Diagnosing Software Bugs using AI-based Root Cause Analysis of Log Data

by Osazuwa Edoigiawerie

B.S. in Computer Science and Engineering, December 2003, University of Colorado, Denver
B.S. in Applied Mathematics with Business Minor, December 2003, University of Colorado, Denver
M.S. in Computer Science, May 2007, University of Colorado, Denver
M.S. in Data Science, December 2022, Johns Hopkins University, Maryland

A Praxis submitted to

The Faculty of
The School of Engineering and Applied Science
of The George Washington University
in partial fulfillment of the requirements
for the degree of Doctor of Engineering

August 15, 2025

Praxis directed by

Hamza F. Alsarhan
Professorial Lecturer of Engineering Management and Systems Engineering

The School of Engineering and Applied Science of The George Washington University certifies that Osazuwa Edoigiawerie has passed the Final Examination for the degree of Doctor of Engineering as of July 21, 2025. This is the final and approved form of the Praxis.

Diagnosing Software Bugs using AI-based Root Cause Analysis of Log Data

Osazuwa Edoigiawerie

Praxis Research Committee:

Hamza F. Alsarhan, Professorial Lecturer in Engineering and Applied Science, Praxis Director

Adewale Akinfaderin, Professorial Lecturer in Engineering and Applied Science, Committee Chair

Kevin Abreu, Professorial Lecturer in Engineering and Applied Science, Committee Member

© Copyright 2025 by Osazuwa Edoigiawerie
All rights reserved

Dedication

The author wishes to dedicate this dissertation to my late father, Godwin U. Edoigiawerie whom unquestionable believe in the value of education was always an inspirational guide, a motivator and a fuel for my passion to continue to strive for more educational attainment.

Acknowledgements

I would like to thank to all the lecturers in the AI/ML program for their support, and dedication to ensuring and fostering a robust exchange of knowledge throughout this program lecture stage.

Finally, and especially, I would like to express my deepest gratitude to my advisor, Dr. Hamza Alsarhan, for making my research journey a success with his continuous support, invaluable guidance, and feedback.

Abstract of Praxis

Faults in software systems predispose them to exhibit behaviors that can be used to fingerprint them. These fingerprints, captured as chronological snapshots in a system's persistent runtime logs, carry information that can potentially aid in pinpointing the originating location of the fault. Thus, manual or automatic analysis of these captured snapshots is utilized in determining the health status of the system, whether nominal or anomalous. A system in a non-nominal state requires root-cause analysis of the faults' symptoms encapsulated in the anomalous log data. However, the complexity prevalent in today's software systems invalidates the manual approach of troubleshooting faults from log data and necessitates powerful DL-based automatic techniques. As such, this research pursues the log data-based defect diagnosis tasks of defect classification and defect localization, leveraging LSTM, Bi-LSTM, and Transformer models.

While the developed defect classification Bi-LSTM and Transformer models were adept at categorizing defects, they also provided insights useful for understanding the relationship between the utilized Hadoop Distributed File System (HDFS) log data processing method and their prediction accuracy. To localize bugs, a summarization encoder-decoder LSTM and Transformer models were trained by establishing a novel technique denoted as Search Enhanced Summarization (SES). In the SES technique, the bug-revealing log message, commonly referred to as the summary, is first extracted from the fault's stack trace. Then, the predicted summary is used to query a pre-populated Retrieval Augmented Generation (RAG) vector database to retrieve one, three, or five matching records. These matches to the summary query vector contain the potential

source code method name responsible for generating the log message query. At least one of the retrievals approximates the fault location.

