	0.97±0.001	0.78±0.001
0.1	0.84±0.002	0.97±0.001	0.74±0.002
0.15	0.81±0.001	0.96±0.001	0.69±0.001
0.2	0.80±0.030	0.97±0.001	0.68±0.030

7.3.2 Sequential Anomaly Classification

Experimental Design; We evaluate the capability of the anomaly classification module of CLog to reuse the historical information from the operator in anomaly classification by leveraging the recurrence assumption. Specifically, we evaluate three representations of the event group sequences for CLog (1. probability score from the HMM (pHMM), 2. count vectors (CV), and 3. combination of both). As a competing method, we considered LogClass implemented with TFIDF representation of the logs [119]. We used the anomaly class of the task ID as a target label and apply the two methods. As multi-class classification methods, we considered several popular classifiers: Random Forest (RF) [16], Decision Trees (DT) [16], Logistic Regression (LR) [67] and AdaBoosting [50] for both of the approaches. The hyperparameters of the anomaly classification methods are set on their implementation defaults from the sckit-learn library. F_1, precision and recall are used as performance scores for the anomaly classification. As the classification problem is multiclass, we consider the macro averaging over the three anomaly classes. We train the models on 60% of the data and evaluate the remaining 40%. To decrease the effect of the randomness in the training data, we repeat the experiments multiple times and report the mean and standard deviation.

Table 7.5 enlists the results of the three different representations of CLog and the baseline on the sequential anomaly classification subproblem. The analysis of the three representations by CLog suggests that the combination (CV+pHMM) achieves the best F_1 score. Predominantly, the improvement originates from the count vectors, seen by the better individual results in comparison with pHMM. The major advantage of the count vectors is that they have greater granularity on the representation, while pHMM summarizes the sequences of event groups on a coarse level. The comparison across the two methods suggests that the improvements in the F_1 score are within the range of 0.03. The combination (CV+pHMM) of CLog outperforms the competing method LogClass. LogClass uses single logs to identify the type of failures. Therefore, if the failure is not explicitly logged, LogClass cannot identify its type. On the contrary, CLog considers the occurrence of the individual event groups and can represent discriminative patterns among the types of anomalies improving the performance.

Table 7.5: Comparison of CLog against baselines on sequential anomaly classification.

Scores	Multiclass method	CLog (pHMM rep.)	CLog (CV rep.)	CLog (CV+pHMM combined)	LogClass (TFIDF)
F1	RF	0.74±0.0	*0.86±0.01*	*0.86±0.01*	0.84±0.08
	DT	0.74±0.0	0.78±0.04	0.78±0.03	*0.84±0.05*
	LR	0.72±0.0	0.86±0.0	**0.87±0.0**	0.8±0.1
	AdaBoost	0.69±0.0	0.86±0.02	**0.87±0.02**	0.62±0.12
Precision	RF	0.74±0.0	*0.86±0.01*	*0.86±0.01*	0.83±0.07
	DT	0.74±0.0	0.78±0.04	0.78±0.03	*0.82±0.05*
	LR	0.71±0.0	0.85±0.0	**0.87±0.0**	0.77±0.08
	AdaBoost	0.71±0.0	0.86±0.02	0.86±0.02	0.59±0.13
Recall	RF	0.75±0.0	0.87±0.01	0.87±0.01	*0.86±0.06*
	DT	0.75±0.0	0.81±0.04	0.81±0.03	*0.85±0.07*
	LR	0.73±0.0	0.88±0.0	**0.89±0.0**	0.88±0.11
	AdaBoost	0.73±0.0	0.87±0.02	0.88±0.02	0.74±0.14

7.4 Chapter Summary

This chapter addressed the problem of the automation of sequential log-based anomaly detection and classification. It introduced a novel method CLog, which addresses the subproblems of sequential 1) anomaly detection and 2) anomaly classification. We noted that by representing the input log data as sequences of event groups instead of sequences of individual events, the entropy in the input is reduced. CLog uses this observation and introduces an event group extraction method, which jointly trains context-aware deep learning and clustering methods to extract event groups. Our experiments demonstrated that the extracted sequences of event groups can be beneficial for not-deteriorating the performance on the two subproblems, and can potentially improve other properties, as was the case with the learning time on the anomaly detector. The *unsupervised design* of CLog and the *modeling of sequences* helps to address the challenges of *insuficient anomaly coverage*, *complex data representation*, *labeling* and *lower detection performance* of competing methods. At the same time, CLog is orthogonal to the single log line analysis module as increases the set of detectable anomalies that are reflected within the sequences, addressing the challenge of *insufficient failure coverage* in the source code. Therefore, it further contributes towards our goal of improving the operational and development activities for supporting system dependability through automation.

As CLog groups the events within a fixed time interval before detecting an anomaly, it needs to aggregate all the windows within a certain time interval before their detection. This may delay the reporting of an anomaly. Thereby, in a practical application where fast anomaly detection is important, smaller window times are preferable. In addition, CLog relies on the existence of similar groups within the log data. As modern systems

are composed of multiple services, the log patterns within the data may be more heterogeneous increasing the number of event groups, while reducing their similarity. In the marginal case, this results in representing each event with a single cluster where the CLog behaves as an HMM trained on individual events. Therefore, to benefit from the improvement of the log representations CLog is expected to perform well when applied to individual service components. Potential improvements in this regard may be achieved when reconstructing log dependencies (e.g., in OpenStack this can be achieved to a certain extent by tracking events that originate from the same request when analyzing the parameters of the logs). This can filter out certain logs, improving the homogeneity of the event groups. Notwithstanding, the ideas presented herein open new possibilities for how to most efficiently extract meaningful event groups with minimal information about the sequences.

Chapter 8

Conclusions

Modern IT systems play a major role in industrial infrastructure and human society. Their characteristics such as large complexity, fast evolution, geo-distributed development, and different working frequencies, among others, challenge the correctness and availability of service offerings because they increase failure proneness. To support the development and operation as a way to support the provisioning of correct service that can justifiably be trusted (system dependability), in this thesis, the automation of different tasks during development and operation was explored. An important enabler of automation is the possibility to externalise the IT system state via monitoring data such as system logs. The goal of this thesis is, therefore, to improve the automation of log-related development and operation activities by introducing intelligent methods and ideas that utilize system logs and other log-related data to support system dependability.

To directly support software development we proposed to automatically assess the quality of the logging code composition, as the quality of the log instructions determines the quality of the subsequent log analysis tasks. To that end, we developed an approach to automatically assess the quality of log instructions from an arbitrary software system with software logs written in natural language. We argued that the development of such an approach is challenged by the heterogeneity of the software systems, the unique writing styles of developers, and different programming languages. Based on the assumption of existing open-source systems with good logging quality, we identified a set of two automatically empirically testable quality properties independent of the programming language and the software system. Building on this observation, we introduced and formalized the problem of quantification of log instruction quality assessment. By leveraging our observations and the textual nature of the logs, we proposed a deep learning method for automatic model-driven log quality assessment as an intelligent tool to aid

the writing of log instructions. Next, we adapted an approach from explainable machine learning and used it to give augmented feedback for possible quality improvement. We performed an extensive evaluation to show the benefit of our approach. Notably, this method indirectly supports system operation.

To directly support system operation, in this thesis, we proposed novel methods for anomaly detection and classification using log data. By observing that the anomalies are reflected in different properties of the log data, i.e., single log lines or log sequences, we proposed two groups of methods accordingly. When analyzing single log lines the anomalies can be reflected either in the semantics of the static text or as abnormal parameter values. We contributed novel methods for the two. First, we proposed a novel method for semantic log anomaly detection. By analyzing log instructions from public software systems we showed that the log instructions contain rich anomaly-related information. The proposed method utilizes the data from the system of interest (target system data) alongside the extracted anomaly-related information as auxiliary data to learn anomaly-discriminative log representations. The evaluations showed that the auxiliary data helps to learn discriminative log properties and improve the semantic anomaly detection performance. By leveraging the *recurrence* assumption of anomalies, we further related the events with the associated log classes to support single line anomaly classification. Through extensive evaluation, we demonstrated the benefit of using external data in improving the generalization of log-based anomaly detection. Regarding the single line log analysis, we further contributed with a novel log parsing method that accurately disentangles the variable parameters from the static text. The parser is learned as a neural network model with the learning task formulated such that the presence of a word on a particular position in the generated log is conditioned on its context. We showed that the parser achieves high parsing accuracy while at the same time having robust performance across many datasets. The robustness is particularly important for parametric anomaly detection which relies on the correct extraction of the parameters. The extracted events and parameters are used to create a time series of numerical parameter values. Finally, by applying a parsimonious model on the univariate parameter values we showed how performance anomalies can be detected. Notably, the proposed parsing method can be used as a preprocessing step by other methods that require log parsing.

To address the sequential properties of the logs, we introduced a novel method for improving the sequential representation. We showed that by representing the log sequences as sequences of event groups the uncertainty in the overall log event sequence is reduced. We proposed a novel method that extracts event groups from a given event sequence. The method combines joint training of a neural network and a clustering method to learn the event groups. We used the learned representation of the log sequences in anomaly detection and classification. The evaluation results demonstrated that the modified input

representation does not degrade the performance for anomaly detection and classification, but can potentially improve other properties. Similarly, as in the case of single log analysis, we leveraged the *recurrence* assumption to enable anomaly classification.

One important aspect of some of the proposed methods is their data-centric nature. While the focus on the sequential log analysis is improving the learning method (i.e., it is model-centric), the log instruction quality assessment and the single line anomaly detection leverage publicly available data to build a model (i.e, they are data-centric). The encouraging results with a data-centric approach demonstrate that using publicly available data for log-specific tasks can be useful for addressing diverse development and operational activities. Therefore, introducing the data-centric view to other dependability tasks can be an important direction for further research. Logs are particularly useful in this regard, as they can be used to query public issue repositories (e.g., Jira).

While the thesis addresses the challenges of modern systems extensively, there is still room for further improvement. In the case of log quality evaluation, an extension of the set of quality properties is possible. For example, an important quality indicator for the logging code composition is the correct log instruction placement. It means that the logging instructions should not miss important events. At the same time, the logging should not be excessive as it increases overhead. As different programming languages have their specifics, a general automatic approach to assessing the log instruction placement requires accounting for the different complexities. Further improvement of the automatic quality assessment may require focusing on a specific programming language or system type instead of broadly targeting the problem in a programming language-agnostic manner. Similarly, when detecting and classifying anomalies using log data there is an existing performance gap between the proposed methods and the optimal performance value. Admittedly, striving for optimal performance may be a "fool's errand" because of the large set of internal and external events that cause failures. Nevertheless, potential improvement may arise when combining different data sources [8] or different log properties. In cases when the source code is available during operational activities, the method invocation chain can be used to filter out the irrelevant logs concerning the analyzed context. This can significantly reduce the uncertainty in the logs and potentially improve the detection performance.

Irrespective of the possible improvements that exist, the proposed methods address the characteristics of modern IT systems. As demonstrated through the experiments, the methods can aid the automation of system development and operation, and ultimately improve system dependability.

Appendix A

Online Services Failure Study

To study the impact of failures on real-world online services we conducted an empirical study on online service dependability. Specifically, we collected and analyzed data from publicly available online issue repositories. We examined one property of system dependability, i.e., availability. The availability is quantified as the mean time to failure (MTF) over a fixed time interval. In addition, we report the median time to failure (MeTF).

Experiment Design; To conduct our study we first collected data from online incidents. Here, we used the willingness of service providers for transparent reporting on the observed failures. For example, https://status.customerio.com provides information for the service *customerio* that is a service for an automatic messaging platform for marketers. It provides detailed descriptions of the failures (e.g., failure description, start time of failure, its end time etc.). We use the start and end times to calculate the failure duration. Many services offering this online failure reporting exist, e.g., statuspage.io, status.io, cachet.io, and statuscast.com [163]. Tola et al. [163] introduce a large set of publicly available links to such services. We used these links to collect all the failure events from January 2018 until January 2022, as the remaining time intervals were analyzed in the previous work. From the initial 96, we were able to retrieve data for a total of 70 services. The reason is that some of the services are deprecated. Note that our goal is not to replicate the study, but to obtain empirical evidence of the availability of services. One important notice is that failure reports can refer to the failure of a single component of the service, not the whole system itself. We do not distinguish between the severity of the failures.

Results and Discussions; We first analyze the service availability. Figure A.1 shows the available time of the services in the four years. We set five different availability ratios, i.e., 97%, 98%, 99%, 99.99% and 99.9999%. For example, the availability of 99.99% means that the services were not available for less than 3.5 hours in the 4 years. The results show that few (eight) services experience very high availability, while 13 experience availability

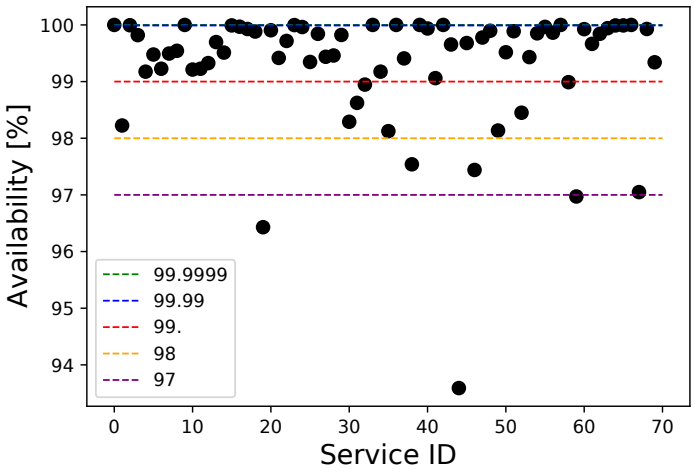


Figure A.1: Availability of the services in a period of four years.

greater than 99.99%. The majority of the services experience availability greater than 99% (in a total of 56). However, there are also services with lower availability, smaller than 98% and 97% (3 for each of the two). Figure A.2 characterizes the failure duration. Specifically, it shows the mean and median times of the failures. By the median failure time, it can be seen that the majority of the failures need a short time for resolution (around an hour). However, the discrepancy between the mean and the median time (2 hours on average) suggests that there exist failures that have a much longer duration

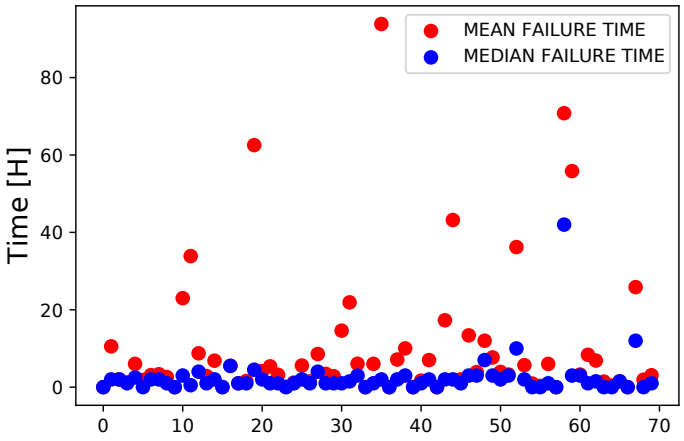


Figure A.2: Mean and median time to failure of the 70 services.

time to be resolved. The two sets of results demonstrate that failures can affect the availability of the systems to a large extent. Moreover, the failures can have non-trivial durations. These two observations motivate the need for developing tools for timely and potentially automatic failure detection and resolution.

Appendix B

Log Level Quality Assessment: Additional Evaluation

Log Level Problem Instances

The experiment on log level assessment presented in Chapter 5 shows that QuLog* performs better than the baselines on log level assignments. However, the results on accuracy across different systems, although good, indicate that there are incorrect assignments. The misclassifications can impair the practical usability of QuLog as if given at a large rate can overwhelm the system development process. To find a way to improve QuLog, we further study QuLog's misclassification types. To that end, we calculated the misclassification contingency table for the nine studied systems. Table B.1 gives the contingency table. Each cell shows the percentage of misclassification prediction rates for the three classes. 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%). We use this 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)

Table B.1: Log level misclassification contingency table (the averaging is done over nine software systems given in Table 5.1).