Table of Contents

Dedication ..... **iv**

Acknowledgements ..... **v**

Abstract of Praxis ..... **vi**

List of Figures ..... **x**

List of Tables ..... **xii**

List of Symbols ..... **xii**

List of Acronyms ..... **xiii**

Chapter 1—Introduction **1**

1.1 Background 1

1.2 Research Motivation 2

1.3 Problem Statement 3

1.4 Thesis Statement 4

1.5 Research Objectives 5

1.6 Research Questions and Hypotheses 6

1.7 Scope of Research 8

1.8 Research Limitations 8

1.9 Organization of Praxis 10

Chapter 2—Literature Review **11**

2.1 Introduction 11

2.2 Pre-Mortem Fault Diagnostics Strategies 12

2.3 Just-In-Time Diagnostics Techniques 14

2.4 Post-Mortem Defect Diagnosis 17
2.5 Gaps in the Literature 32
2.6 Summary and Conclusion 33
Chapter 3—Methodology 36
3.1 Introduction 36
3.2 Data Preparation and Exploration 36
3.3 Hypothesis Verification Methods and Tools 46
3.4 Model Training, Validation and Evaluation 48
Chapter 4—Results 59
4.1 Introduction 59
4.2 Data Analysis Results 60
4.3 Fault Classifiers Performance Evaluation 70
4.4 Fault Localizers Performance Evaluation 74
4.5 Findings from Evaluation Results 76
4.6 Results Summary 81
Chapter 5—Discussion and Conclusions 82
5.1 Discussion 82
5.2 Conclusions 86
5.3 Contributions to Body of Knowledge 87
5.4 Recommendations for Future Research 88
References 90
Appendix A 97

List of Figures

Figure 1-1. How Developers Spend Their Time. ……...7

Figure 3-1. Bar Plot of Defect Types Block Frequency. ……..40

Figure 3-2. Log Message Parsing Output for Drain and Custom Parser. ……....42

Figure 3-3. Classification Bi-LSTM Training Process. ……..…..51

Figure 3-4. Classification Transformer Training Process. ……...52

Figure 3-5. Log Message Summarization Encoder-Decoder LSTM Training Process. ...56

Figure 4-1. Invalid Set Deletion Word and Word Count Length Distributions. 61

Figure 4-2. IO Failure Word and Word Count Length Distributions. 62

Figure 4-3. Missing Block Info Word and Word Count Length Distributions. 64

Figure 4-4. Orphan Block Request Word and Word Count Length Distributions. 65

Figure 4-5. Redundant Block Request Word and Word Count Length Distributions. 66

Figure 4-6. Unexpected Early Halt Word and Word Count Length Distributions. 68

Figure 4-7. Log (Left) and Summary (Right) Word Count Length Distributions. ……...69

Figure 4-8. Defect Locator Runtime Log Data Tokens (Left) and Summary Data Tokens (Right). ……..70

Figure 4-9. Defect Localizers Accuracy@k Report. ……..77

Figure 5-1. Classification Training and Evaluation Data Size by Fault Type. ……...83

List of Tables

Table 3-1. Typical HDFS Log Message Structure ……... 38

Table 3-2. Count of Anomaly and Nominal Block Sequences …..……. 38

Table 3-3. Count of Fault Labels ……...……. 39

Table 3-4. HDFS Data Allocation ……...…...……. 49

Table 4-1. Invalid Set Deletion Word Count Length Statistics ……... 61

Table 4-2. IO Failure Word Count Length Statistics ..……...…...…..……. 62

Table 4-3. Missing Block Info Word Count Length Statistics ..……...…...…… 63

Table 4-4. Orphan Block Request Word Count Length Statistics ..……...…...…... 65

Table 4-5. Redundant Block Request Word Count Length Statistics ..……...…..... 66

Table 4-6. Unexpected Early Halt Word Count Length Statistics ..……...…..... 67

Table 4-7. Log Data and Summary Data Vocabulary Statistics …….. 69

Table 4-8. Fault Log Points Fully Qualified Java Method Names …….. 71

Table 4-9. Bi-LSTM Layers ……72

Table 4-10. Transformer Layers ……. 73

Table 4-11. Defect Types Label Name and ID …… 73

Table 4-12. Bi-LSTM and Transformer (on custom parser outputs) Classification Report ……... 75

Table 4-13. Bi-LSTM and Transformer (on Drain parser outputs) Classification Report 76

Table 5-1. Classification Training and Evaluation Data Size by Fault Type …….. 84

List of Symbols

s_i	Log message sequences of block i, where i = 1,2,3,…
t_i	Defect type i, where i = 0,1,2,3,4,5
c_i	Source code file i, where i = 1,2,3,…
r_i	Bug report i, where i = 1,2,3,…
q_i	Log sequence summary i for sequence s_i, where i = 1,2,3,…
m_i	Source code method i, where i = 1,2,3,…

List of Acronyms

BPNN	Back-Propagation Neural Network
IEEE	Institute of Electrical and Electronics Engineers
EDR	Endpoint Detection and Response
U.S.	United States
IPC	Inter-process communication
SDLC	Software Development Life Cycle
AI	Artificial Intelligence
DL	Deep Learning
Bi-LSTM	Bi-directional Long Short-Term Memory
CSV	Comma-Separated Values
LSTM	Long Short-Term Memory
RAG	Retrieval Augmented Retrieval
SFP	Software Fault Prediction
LOC	Lines of Code
HDFS	Hadoop Distributed File System
SBFL	Spectrum-based fault localization
MBFL	Mutation-based fault localization
ANN	Artificial Neural Network
CNN	Convolutional Neural Network
TF-IDF	Term Frequency-Inverse Document Frequency
SES	Search-Enhanced Summarization

Chapter 1—Introduction

1.1 Background

The trifecta of fault, error, and failure are the three basic elements of software system failures (Laprie, J.-C. 1995). IEEE standard 1044 (2009) defines a fault, generically referred to as a defect or bug, as "an imperfection or deficiency in a work product where that work product does not meet its requirements or specifications and needs to be either repaired or replaced" (IEEE Standard classification for software anomalies, 1044, 2009). The genesis of a system failure is a fault – any combination of misconfiguration, hardware malfunction, or software bug. The fault state observed as an error, whether latent or active, might eventually lead to a failure, depending on the robustness of the system. The same standard from the IEEE Software Engineering Committee of 2009 defines failures as "[E]vents in which a system or system component does not perform a required function within specified limits".

In 2022, the cost incurred due to failure from poor software quality is estimated at US $2.41 trillion (Krasner, H. 2022). A recent poignant example is the worldwide outage resulting from a failure in the Endpoint Detection and Response (EDR) software agents of the cybersecurity firm, CrowdStrike, on July 19, 2024. This failure caused system crashes in about 8.5 million Microsoft Windows computers, with the worldwide financial cost estimated to be at least US $10 billion. The root cause was a faulty update pushed by CrowdStrike to its Falcon Sensor security software. The fault primarily emanated from an Inter-process communication (IPC) template type update with an extra field that triggered a mismatch between the expected input and the supplied input.

Given the complexity of software, the dynamism of operating environment conditions, and humans' predisposition to make mistakes, defects will inevitably be introduced in any phase of the SDLC. The maintenance phase, which takes up a large portion of the SDLC, unsurprisingly surfaces most defects and consequently consumes between 40% and 80% of software cost (Glass R. L. 2001). To continue to meet functional requirements and achieve business goals, addressing defects with fixes that correct the root cause becomes imperative. Fortunately, most software systems, whether open source or commercial, keep track of and persist their runtime status in log files. Mining these status records facilitates defect diagnosis by providing relevant information needed for bug detection, bug classification, bug localization, and bug fixes.

1.2 Research Motivation

In a study to ascertain whether critical software failure can be prevented through simple testing (Yuan D. et al., 2014), it was reported that in 76% of system failure occurrences, a failure message was explicitly printed in log files by the system. Furthermore, in 84% of failures, all the triggering events that caused the failure were recorded in the system logs.

Consequently, when failures are reported, log messages provide the sole source of data for postmortem analysis. Typically, the developer who is assigned to fix the bug follows a diagnostic process that includes (1) mapping the log messages static portion (known as log template or event) from the log data to the source code location of the error; (2) systematically following the trail of bug symptoms in the execution path of the

recorded triggering events backward from the initial events to the cause of the bug; and (3) exploiting this knowledge to design and implement an appropriate fix.

In an empirical study by Chen, A. R. (2019) across 16 Apache Foundation open-source projects on the usefulness of log data for debugging production failures, it was discovered that the reconstructed execution path from log messages correctly maps to the right context in the source code. As mentioned above, log data available in most failure cases (Yuan D. et al., 2014), containing complete bug symptom information, makes it possible to zero in on the root triggering event and deduce the root fault. Even in cases with insufficient logging, the available log messages can help in narrowing the diagnostics search space.

Due to the increasing volume of log data generated by software systems that grow in size, number, and complexity every day (Krasner H., 2022), automated log analysis has become essential for failure debugging. While traditional software techniques and traditional machine learning methods are mainly leveraged for these diagnostics endeavors, the use of deep learning methods is not yet prevalent. Given the success achieved in deep learning-based anomaly detection, it is encouraging and timely to attempt to replicate gained experience and knowledge in pursuit of AI-based defect diagnosis or debugging.

1.3 Problem Statement

Large software systems are prone to complex defects due to their scale and architectural complexity (Krasner, H., 2021), resulting in an annual expenditure of $607 billion on defects fixes in the US alone (Krasner, H., 2022)..