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

Table B.2: Performance scores on the task of log level assignment.

Scenario	IE	IWE	IW	WE
F1	0.88±0.03	0.73±0.03	0.68±0.06	0.61±0.04
Precision	0.88±0.02	0.72±0.03	0.75±0.04	0.69±0.09
Recall	0.89±0.05	0.73±0.03	0.62±0.08	0.56±0.07

and "error-warning" (EW). The goal with this is to find easily learnable scenarios, where the performance score can be increased, making the learned model more reliable. The examination of individual class pairs is practically relevant because different stakeholders have different expectations from logs [100]. For example, the operators usually examine the log levels "error" and "warning". Therefore, misclassifying an error event as "info" can hide important events from operators.

Experiment design; We considered QuLog* log level assignment approach because it is trained on many software projects and has more desirable system-agnostic properties. 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 use the extended knowledge base to randomly sample 60% of the repositories for training, 20% for validation and 20% of the projects for evaluation. To reduce the variance of the results due to the random repository selection, we repeated the sampling procedure several 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 [54].

Results and discussion; Table B.2 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 in the IE scenario, QuLog achieves the highest values on the F_1 score (average of 0.88), i.e., trades off the precision (0.88) and recall (0.89) quite well. Therefore, this model is more reliable for correctly assessing the "info" and "error" log instructions. The good performance is attributed to the larger differences in the vocabulary between the "error" and "info" log instructions. Therefore, this model is expected not to overwhelm developers with as many incorrect predictions as the remaining models. The scenarios of IWE and IW are harder to learn due to the overlapping vocabularies. Finally, the results on WE show that it is the hardest problem instance, as seen by the biggest performance drop in comparison to the IE scenario. The drop in performance can be attributed to the similarity in the usages of the two levels, i.e., both are used to describe potentially erroneous system states that have not yet resulted in failure which results in sharing similar vocabularies [100].

Bibliography

[1] van der Aalst and et al et. "Process Mining Manifesto". In: *Business Process Management Workshops*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 169–194.

[2] Alexander Acker. "Anomaly symptom recognition in distributed IT systems". PhD thesis. Berlin, Germany: Technical University Berlin, 2021.

[3] Mohiuddin Ahmed, Abdun Naser Mahmood, and Md. Rafiqul Islam. "A Survey of Anomaly Detection Techniques in Financial Domain". In: *Future Gener. Comput. Syst.* 55 (2016), pp. 278–288. DOI: 10.1016/j.future.2015.01.001.

[4] Algirdas Avizienis and Jean-Claude Laprie. "Dependable computing: From concepts to design diversity". In: *Proceedings of the IEEE* 74 (1986), pp. 629–638. DOI: 10.1109/PROC.1986.13527.

[5] Algirdas Avižienis, Jean-Claude. Laprie, Brian Randell, and Carl Landwehr. "Basic concepts and taxonomy of dependable and secure computing". In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (2004), pp. 11–33. DOI: 10.1109/TDSC.2004.2.

[6] Peter Bodik, Moises Goldszmidt, Armando Fox, Dawn B. Woodard, and Hans Andersen. "Fingerprinting the Datacenter: Automated Classification of Performance Crises". In: *Proceedings of the 5th European Conference on Computer Systems*. New York, NY, USA: ACM, 2010, pp. 111–124.

[7] Barry Boehm, Chris Abts, and Sunita Chulani. "Software development cost estimation approaches — A survey". In: *Annals of Software Engineering* 10.1 (2000), pp. 177–205. DOI: 10.1023/A:1018991717352.

[8] Jasmin **Bogatinovski** and Sasho Nedelkoski. "Multi-source Anomaly Detection in Distributed IT Systems". In: *Service-Oriented Computing – ICSOC 2020 Workshops*. Cham: Springer International Publishing, 2021, pp. 201–213. DOI: https://doi.org/10.1007/978-3-030-76352-7_22.

[9] Jasmin **Bogatinovski**, Sasho Nedelkoski, Alexander Acker, Jorge Cardoso, and Odej Kao. "QuLog: Data-Driven Approach for Log Instruction Quality Assessment". In: *30th International Conference on Program Comprehension (ICPC '22)*. USA: ACM, 2022. DOI: https://doi.org/10.1145/3524610.3527906.

[10] Jasmin **Bogatinovski**, Sasho Nedelkoski, Jorge Cardoso, and Odej Kao. "Self-Supervised Anomaly Detection from Distributed Traces". In: *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*. 2020, pp. 342–347. DOI: 10.1109/UCC48980.2020.00054.

[11] Jasmin **Bogatinovski**, Sasho Nedelkoski, Gjorgji Madjarov, Jorge Cardoso, and Odej Kao. "Leveraging Log instructions for Log-based Anomaly Detection". In: *2022 IEEE International Conference on Services Computing (SCC)*. 2022, pp. 321–326. DOI: 10.1109/SCC55611.2022.00053.

[12] Jasmin **Bogatinovski**, Sasho Nedelkoski, Li Wu, Jorge Cardoso, and Odej Kao. "Failure Identification from Unstable Log Data using Deep Learning". In: *22nd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. NY: IEEE Press, 2022. DOI: 10.1109/CCGrid54584.2022.00044. eprint: 1646836319158.

[13] Mihail Bogojeski, Leslie Vogt-Maranto, Mark E. Tuckerman, Klaus-Robert Müller, and Kieron Burke. "Quantum chemical accuracy from density functional approximations via machine learning". In: *Nature Communications* 11 (2020). DOI: 10.1038/s41467-020-19093-1.

[14] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. "Enriching Word Vectors with Subword Information". In: *Transactions of the Association for Computational Linguistics* 5 (2017), pp. 135–146. DOI: 10.1162/tacl_a_00051.

[15] Richard J. Bolton and David J. Hand. "Statistical Fraud Detection: A Review". In: *Statistical Science* 17 (2002), pp. 235–255. DOI: 10.1214/ss/1042727940.

[16] Leo Breiman. "Random Forests". In: *Machine Learning* 45.1 (2001), pp. 5–32.

[17] 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. "Language Models are Few-Shot Learners". In: *Advances in Neural Information Processing Systems*. Vol. 33. Curran Associates, Inc., 2020, pp. 1877–1901.

[18] Nadia Burkart and Marco F. Huber. "A Survey on the Explainability of Supervised Machine Learning". In: *J. Artif. Int. Res.* 70 (2021), pp. 245–317. DOI: 10.1613/jair.1.12228. URL: https://doi.org/10.1613/jair.1.12228.

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

[20] Jorge Cardoso, Jasmin **Bogatinovski**, and Sasho Nedelkoski. *Distributed Trace Anomaly Detection with Self-Attention based Deep Learning*. Approved by the European Patent Office, WO2022053163A1. 2022.

[21] Olmo Cerri, Thong Q. Nguyen, Maurizio Pierini, Maria Spiropulu, and Jean-Roch Vlimant. "Variational autoencoders for new physics mining at the Large Hadron Collider". In: *Journal of High Energy Physics* 2019 (2019), p. 36. DOI: 10.1007/JHEP05(2019)036.

[22] Varun Chandola, Arindam Banerjee, and Vipin Kumar. "Anomaly Detection: A Survey". In: *ACM Comput. Surv.* 41 (2009). DOI: 10.1145/1541880.1541882.

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

[24] Boyuan Chen and Zhen Ming Jiang. "Characterizing and Detecting Anti-Patterns in the Logging Code". In: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 2017, pp. 71–81. DOI: 10.1109/ICSE.2017.15.

[25] Boyuan Chen and Zhen Ming (Jack) Jiang. "A Survey of Software Log Instrumentation". In: *ACM Comput. Surv.* 54 (2021). DOI: 10.1145/3448976.

[26] Boyuan Chen and Zhen Ming (Jack) Jiang. "Extracting and studying the Logging-Code-Issue-Introducing changes in Java-based large-scale open source software systems". In: *Empirical Software Engineering* 24 (2019), pp. 2285–2322. DOI: 10.1007/s10664-019-09690-0.

[27] Boyuan Chen and Zhen Ming (Jack) Jiang. "Studying the Use of Java Logging Utilities in the Wild". In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 397–408. DOI: 10.1145/3377811.3380408.

[28] M. Chen, A.X. Zheng, J. Lloyd, M.I. Jordan, and E. Brewer. "Failure diagnosis using decision trees". In: *International Conference on Autonomic Computing, 2004. Proceedings.* 2004, pp. 36–43. DOI: 10.1109/ICAC.2004.1301345.

[29] Rui Chen, Shenglin Zhang, Dongwen Li, Yuzhe Zhang, Fangrui Guo, Weibin Meng, Dan Pei, Yuzhi Zhang, Xu Chen, and Yuqing Liu. "LogTransfer: Cross-System Log Anomaly Detection for Software Systems with Transfer Learning". In: *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. 2020, pp. 37–47. DOI: 10.1109/ISSRE5003.2020.00013.

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

[31] Zhuangbin Chen, Jinyang Liu, Wenwei Gu, Yuxin Su, and Michael R. Lyu. "Experience Report: Deep Learning-based System Log Analysis for Anomaly Detection". In: *CoRR* 2107.05908 (2021). URL: https://arxiv.org/abs/2107.05908.

[32] R. Chillarege, I.S. Bhandari, J.K. Chaar, M.J. Halliday, D.S. Moebus, B.K. Ray, and M.-Y. Wong. "Orthogonal defect classification-a concept for in-process measurements". In: *IEEE Transactions on Software Engineering* 18 (1992), pp. 943–956. DOI: 10.1109/32.177364.

[33] Sunita Chulani and Barry Boehm. *Modeling Software Defect Introduction and Removal*. Tech. rep. 1999.

[34] Corinna Cortes and Vladimir Vapnik. "Support-vector networks". In: *Machine Learning* 20 (1995), pp. 273–297.

[35] Domenico Cotroneo, Luigi De Simone, Pietro Liguori, and Roberto Natella. "Enhancing the analysis of software failures in cloud computing systems with deep learning". In: *Journal of Systems and Software* 181 (2021), p. 111043. DOI: https://doi.org/10.1016/j.jss.2021.111043. URL: https://www.sciencedirect.com/science/article/pii/S0164121221001400.

[36] Domenico Cotroneo, Luigi De Simone, Pietro Liguori, Roberto Natella, and Nematollah Bidokhti. "How bad can a bug get? an empirical analysis of software failures in the OpenStack cloud computing platform". In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2019, pp. 200–211.

[37] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory (Wiley Series in Telecommunications and Signal Processing)*. USA: Wiley-Interscience, 2006. ISBN: 0471241954.

[38] Yingnong Dang, Qingwei Lin, and Peng Huang. "AIOps: Real-World Challenges and Research Innovations". In: *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 2019, pp. 4–5. DOI: 10.1109/ICSE-Companion.2019.00023.

[39] Alex Davies, Petar Veličković, Lars Buesing, Sam Blackwell, Daniel Zheng, Nenad Tomašev, Richard Tanburn, Peter Battaglia, Charles Blundell, András Juhász, Marc Lackenby, Geordie Williamson, Demis Hassabis, and Pushmeet Kohli. "Advancing mathematics by guiding human intuition with AI". In: *Nature* 600.7887 (2021), pp. 70–74. DOI: 10.1038/s41586-021-04086-x.

[40] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. "Bert: Pre-training of deep bidirectional transformers for language understanding". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Minneapolis, Minnesota: Association for Computational Linguistics, 2019, pp. 4171–4186. DOI: 10.18653/v1/N19-1423.

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

[42] Zishuo Ding, Heng Li, and Weiyi Shang. "LoGenText: Automatically Generating Logging Texts Using Neural Machine Translation". In: *Proceedings of the 9th IEEE International Conference on Software Analysis, Evolution and Reengineering*. SANER '22. IEEE. 2022.

[43] Min Du and Feifei Li. "Spell: Streaming Parsing of System Event Logs". In: *2016 IEEE 16th International Conference on Data Mining (ICDM)*. 2016, pp. 859–864. DOI: 10.1109/ICDM.2016.0103.

[44] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. "DeepLog: Anomaly Detection and Diagnosis from System Logs through Deep Learning". In: *Proceedings of the 2017 ACM SIGSAC*. New York, NY, USA: Association for Computing Machinery, 2017, pp. 1285–1298.

[45] F.Y. Edgeworth. "XLI. On discordant observations". In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 23 (1887), pp. 364–375. DOI: 10.1080/14786448708628471.

[46] Edward Finegan. *Language: Its structure and use*. 7th ed. Florence, AL: Cengage Learning, 2014, p. 289.

[47] Wendy D. Fisher, Tracy K. Camp, and Valeria V. Krzhizhanovskaya. "Anomaly detection in earth dam and levee passive seismic data using support vector machines and automatic feature selection". In: *Journal of Computational Science* 20 (2017), pp. 143–153. DOI: https://doi.org/10.1016/j.jocs.2016.11.016.

[48] Forecast. *Cisco visual networking index: global mobile data traffic forecast update*. 2017. URL: https://branden.biz/wp-content/uploads/2019/05/white-paper-c11-738429.pdf.

[49] The Apache Software Foundation. *Logging Service Project*. Appache. 2022. URL: https://logging.apache.org/.

[50] Yoav Freund and Robert E. Schapire. "A Short Introduction to Boosting". In: *Proc. of the 16 International Joint Conference on Artificial Intelligence*. 1999, pp. 1401–1406.

[51] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. "Execution Anomaly Detection in Distributed Systems through Unstructured Log Analysis". In: *2009 Ninth IEEE International Conference on Data Mining*. 2009, pp. 149–158. DOI: 10.1109/ICDM.2009.60.

[52] Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. "Where Do Developers Log? An Empirical Study on Logging Practices in Industry". In: *Companion Proceedings of the 36th International Conference on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2014, pp. 24–33. DOI: 10.1145/2591062.2591175.

[53] Peter Garraghan, Paul Townend, and Jie Xu. "An Empirical Failure-Analysis of a Large-Scale Cloud Computing Environment". In: *2014 IEEE 15th International Symposium on High-Assurance Systems Engineering*. IEEE Press, 2014, pp. 113–120. DOI: 10.1109/HASE.2014.24.

[54] Eva Gibaja and Sebastián Ventura. "A Tutorial on Multilabel Learning". In: *ACM Computing Surveys* 47.3 (2015), 52:1–52:38.

[55] Federico Girosi, Michael Jones, and Tomaso Poggio. "Regularization Theory and Neural Networks Architectures". In: *Neural Computation* (1995), pp. 219–269. DOI: 10.1162/neco.1995.7.2.219.

[56] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. URL: http://www.deeplearningbook.org.

[57] Piotr S. Gromski, Alon B. Henson, Jarosław M. Granda, and Leroy Cronin. "How to explore chemical space using algorithms and automation". In: *Nature Reviews Chemistry* 3 (2019), pp. 119–128. DOI: 10.1038/s41570-018-0066-y.

[58] Hossein Hamooni, Biplob Debnath, Jianwu Xu, Hui Zhang, Guofei Jiang, and Abdullah Mueen. "LogMine: Fast Pattern Recognition for Log Analytics". In: *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. 2016, pp. 1573–1582.

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

[60] Xiao Han and Shuhan Yuan. "Unsupervised Cross-System Log Anomaly Detection via Domain Adaptation". In: *Proceedings of the 30th ACM International Conference on Information and Knowledge Management*. 2021, pp. 3068–3072. DOI: 10.1145/3459637.3482209.

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

[62] Shayan Hashemi and Mika Mäntylä. *OneLog: Towards End-to-End Training in Software Log Anomaly Detection*. Last Access 20 Jan 2022. 2021. URL: https://arxiv.org/abs/2104.07324.

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

[64] Mehran Hassani, Weiyi Shang, Emad Shihab, and Nikolaos Tsantalis. "Studying and Detecting Log-Related Issues". In: *Empirical Software Engineering* 23 (2018), pp. 3248–3280. DOI: 10.1007/s10664-018-9603-z.