To facilitate the detection of defect root causes and access to more diagnostic information, some level of automation of a bug fix or automatic check of a manually provided patch will be indispensable. Typically, log messages, whether informational, a warning, or an error, are inputted by developers into source code in an ad-hoc, as-needed manner (Yuan D. et al., 2011). Log enhancement methods like LogEnhancer (Yuan D. et al., 2011), ErrLog (Yuan D. et al., 2012), etc., that systematically add appropriate logging messages to source code have been proposed, implemented, and validated.

Efforts towards automated solutions have also been demonstrated to work in these three types of defect-handling cases. The first type is solutions that check for patch correctness, as in FixCheck (Molina F. et al., 2024), Shibboleth (Ghanbari & Marcus, 2022), PATCH-SIM (Xiong Y. et al., 2018), etc. The second group is the Automatic Program Repair (APR) techniques, such as Getafix (Bader J. et al., 2019), Sequencer (Chen Z. et al., 2021), etc. The last type of defect diagnostics methods can reconstruct execution paths from logs and include tools such as LogMap (Chen, A. R., 2019), Sherlog (Yuan D. et al., 2010), etc.

1.4 Thesis Statement

A deep learning-based root cause analysis model using log data from defective software systems will help ensure appropriate solutions to discovered defects and mitigate unintended defects.

To alleviate the challenge inherent in manual diagnosis and potentially reduce the effort and time needed to locate the code that introduced the bug, an automated solution

that cannot only find potential diagnostic execution paths but also select the most likely execution path is essential. Automated defect root cause localization and categorization can be achieved through these 3 methods: traditional software techniques, statistical machine learning algorithms, and deep learning methods.

LogEnhancer (Yuan D. et al., 2011), LogMap (Chen, A. R., 2019), Sherlog (Yuan D. et al., 2010), etc., are some examples of traditional software methods leveraging a combination of log data and source code for defect diagnosis. Meanwhile, Dobrowolski W. et al. (2022) research on identifying defect types demonstrates the use of statistical machine learning techniques for defect diagnosis. Lastly, Deeplog (Du M. et al., 2017), which pioneered the use of deep learning for anomaly detection, also confirms the effectiveness of deep learning techniques for defect diagnosis. DL-based models' ability to learn complex patterns from data makes them adept at reconstructing the execution path, which contains the defect location from which the defect was triggered.

1.5 Research Objectives

This research aims to develop an automated defect diagnosis method that will aid in categorizing a defect, finding or narrowing down the log point in the source code from which the bug emanated, and fostering an understanding of other code points and their combinations that can surface defects. To achieve these objectives, a transformer-based model, LSTM-based model and a BI-LSTM deep learning model will be trained with the following high-level goals.

- To provide a solution that leverages the powerful pattern-learning capability of DL networks to aid in inferring the cause and type of a software defect.
- To pinpoint the bug-revealing message log in a defect log sequence using summarization.
- To localize a bug to the correct log points in source code mapped from the bug symptoms messages in the log data.
- To narrow down the bug diagnosis search space in cases where a limited execution path is emitted due to insufficient log messages
- To generate alternative execution paths that can lead to similar or the same defect.

1.6 Research Questions and Hypotheses

Minelli R. et al. (2015) provided an in-depth analysis of how developers spend their time. Their verdict : software developers now spend about 70% of development time understanding code, while only expending just about 5% of their time in writing code.

Given AI's ability to learn correct and incorrect programming patterns (Krasner H., 2022) from millions of lines of code and exploit that knowledge to understand other code, it would be more effective to pursue bug troubleshooting by pairing AI with developers. With this context in mind, the following questions and hypotheses are proposed.

RQ1: Between Bi-LSTM and Transformer-based models, which will be more accurate at software system defect categorization?