[65] Mehran Hassani, Weiyi Shang, Emad Shihab, and Nikolaos Tsantalis. "Studying and detecting log-related issues". In: *Empirical Software Engineering* 23 (2018), pp. 3248–3280. DOI: 10.1007/s10664-018-9603-z.

[66] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. New York, NY, USA: Springer New York Inc., 2001.

[67] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. New York, NY, USA: Springer New York Inc., 2001.

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

[69] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R. Lyu. "Drain: An Online Log Parsing Approach with Fixed Depth Tree". In: *2017 IEEE International Conference on Web Services*. NY, USA: Curran Associates, 2017, pp. 33–40.

[70] Shilin He, Pinjia He, Zhuangbin Chen, Tianyi Yang, Yuxin Su, and Michael R. Lyu. "A Survey on Automated Log Analysis for Reliability Engineering". In: *ACM Comput. Surv.* 54 (2021).

[71] Dan Hendrycks, Mantas Mazeika, and Thomas G. Dietterich. "Deep Anomaly Detection with Outlier Exposure". In: (2018). eprint: 1812.04606.

[72] Marc Henrion, Daniel J. Mortlock, David J. Hand, and Axel Gandy. "Classification and Anomaly Detection for Astronomical Survey Data". In: *Astrostatistical Challenges for the New Astronomy*. New York, NY: Springer New York, 2013, pp. 149–184. DOI: 10.1007/978-1-4614-3508-2_8.

[73] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. *Distilling the Knowledge in a Neural Network*. 2015. DOI: 10.48550/ARXIV.1503.02531. URL: https://arxiv.org/abs/1503.02531.

[74] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* 9 (1997), pp. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735.

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

[76] Shaohan Huang, Yi Liu, Carol Fung, Rong He, Yining Zhao, Hailong Yang, and Zhongzhi Luan. "HitAnomaly: Hierarchical Transformers for Anomaly Detection in System Log". In: *IEEE Transactions on Network and Service Management* 17 (2020), pp. 2064–2076. DOI: 10.1109/TNSM.2020.3034647.

[77] Mark Hung. *Leading the iot, gartner insights on how to lead in a connected world*. 2017. URL: https://www.gartner.com/imagesrv/books/iot/iotEbook_digital.pdf.

[78] Zhouyang Jia, Shanshan Li, Xiaodong Liu, Xiangke Liao, and Yunhuai Liu. "SMARTLOG: Place error log statement by deep understanding of log intention". In: *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2018, pp. 61–71. DOI: 10.1109/SANER.2018.8330197.

[79] Zhen Ming Jiang, Ahmed E. Hassan, Parminder Flora, and Gilbert Hamann. "Abstracting Execution Logs to Execution Events for Enterprise Applications (Short Paper)". In: *2008 The Eighth International Conference on Quality Software*. 2008, pp. 181–186. DOI: 10.1109/QSIC.2008.50.

[80] Zhen Ming Jiang, Ahmed E. Hassan, Gilbert Hamann, and Parminder Flora. "Automatic identification of load testing problems". In: *2008 IEEE International Conference on Software Maintenance*. 2008, pp. 307–316. DOI: 10.1109/ICSM.2008.4658079.

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

[82] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, Alex Bridgland, Clemens Meyer, Simon A. A. Kohl, Andrew J. Ballard, Andrew Cowie, Bernardino Romera-Paredes, Stanislav Nikolov, Rishub Jain, Jonas Adler, Trevor Back, Stig Petersen, David Reiman, Ellen Clancy, Michal Zielinski, Martin Steinegger, Michalina Pacholska, Tamas Berghammer, Sebastian Bodenstein, David Silver, Oriol Vinyals, Andrew W. Senior, Koray Kavukcuoglu, Pushmeet Kohli, and Demis Hassabis. "Highly accurate protein structure prediction with AlphaFold". In: *Nature* 596 (2021), pp. 583–589. DOI: 10.1038/s41586-021-03819-2.

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

[84] Shima Keiichi. *Length matters: Clustering system log messages using length of words*. 2016. URL: https://arxiv.org/pdf/1611.03213.pdf.

[85] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. "Aspect-oriented programming". In: *ECOOP'97 — Object-Oriented Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 220–242.

[86] Taeyoung Kim, Suntae Kim, Cheol-Jung Yoo, Soohwan Cho, and Sooyong Park. "An Automatic Approach to Validating Log Levels in Java". In: *25th Asia-Pacific Software Engineering Conference (APSEC)*. 2018, pp. 623–627. DOI: 10.1109/APSEC.2018.00078.

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

[88] Max Landauer, Sebastian Onder, Florian Skopik, and Markus Wurzenberger. *Deep Learning for Anomaly Detection in Log Data: A Survey*. 2022. DOI: 10.48550/ARXIV.2207.03820.

[89] J. C. Laprie. *Dependability: Basic Concepts and Terminology*. Ed. by J. C. Laprie. Vienna: Springer Vienna, 1992, pp. 3–245. ISBN: 978-3-7091-9170-5. DOI: 10.1007/978-3-7091-9170-5_1. URL: https://doi.org/10.1007/978-3-7091-9170-5_1.

[90] Jean-Claude Laprie. "DEPENDABLE COMPUTING AND FAULT TOLERANCE : CONCEPTS AND TERMINOLOGY". In: *Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995*. 1995, pp. 2–12. DOI: 10.1109/FTCSH.1995.532603.

[91] Van-Hoang Le and Hongyu Zhang. "Log-based Anomaly Detection with Deep Learning: How Far Are We?" In: *2022 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 2022, pp. 1356–1367. DOI: 10.1145/3510003.3510155.

[92] H. Li, W. Shang, B. Adams, M. Sayagh, and A. E. Hassan. "A Qualitative Study of the Benefits and Costs of Logging from Developers' Perspectives". In: *IEEE Transactions on Software Engineering* (2020), pp. 1–17. DOI: 10.1109/TSE.2020.2970422.

[93] Heng Li, Tse-Hsun (Peter) Chen, Weiyi Shang, and Ahmed E. Hassan. "Studying software logging using topic models". In: *Empirical Software Engineering* 23.5 (2018), pp. 2655–2694. DOI: 10.1007/s10664-018-9595-8.

[94] Heng Li, Weiyi Shang, and Ahmed E. Hassan. "Which Log Level Should Developers Choose for a New Logging Statement?" In: *Empirical Software Engineering* 22.4 (2017), pp. 1684–1716.

[95] Heng Li, Weiyi Shang, and Ahmed E. Hassan. "Which Log Level Should Developers Choose for a New Logging Statement?" In: *Empir. Softw. Eng.* 22 (2017), pp. 1684–1716.

[96] Shanshan Li, Xu Niu, Zhouyang Jia, Xiangke Liao, Ji Wang, and Tao Li. "Guiding log revisions by learning from software evolution history". In: *Empirical Software Engineering* 25 (2020), pp. 2302–2340. DOI: 10.1007/s10664-019-09757-y.

[97] Tao Li, Yexi Jiang, Chunqiu Zeng, Bin Xia, Zheng Liu, Wubai Zhou, Xiaolong Zhu, Wentao Wang, Liang Zhang, Jun Wu, Li Xue, and Dewei Bao. "FLAP: An End-to-End Event Log Analysis Platform for System Management". In: *Proceedings of the 23rd ACM SIGKDD International Conference on KDD*. New York, NY, USA: Association for Computing Machinery, 2017, pp. 1547–1556. DOI: 10.1145/3097983.3098022.

[98] Xiaoyun Li, Pengfei Chen, Linxiao Jing, Zilong He, and Guangba Yu. "SwissLog: Robust and Unified Deep Learning Based Log Anomaly Detection for Diverse Faults". In: *2020 IEEE 31st ISSRE*. 2020, pp. 92–103. DOI: 10.1109/ISSRE5003.2020.00018.

[99] Zhenhao Li, Tse-Hsun Chen, and Weiyi Shang. "Where Shall We Log? Studying and Suggesting Logging Locations in Code Blocks". In: *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2020, pp. 361–372.

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

[101] Pietro Liguori, Erfan Al-Hossami, Domenico Cotroneo, Roberto Natella, Bojan Cukic, and Samira Shaikh. "Can we generate shellcodes via natural language? An empirical study". In: *Automated Software Engineering* 29 (2022). DOI: 10.1007/s10515-022-00331-3.

[102] Qingwei Lin, Hongyu Zhang, Jian-Guang Lou, Yu Zhang, and Xuewei Chen. "Log Clustering Based Problem Identification for Online Service Systems". In: *Proceedings of the 38th ICSE*. New York, NY, USA: Association for Computing Machinery, 2016, pp. 102–111. DOI: 10.1145/2889160.2889232.

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

[104] B. Liu, Y. Dai, X. Li, W.S. Lee, and P.S. Yu. "Building text classifiers using positive and unlabeled examples". In: *Third IEEE International Conference on Data Mining*. 2003, pp. 179–186. DOI: 10.1109/ICDM.2003.1250918.

[105] Jinping Liu, Yuming Zhou, Yibiao Yang, Hongmin Lu, and Baowen Xu. "Code Churn: A Neglected Metric in Effort-Aware Just-in-Time Defect Prediction". In: *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 2017, pp. 11–19. DOI: 10.1109/ESEM.2017.8.

[106] Weiyang Liu, Yan-Ming Zhang, Xingguo Li, Zhiding Yu, Bo Dai, Tuo Zhao, and Le Song. "Deep Hyperspherical Learning". In: *Proceedings of the 31st International Conference on NeurIPS*. Red Hook, NY, USA: Curran Associates Inc., 2017, pp. 3953–3963.

[107] Xiaotong Liu, Tong Jia, Ying Li, Hao Yu, Yang Yue, and Chuanjia Hou. "Automatically Generating Descriptive Texts in Logging Statements: How Far Are We?" In: *Programming Languages and Systems*. Cham: Springer International Publishing, 2020, pp. 251–269.

[108] Zhongxin Liu, Xin Xia, David Lo, Zhenchang Xing, Ahmed E. Hassan, and Shanping Li. "Which Variables Should I Log?" In: *IEEE Transactions on Software Engineering* 47.9 (2021), pp. 2012–2031. DOI: 10.1109/TSE.2019.2941943.

[109] Zhongxin Liu, Xin Xia, David Lo, Zhenchang Xing, Ahmed E. Hassan, and Shanping Li. "Which Variables Should I Log?" In: *IEEE Transactions on Software Engineering* 47 (2021), pp. 2012–2031. DOI: 10.1109/TSE.2019.2941943.

[110] Felipe Lopez, Miguel Saez, Yuru Shao, Efe C. Balta, James Moyne, Z. Morley Mao, Kira Barton, and Dawn Tilbury. "Categorization of Anomalies in Smart Manufacturing Systems to Support the Selection of Detection Mechanisms". In: *IEEE Robotics and Automation Letters* 2 (2017), pp. 1885–1892. DOI: 10.1109/LRA.2017.2714135.

[111] Siyang Lu, BingBing Rao, Xiang Wei, Byungchul Tak, Long Wang, and Liqiang Wang. "Log-based Abnormal Task Detection and Root Cause Analysis for Spark". In: *2017 IEEE International Conference on Web Services (ICWS)*. 2017, pp. 389–396. DOI: 10.1109/ICWS.2017.135.

[112] Siyang Lu, Xiang Wei, Yandong Li, and Liqiang Wang. "Detecting Anomaly in Big Data System Logs Using Convolutional Neural Network". In: *IEEE 16th Conf on Dependable, Autonomic and Secure Computing*. 2018, pp. 151–158. DOI: 10.1109/DASC/PiCom/DataCom/CyberSciTec.2018.00037.

[113] Scott Lundberg. *SHAP Github Implementation*. GitHub. 2019. URL: https://github.com/slundberg/shap.

[114] Scott M. Lundberg, Gabriel Erion, Hugh Chen, Alex DeGrave, Jordan M. Prutkin, Bala Nair, Ronit Katz, Jonathan Himmelfarb, Nisha Bansal, and Su-In Lee. "From local explanations to global understanding with explainable AI for trees". In: *Nature Machine Intelligence* 2.1 (2020), pp. 56–67.

[115] Adetokunbo A.O. Makanju, A. Nur Zincir-Heywood, and Evangelos E. Milios. "Clustering Event Logs Using Iterative Partitioning". In: *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. New York, NY, USA: Association for Computing Machinery, 2009, pp. 1255–1264. DOI: 10.1145/1557019.1557154.

[116] Sole Marc, Muntes-Mulero Victor, Ibrahim Rana Annie, and Estrada Giovani. *Survey on Models and Techniques for Root-Cause Analysis*. 2017. URL: http://arxiv.org/abs/1701.08546.

[117] Weibin Meng and et al. "LogAnomaly: Unsupervised Detection of Sequential and Quantitative Anomalies in Unstructured Logs". In: *Proceedings of the 28, IJCAI-19*. IJCAI, 2019, pp. 4739–4745.

[118] Weibin Meng, Ying Liu, Shenglin Zhang, Dan Pei, Hui Dong, Lei Song, and Xulong Luo. "Device-Agnostic Log Anomaly Classification with Partial Labels". In: *Proc of 26th International Symposium on Quality of Service*. 2018, pp. 1–6.

[119] Weibin Meng, Ying Liu, Shenglin Zhang, Federico Zaiter, Yuzhe Zhang, Yuheng Huang, Zhaoyang Yu, Yuzhi Zhang, Lei Song, Ming Zhang, and Dan Pei. "LogClass: Anomalous Log Identification and Classification With Partial Labels". In: *IEEE Trans. Netw.* 18 (2021), pp. 1870–1884.

[120] Salma Messaoudi, Annibale Panichella, Domenico Bianculli, Lionel Briand, and Raimondas Sasnauskas. "A Search-Based Approach for Accurate Identification of Log Message Formats". In: *Proceedings of the 26th Conference on Program Comprehension*. New York, NY, USA: Association for Computing Machinery, 2018, pp. 167–177. DOI: 10.1145/3196321.3196340.

[121] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. "Distributed Representations of Words and Phrases and their Compositionality". In: *Advances in Neural Information Processing Systems*. Vol. 26. Curran Associates, Inc., 2013. URL: https://proceedings.neurips.cc/paper/2013/file/9aa42b31882ec039965f3c4923ce901b-Paper.pdf.

[122] Thomas M. Mitchell. *Machine Learning*. New York, USA: McGraw-Hill, 1997.

[123] Masayoshi Mizutani. "Incremental Mining of System Log Format". In: *2013 IEEE International Conference on Services Computing*. 2013, pp. 595–602. DOI: 10.1109/SCC.2013.73.

[124] N.R. Murphy, B. Beyer, C. Jones, and J. Petoff. *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, 2016. Chap. Chapter 6. ISBN: 9781491951170. URL: https://books.google.de/books?id=tYrPCwAAQBAJ.

[125] Meiyappan Nagappan and Mladen A Vouk. "Abstracting log lines to log event types for mining software system logs". In: *Proceedings of the 2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. 2010, pp. 114–117.

[126] Animesh Nandi, Atri Mandal, Shubham Atreja, Gargi B Dasgupta, and Subhrajit Bhattacharya. "Anomaly detection using program control flow graph mining from execution logs". In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2016, pp. 215–224.

[127] Sasho Nedelkoski, Jasmin **Bogatinovski**, Alexander Acker, Jorge Cardoso, and Odej Kao. "Self-Attentive Classification-Based Anomaly Detection in Unstructured Logs". In: *2020 IEEE International Conference on Data Mining (ICDM)*. 2020, pp. 1196–1201. DOI: 10.1109/ICDM50108.2020.00148.

[128] Sasho Nedelkoski, Jasmin **Bogatinovski**, Alexander Acker, Jorge Cardoso, and Odej Kao. "Self-supervised Log Parsing". In: *Machine Learning and Knowledge Discovery in Databases: Applied Data Science Track*. Cham: Springer International Publishing, 2021, pp. 122–138. DOI: https://doi.org/10.1007/978-3-030-67667-4_8.