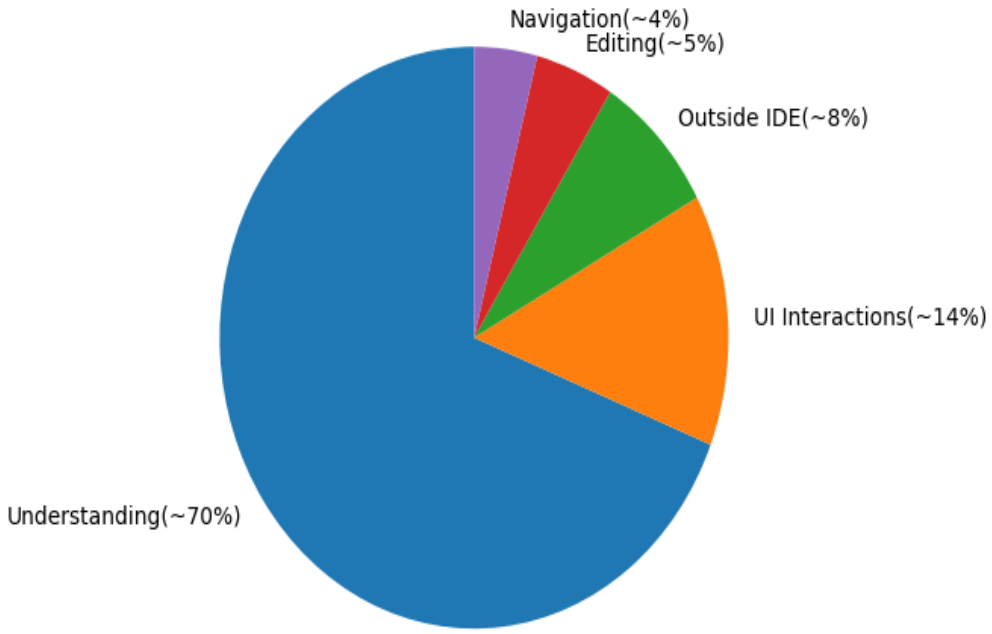

Figure 1-1. How Developers Spend Their Time (Minelli R. et al., 2015)

RQ2: Does pre-processed log data using the Drain parser improve the diagnostic prediction accuracy of transformer-based and Bi-LSTM models as compared to raw log data?

RQ3: Is the log message summarization query predicted by an LSTM model more effective at defect localization than that generated by a Transformer-based model?

H1: A transformer-based model will be more accurate than a Bi-LSTM model for software system defect categorization.

H2: Diagnostic prediction accuracy will be better for a Bi-LSTM and Transformer-based model trained on raw log data relative to pre-processed data using the Drain parser.

H3: The log message summarization query predicted by a Transformer model will aid in a more effective defect localization than the log message summarization query predicted by an LSTM model.

1.7 Scope of Research

The detective work that developers undertake to fix detected bugs in software systems includes two main tasks: bug localization and bug classification or categorization. While the former is concerned with mapping bug symptoms in the log data to the log points in source code, the latter focus is on the type of the defect. Following successful completion of these tasks, it becomes easier to identify the actual fault that triggered the defect.

This research attempts to automate these steps using deep learning algorithms. To achieve these pursuits, a Bi-LSTM model, an LSTM-based model and a Transformer-based model are developed. These models are trained on the health information (emitted as log message) of the defective system. The models' performance also undergoes comparative analysis on the standard metrics, such as accuracy, precision, recall, and F1-score. The outputs of the created models are either a fault category or at least a partial execution path that can aid in bug classification or bug localization, respectively.

1.8 Research Limitations

The objective of this praxis research is to aid in diagnosing the root cause of software failure by classifying and localizing the fault that caused the observed failure using log data alone. Ideally, in order for a developer to troubleshoot a bug and provide a

patch that addresses the root cause of that bug, she/he extracts and collates information from the bug report, associated log data, the source code log points, and a defect-revealing unit test. Thus, leveraging just log data and source code log points as the training data inevitably introduces limitations on the developed models.

The models are limited to learning the defect fingerprint of certain bohrbugs. A bohrbug is defined as "a fault that is easily isolated and that manifests consistently under a well-defined set of conditions, because its activation and error propagation lack 'complexity' as set out in the definition of Mandelbug" (Grottke & Trivedi, 2005). The defect sites, such as the log point source code method or function and other related caller methods, must be present in the log data that the models learn from.

Another limitation is the distribution of the log data. First, there is a large imbalance in the different defect types of the log data sequences, and that would have an adverse effect on the performance of the classification models. Secondly, repetitions in the training data addressed via deduplication to reduce noise and handle model bias are patterns that are very useful in manual defect troubleshooting. Thirdly, although the detective work of seeking the generating source code for the log data was achieved through a careful manual process, the task of mapping source code log points to the log data was done programmatically. Thus, the mapping could be wrong in some cases, especially for cases where the log points are in external libraries. Nevertheless, this does not diminish the usefulness of the diagnostics models, as this can be easily addressed by retraining on correct mapping data. Lastly, the generated log data and source code log points used for training and evaluating models are from a single software system (HDFS) written in Java .