[129] Sasho Nedelkoski, Jasmin **Bogatinovski**, Ajay Kumar Mandapati, Soeren Becker, Jorge Cardoso, and Odej Kao. "Multi-source Distributed System Data for AI-Powered Analytics". In: *Service-Oriented and Cloud Computing*. Cham: Springer International Publishing, 2020, pp. 161–176. DOI: https://doi.org/10.1007/978-3-030-44769-4_13.

[130] Paolo Notaro, Jorge Cardoso, and Michael Gerndt. "A Survey of AIOps Methods for Failure Management". In: *ACM Trans. Intell. Syst. Technol.* 12 (2021). DOI: 10.1145/3483424.

[131] Adam Oliner, Archana Ganapathi, and Wei Xu. "Advances and Challenges in Log Analysis: Logs Contain a Wealth of Information for Help in Managing Systems." In: *Queue* 9 (2011), pp. 30–40. DOI: 10.1145/2076796.2082137.

[132] Adam Oliner and Jon Stearley. "What Supercomputers Say: A Study of Five System Logs". In: *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. Los Alamitos, CA, USA: IEEE Computer Society, 2007, pp. 575–584.

[133] OpenSource. *logging module Python*. 2022. URL: https://docs.python.org/3/library/logging.html.

[134] OpenSource. *spdlog C++ Library*. 2022. URL: https://github.com/gabime/spdlog.

[135] Harold Ott, Jasmin **Bogatinovski**, Alexander Acker, Sasho Nedelkoski, and Odej Kao. "Robust and Transferable Anomaly Detection in Log Data using Pre-Trained Language Models". In: *2021 IEEE/ACM International Workshop on Cloud Intelligence (CloudIntelligence)*. 2021, pp. 19–24. DOI: 10.1109/CloudIntelligence52565.2021.00013.

[136] Guansong Pang, Chunhua Shen, Longbing Cao, and Anton Van Den Hengel. "Deep Learning for Anomaly Detection: A Review". In: *ACM Comput. Surv.* 54 (2021). DOI: 10.1145/3439950.

[137] Antonio Pecchia, Marcello Cinque, Gabriella Carrozza, and Domenico Cotroneo. "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. New York, USA: IEEE Press, 2015, pp. 169–178.

[138] McGuthrie Peter. *Tesla Sued in California, Lawsuit Alleges Autopilot Responsible for Fatality*. 2021. URL: https://teslanorth.com/2021/07/07/tesla-sued-in-california-lawsuit-alleges-autopilot-responsible-for-fatality/.

[139] Cuong Pham, Long Wang, Byung Chul Tak, Salman Baset, Chunqiang Tang, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. "Failure Diagnosis for Distributed Systems Using Targeted Fault Injection". In: *IEEE Trans. Parallel Distrib. Syst.* 28 (2017), pp. 503–516. DOI: 10.1109/TPDS.2016.2575829.

[140] Friedrich Pukelsheim. "The Three Sigma Rule". In: *The American Statistician* (1994), pp. 88–91. DOI: 10.1080/00031305.1994.10476030.

[141] QOS. *Simple Logging Faced for Java*. QOS. 2022. URL: https://www.slf4j.org/.

[142] J. R. Quinlan. "Induction of decision trees". In: *Mach. Learn.* 1 (1986), pp. 81–106.

[143] Julien Rabatel, Sandra Bringay, and Pascal Poncelet. "Anomaly detection in monitoring sensor data for preventive maintenance". In: *Expert Systems with Applications* 38 (2011), pp. 7003–7015. DOI: https://doi.org/10.1016/j.eswa.2010.12.014.

[144] Ariel Rabkin, Wei Xu, Avani Wildani, Armando Fox, David Patterson, and Randy Katz. "A Graphical Representation for Identifier Structure in Logs". In: *Proceedings of the 2010 Workshop on Managing Systems via Log Analysis and Machine Learning Techniques*. USA: USENIX Association, 2010.

[145] Weischedel Ralph, Palmer Martha, Marcus Mitchell, Hovy Eduard, Pradhan Sameer, Ramshaw Lance, Xue Nianwen, Taylor Ann, Kaufman Jeff, Franchini Michelle, El-Bachouti Mohammed, Belvin Robert, and Houston Ann. *OntoNotes Release 5.0*. DOI: https://doi.org/10.35111/xmhb-2b84. URL: https://catalog.ldc.upenn.edu/LDC2013T19.

[146] Lukas Ruff, Jacob R. Kauffmann, Robert A. Vandermeulen, Grégoire Montavon, Wojciech Samek, Marius Kloft, Thomas G. Dietterich, and Klaus-Robert Müller. "A Unifying Review of Deep and Shallow Anomaly Detection". In: *Proceedings of the IEEE* 109.5 (2021), pp. 756–795.

[147] Lukas Ruff, Robert Vandermeulen, Nico Goernitz, Lucas Deecke, Shoaib Ahmed Siddiqui, Alexander Binder, Emmanuel Müller, and Marius Kloft. "Deep One-Class Classification". In: *Proceedings of the 35th International Conference on Machine Learning*. PMLR, 2018, pp. 4393–4402.

[148] Lukas Ruff, Robert A. Vandermeulen, Nico Görnitz, Alexander Binder, Emmanuel Müller, Klaus-Robert Müller, and Marius Kloft. "Deep Semi-Supervised Anomaly Detection". In: *International Conference on Learning Representations*. 2020. URL: https://openreview.net/forum?id=HkgH0TEYwH.

[149] Felix Salfner, Maren Lenk, and Miroslaw Malek. "A Survey of Online Failure Prediction Methods". In: *ACM Comput. Surv.* 42.3 (2010). DOI: 10.1145/1670679.1670680. URL: https://doi.org/10.1145/1670679.1670680.

[150] Felix Salfner, Steffen Tschirpke, and Miroslaw Malek. "Comprehensive logfiles for autonomic systems". In: *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.* 2004. DOI: 10.1109/IPDPS.2004.1303243.

[151] Oxley Michael. *"Sarbanes-Oxley Act of 2002"*. USA Senat. 2002. URL: https://www.govtrack.us/congress/bills/107/hr3763.

[152] Weiyi Shang, Zhen Ming Jiang, Bram Adams, Ahmed E. Hassan, Michael W. Godfrey, Mohamed Nasser, and Parminder Flora. "An exploratory study of the evolution of communicated information about the execution of large software systems". In: *Journal of Software: Evolution and Process* 26 (2014), pp. 3–26. DOI: https://doi.org/10.1002/smr.1579. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.1579.

[153] Weiyi Shang, Meiyappan Nagappan, and Ahmed E. Hassan. "Studying the relationship between logging characteristics and the code quality of platform software". In: *Empirical Software Engineering* 20 (2015), pp. 1–27. DOI: 10.1007/s10664-013-9274-8.

[154] He Shilin, Zhu Jieming, He Pinjia, and Lyu Michael R. "Experience Report: System Log Analysis for Anomaly Detection". In: *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*. River Side, USA: IEEE Press, 2016, pp. 207–218.

[155] R. Short and K. Fukunaga. "The optimal distance measure for nearest neighbor classification". In: *IEEE Transactions on Information Theory* 27 (1981), pp. 622–627. DOI: 10.1109/TIT.1981.1056403.

[156] Julius Sim and Chris C Wright. "The Kappa Statistic in Reliability Studies: Use, Interpretation, and Sample Size Requirements". In: *Physical Therapy* 85 (2005), pp. 257–268.

[157] Cindy Sridharan. *Distributed Systems Observability*. O'Reilly Media, Inc., 2018. Chap. 4. ISBN: 9781492033424.

[158] Ching Y. Suen. "n-Gram Statistics for Natural Language Understanding and Text Processing". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (1979), pp. 164–172. DOI: 10.1109/TPAMI.1979.4766902.

[159] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms (2nd Edition)*. USA: Prentice-Hall, Inc., 2006. ISBN: 0132392275.

[160] Liang Tang, Tao Li, and Chang-Shing Perng. "LogSig: Generating System Events from Raw Textual Logs". In: *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*. New York, NY, USA: Association for Computing Machinery, 2011, pp. 785–794. DOI: 10.1145/2063576.2063690.