1.9 Organization of Praxis

The rest of the praxis is organized as follows. Chapter 2 will delve into related studies, which would include automated diagnosis techniques spanning the prevalent traditional software solutions, statistical/traditional machine learning methods, and deep learning-focused techniques. In Chapter 3, the focus would be on the methodology and experiments that were carried out to attempt to answer the research questions and validate the hypotheses. Chapter 4 presents the results of the empirical studies conducted in the previous chapter. Finally, Chapter 5 will conclude the praxis research, reflecting on the lessons learned and potential future directions that the lessons will spawn.

Chapter 2—Literature Review

2.1 Introduction

The sequences of log messages that log data is composed of are a representation of the underlying execution workflow of the system that generated them, as well as a snapshot of the health of the system at the time they were written. In all software systems, log statements, like println, console.log, and logger.info, are embedded in the source code of the software to record information about the runtime state of the system.

Across software systems, there is no consensus on the format of each log message. Thus, the format of log messages, which are typically human-readable text, can range from the structured format (such as delimited text as in a Comma-Separated Values (CSV) file) through semi-structured format (as in key-value pair) to unstructured free-form text. Nevertheless, generally there is a header portion, like the timestamp or log level (e.g. Error, warning, info, etc.), usually followed by the message content. The message content is composed of static text, added by the source code developer, and a dynamic part, such as variables, that is produced at runtime. For model training, the feature set is commonly the extracted static text portion, commonly referred to as the log template (or log event or log signature), and in some cases includes the dynamic runtime part of the message, which can provide additional contextually relevant information for defect diagnosis.

To shine some light on defect diagnosis, Yuan W. et al. (2020) embarked on a comprehensive study of 106 distributed bugs, randomly sampled from five widely used distributed software systems: Cassandra, HBase, HDFS, MapReduce, and ZooKeeper.

Their research objective focused on answering the question of how developers leverage log to diagnose and fix distributed bugs produced seven findings. The resonating theme in all seven findings, the actual trail of a defects' symptoms that triggered the defect and potentially led to its root cause is not always obvious or present in the log messages emitted by a single node. Rather, one would need to untangle interleaved messages, reconcile unsynchronized times spanning multiple nodes, and decipher logged message content. These operations describe the process of defect diagnosis that falls under the domain of post-mortem or forensic log analysis.

To engender a robust post-mortem defect diagnosis through log analysis, a multi-prong approach is essential. Diagnosing faults should additionally include an understanding of the need for preventative pre-mortem steps involving software defect prediction and effective logging strategies and a proactive (near) real-time online anomaly detection of software faults.

2.2 Pre-Mortem Fault Diagnostics Strategies

Software defect mitigation through preventative diagnostics methods either leverages software source code and/or runtime log data that the software generates. While the aim of exploiting source code for fault prediction is to learn about the susceptibility of certain modules/classes/functions to bugs; using log data for this purpose provides guidance with respect to two equally important logging decisions: location of log points in source code and what to log.

Pandey et al. (2020) Systematic Literature Review (SLR) of various studies spanning 1990 to 2019, focused on the use of statistical methods and machine learning

models for Software Fault Prediction (SFP), concluded that machine learning models are more effective at predicting software components' fault proneness. Furthermore, Esteves, G. et al. (2019) research going a step further gave an explanation on how machine learning models can handle software defect prediction tasks by enumerating two types of input features that these models learn from. The two types of code features include: features based on source code metrics, like Lines of Code (LOC), and feature extracted from source code and metadata information. The benefits of source code-based proactive defect identification include a reduction in the software maintenance and development cost (Seliya, N. et al., 2010).

Meanwhile, non-source code-based fault diagnosis mitigation strategy proffers a technique to procure more targeted diagnostics information without directly identifying individual faults. The provided information, in the form of log data, is the backbone of a successful and effective defect diagnosis. Without appropriate log data, finding the root cause will take far more effort, resulting in increased software maintenance costs in terms of developer time and customer churn. The main goal of this strategy is to enhance log messages by helping developers make good logging decisions on where in the source code to locate messages and what messages to log.

Zhu J. et al. (2015) provides insight on where to log in an empirical study, in which they proposed, designed and implemented a logging suggestion tool, LogAdvisor to help guide developers on strategic placement of logging statements in software source code. Their assumption in designing the model that drives this tool is that aggregating the best logging practice from experienced developers, as the input features into a machine learning model can automate the process of how to log. The tool, which works by

learning logging instances from source code, leverages two typical error reporting points in source code – sites with exception handling statements and function call return sites in the caller. The next step after having achieved the goal of locating log statements is to log content that will be insightful for post-failure debugging.