[161] Wilson L Taylor. "Cloze procedure: A new tool for measuring readability". In: *Journalism quarterly* 30 (1953), pp. 415–433.

[162] Robert Tibshirani and Trevor Hastie. "Outlier sums for differential gene expression analysis." In: *Biostatistics (Oxford, England)* 8 (2007), pp. 2–8. DOI: https://doi.org/10.1093/biostatistics/kxl005.

[163] Besmir Tola, Yuming Jiang, and Bjarne E. Helvik. "Failure process characteristics of cloud-enabled services". In: *2017 9th International Workshop on Resilient Networks Design and Modeling (RNDM)*. 2017, pp. 1–7. DOI: 10.1109/RNDM.2017.8093033.

[164] Risto Vaarandi. "A data clustering algorithm for mining patterns from event logs". In: *Proceedings of the 3rd IEEE Workshop on IP Operations Management (IPOM 2003) (IEEE Cat. No.03EX764)*. 2003, pp. 119–126. DOI: 10.1109/IPOM.2003.1251233.

[165] Joaquin Vanschoren. "Meta-Learning: A Survey". In: (2018). eprint: 1810.03548. URL: http://arxiv.org/abs/1810.03548.

[166] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, undefinedukasz Kaiser, and Illia Polosukhin. "Attention is All You Need". In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. Red Hook, NY, USA: Curran Associates Inc., 2017, pp. 6000–6010.

[167] Yi Wan, Yilin Liu, Dong Wang, and Yujin Wen. "GLAD-PAW: Graph-Based Log Anomaly Detection by Position Aware Weighted Graph Attention Network". In: *Advances in Knowledge Discovery and Data Mining*. 2021, pp. 66–77. DOI: 10.1007/978-3-030-75762-5_6.

[168] Andrew Watters and Sarah Boslaugh. *Statistics in a Nutshell: A Desktop Quick Reference*. In a Nutshell (O'Reilly). USA: O'Reilly Media, 2008.

[169] Thorsten Wittkopp, Alexander Acker, Sasho Nedelkoski, Jasmin **Bogatinovski**, Dominik Scheinert, Wu Fan, and Odej Kao. "A2Log: Attentive Augmented Log Anomaly Detection". In: *Proceedings of the 55th Annual Hawaii International Conference on System Sciences*. Honolulu, HI: ScholarSpace, University of Hawaii at Mano, Hamilton Library, 2022. DOI: 10.24251/HICSS.2022.234.

[170] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. "A Survey on Software Fault Localization". In: *IEEE Transactions on Software Engineering* 42 (2016), pp. 707–740. DOI: 10.1109/TSE.2016.2521368.

[171] Erci Xu, Mai Zheng, Feng Qin, Yikang Xu, and Jiesheng Wu. "Lessons and Actions: What We Learned from 10K SSD-Related Storage System Failures". In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, 2019, pp. 961–976.

[172] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. "Detecting Large-Scale System Problems by Mining Console Logs". In: *Proceedings of the ACM 22nd SOSP*. New York, NY, USA: Association for Computing Machinery, 2009, pp. 117–132.

[173] Kenji Yamanishi and Yuko Maruyama. "Dynamic Syslog Mining for Network Failure Monitoring". In: *Proc. of the 11nd SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2005, pp. 499–508.

[174] Bo Yang, Xiao Fu, Nicholas D. Sidiropoulos, and Mingyi Hong. "Towards K-Means-Friendly Spaces: Simultaneous Deep Learning and Clustering". In: *Proc. of the 34th International Conference on Machine Learning*. 2017.

[175] Kundi Yao, Guilherme B. de Pádua, Weiyi Shang, Steve Sporea, Andrei Toma, and Sarah Sajedi. "Log4Perf: Suggesting Logging Locations for Web-Based Systems' Performance Monitoring". In: *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*. New York, NY, USA: Association for Computing Machinery, 2018, pp. 127–138. DOI: 10.1145/3184407.3184416.

[176] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael M. Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. "Be Conservative: Enhancing Failure Diagnosis with Proactive Logging". In: *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. Hollywood, CA: USENIX Association, 2012, pp. 293–306.

[177] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. "Characterizing logging practices in open-source software". In: *34th International Conference on Software Engineering (ICSE)*. 2012, pp. 102–112. DOI: 10.1109/ICSE.2012.6227202.

[178] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. "Improving Software Diagnosability via Log Enhancement". In: *ACM Trans. Comput. Syst.* 30 (2012). DOI: 10.1145/2110356.2110360.

[179] Zebrium. *Zebrium*. Last Access 20 Jan 2022. 2021. URL: https://www.zebrium.com/.

[180] Xu Zhang, Yong Xu, Qingwei Lin, Bo Qiao, Hongyu Zhang, Yingnong Dang, Chunyu Xie, Xinsheng Yang, Qian Cheng, Ze Li, Junjie Chen, Xiaoting He, Randolph Yao, Jian-Guang Lou, Murali Chintalapati, Furao Shen, and Dongmei Zhang. "Robust Log-Based Anomaly Detection on Unstable Log Data". In: *Proceedings of the 2019 27th ACM Joint Meeting on ESEC/FSE*. New York, NY, USA: Association for Computing Machinery, 2019, pp. 807–817.

[181] Xu Zhao, Kirk Rodrigues, Yu Luo, Michael Stumm, Ding Yuan, and Yuanyuan Zhou. "The Game of Twenty Questions: Do You Know Where to Log?" In: *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*. New York, NY, USA: Association for Computing Machinery, 2017, pp. 125–131. DOI: 10.1145/3102980.3103001.

[182] Zhenfei Zhao, Weina Niu, Xiaosong Zhang, Runzi Zhang, Zhenqi Yu, and Cheng Huang. "Trine: Syslog anomaly detection with three transformer encoders in one generative adversarial network". In: *Applied Intelligence* 52 (2022), pp. 8810–8819. DOI: 10.1007/s10489-021-02863-9.

[183] Jiang Zhaoxue, Li Tong, Zhang Zhenguo, Ge Jingguo, You Junling, and Li Liangxiong. "A Survey On Log Research Of AIOps: Methods and Trends". In: *Mobile Networks and Applications* 26 (2021), pp. 2353–2364. DOI: 10.1007/s11036-021-01832-3.

[184] Chen Zhi, Jianwei Yin, Shuiguang Deng, Maoxin Ye, Min Fu, and Tao Xie. "An Exploratory Study of Logging Configuration Practice in Java". In: *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. New York, USA: IEEE Press, 2019, pp. 459–469. DOI: 10.1109/ICSME.2019.00079.

[185] Wubai Zhou, Liang Tang, Chunqiu Zeng, Tao Li, Larisa Shwartz, and Genady Ya. Grabarnik. "Resolution Recommendation for Event Tickets in Service Management". In: *IEEE Transactions on Network and Service Management* 13.4 (2016), pp. 954–967. DOI: 10.1109/TNSM.2016.2587807.

[186] Jieming Zhu, Pinjia He, Qiang Fu, Hongyu Zhang, Michael R. Lyu, and Dongmei Zhang. "Learning to Log: Helping Developers Make Informed Logging Decisions". In: *Proceedings of the 37th International Conference on Software Engineering - Volume 1*. IEEE Press, 2015, pp. 415–425.

[187] Jieming Zhu, Shilin He, Jinyang Liu, Pinjia He, Qi Xie, Zibin Zheng, and Michael R. Lyu. "Tools and Benchmarks for Automated Log Parsing". In: *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*. Montreal, Quebec, Canada: IEEE Press, 2019, pp. 121–130.

[188] Yichen Zhu, Weibin Meng, Ying Liu, Shenglin Zhang, Tao Han, Shimin Tao, and Dan Pei. *UniLog: Deploy One Model and Specialize it for All Log Analysis Tasks*. 2021.

[189] Marinka Zitnik, Monica Agrawal, and Jure Leskovec. "Modeling polypharmacy side effects with graph convolutional networks". In: *Bioinformatics* 34.13 (2018), pp. 457–466. DOI: 10.1093/bioinformatics/bty294.