On the question of what to log, two studies provide some clues. The first study presented ErrLog, a tool designed and implemented by Yuan D. et al. (2012). This tool proactively adds logging statements to source code at the return location of function calls. Their decision to develop this tool was motivated by the discovery that half of the failures from a comprehensive study of 250 failures in five large and widely used software systems could not be diagnosed with the current log data generated by the systems. From the second study by Yuan D. et al. (2011), another log content guidance tool, LogEnhancer was created. This tool aids in meeting the challenge of insufficient logging in production systems, challenged by the need for an urgent patch to a production bug. It achieves this by providing more diagnostics information for rare failures that otherwise will require an enormous effort to fix. However, unlike ErrLog, that adds new log statements, LogEnhancer just enhances already existing log messages with additional information from variable values.

2.3 Just-In-Time Diagnostics Techniques

These methods are aimed at identifying defects, as they are happening, in real time or near real time. They are the methods employed in the well-studied issue of anomaly detection leveraging log data, traces, and metrics.

Traditionally, anomaly detection techniques involve the use of rules or keywords to search through log data to identify suspicious logs that signal an anomaly. This worked well for software of yesteryears, with far less complexity, relative to current large, modern software systems that are deployed across many nodes, and produce an increasing volume, velocity, and variety of log data. Much of the unsuitability of manually coded, rule-based signatures mining in systems' logs centers around their lack of adaptability to new signatures introduced from system upgrades, bug fixes, and feature additions. To surmount the challenge presented by this inadequacy in manual log analysis, machine learning models, and more recently, deep learning models, are being exploited to automate the task of anomaly detection through log analysis.

Log data anomaly detection methods pioneered the log data pre-processing technique of creating a log template or event and vector of parameters from the static and dynamic portions, respectively, of each log message, in otherwise unstructured data. Following log messages parsing into log templates, template clustering based on ID (such as session ID or thread ID) or time-based partitioning is performed. The resulting sequence partitions then serve as input features for the models. For many ML-based techniques, further transformation of the log templates for each partition into a numeric vector is pursued. In this case, the feature vector content is the count of the log templates in each log partition for all partitions. On the other hand, DL-based models either use as input the index vector of log events or directly use a semantic and/or contextually relevant vector embedding of log events.

While machine learning and statistical methods (such as PCA, clustering, logistic regression, support vector machines, decision trees, etc.) are an improvement over the

expert manually crafted rules, they are still limited in understanding the semantic information in logs. A limitation is that they share the shortcoming of generalizability to unseen or rare log events with rule-based techniques. With DL-based anomaly diagnosis models adept at learning complex patterns from data, they can decipher semantic and contextual information. Consequently, these DL models can identify patterns that don't match the training data class pattern.

Le & Zhang (2021), after establishing the prevalence of log data parsing errors, through an empirical study, in which 4 top-ranked log parsers were used to pre-process Thunderbird and BGL logs, developed an anomaly detection transformer-based model with a multi-head attention mechanism. Using BERT, they encoded raw log messages into vector embeddings consumed by the model. The resultant model showed good performance on four publicly available datasets, as evident in the F1-score of at least 0.95.

Zhang, X. et al. (2019) also exploited DL networks to address the issue of log data instability due to noise and modifications from bug fixes and new features. They trained a Bi-LSTM model on a log template semantic vector to achieve robustness in the presence of dynamic log data. Their Bi-LSTM model evaluation on evolving log event data from Hadoop and Microsoft online service achieved an impressive F1-score of 0.89 and 0.84, respectively, even for log event changes of up to 20%.

Given the inherent imbalance in log data distribution, with far more nominal status data than anomalous scenarios, Xia, B. et al. (2020) set out to address this issue head-on by leveraging the generative capability of an LSTM-based GAN model. They achieved this objective through a permutation event modeling technique, involving

sliding different window sizes on the log messages to construct new training data sets that facilitated effective pattern-based training. Their GAN model performance was suboptimal with precision, recall, and F1-score of 54.60%, 97.60%, and 70.00%, respectively.

Du, M. et al. (2017) pioneered deep learning-based log anomaly detection, under the name of DeepLog,. Their approach involved the creation of several LSTM-based models: an anomaly detection model, a workflow model, and a parameter value model. To train the models, they utilized an index vector synthesized from log message templates to handle the tasks of both just-in-time diagnosis and forensic diagnosis. The performance of the anomaly model was 100.00% accuracy on 99.00% of the datasets after being trained on just 1.00% of the data.

2.4 Post-Mortem Defect Diagnosis

After a defect has been detected, it is necessary to provide a fix that addresses the right fault, with the right solution, i.e., the correct patch for the root cause of the discovered bug. In software engineering, this stage of fault diagnosis is referred to colloquially as program debugging or bug troubleshooting. Debugging is one of the most critical and time-consuming software development activities that programmers perform. Typically, developers manually hunt for keywords in logs to answer two main questions: where is the fault located? and what is the failure type?

As modern software systems continue to explore and leverage distributed architectures, they will inevitably grow in scale and complexity. Consequently, the log

data they generate will undoubtedly increase in velocity, variety, and volume, making the manual hunt for bug causes infeasible.

As a result, fault localization (where) and fault categorization (what) automation, although feasible, are not trivial and need to be pursued. Yuan, W. et al. (2019) comprehensive study spanning 106 real-world distributed bugs in 5 large-scale and popular distributed systems - Cassandra, HBase, HDFS, MapReduce, and ZooKeeper – made known some significant challenges that need to be overcome and opportunities to exploit to be able to automate bug diagnosis from log data. These challenges spanned the disparity in deciphering failure signatures propagation across multiple nodes as opposed to a single node, to the inconsistency in reported log level types.

Nevertheless, the two core defect diagnosis tasks - fault localization and fault classification - are achievable through the analysis of entries in a log, in most cases. To locate a source code line, method/function, or component, and categorize a defect type that identifies the bug root cause, the troubleshooting information should include one or all of the following: the bug's failure stack trace, the log statement emission point, or variable values.

2.4.1 Fault Localization

The task of attempting to find the exact location in a software system from where a reported bug was triggered encapsulates the fault localization process. Due to the scale of most software today, this task equates to the popular saying of looking for a needle in a haystack. Fault localization difficulty is inherent in the dependencies between code elements at all levels of granularity – code statements, methods/functions, classes,

services, and service instances. Thus, the techniques used to identify and tackle these buggy elements have also metamorphosed to adapt to and exploit the current technological innovations.

2.4.1.1 Log data – Source code Mapping with Traditional Fault Localization Techniques

Yuan D. et al. (2010) developed the Sherlog tool with the objective of identifying the triggering location of a fault. Sherlog infers the production failure execution path through an analysis of runtime log information and the corresponding source code. Its algorithm, which mimics how a human programmer would typically go about troubleshooting a bug, starts at the source code elements and variable values in the logs and strategically uses this information as a guide to walk through the code. What results from the walks are control flow information and data flow information. The control flow information is the execution paths category: Must-path, May-path, and Pruned-paths. While Must-paths are partial paths that were executed in the production failure, May-paths are paths that may have been executed when the production failure happened and Pruned-paths are execution paths that were not executed. On the other hand, data flow information in the form of variable values provides a mechanism to constrain the inferred execution paths. With this information, Sherlog can reconstruct the error execution path and, consequently, identify the potential root cause of the failure.

2.4.1.2 Slice-based Traditional Fault Localization Techniques

The approach pursued by slicing-based techniques involves the abstraction of all

the code statements from the faulty program, which were potentially executed along the path that led to the failure. This has the effect of reducing the search space for the fault location responsible for the bug being diagnosed. While the various slice-based methods are similar in their assumption that the location of a fault is in the slice containing the variable-statement(s) pair that precipitated the error, they differ with respect to the code statement(s) included in the constructed slice.

In static slicing, the code statements in a variable's slice are all the executable statements that could have potentially affected the value of the variable. Clearly, this would result in the inclusion of extra statements that did not trigger the error. To subvert this problem of a larger fault search space, dynamic information about the variable is needed.

Building on this idea, dynamic slicing focuses on the code statements that affected the value of a variable; therefore, it is an improvement on statically created variable slices. The improvement stems from the fact that, while static slicing uses just source code to construct a program slice, dynamic slicing uses both source code and runtime information, such as failure stack trace, to create the execution slice.

Other types of program slicing techniques have also been proposed to address the limitation of dynamic slicing. One such limitation is failure to handle execution omission errors, in which an error causes the omission of critical statements. This, of course, means that the created slice will not contain the omitted statements. Gyimothy et al., (1999) proposed relevant slicing, which augments dynamic information with potentially dependent information that captures the missing statements.

Lastly, Wong, W. E. et al. (2016) mentioned an alternative to both static and dynamic slicing in their comprehensive survey study on defect localization techniques, called execution slicing. This technique exploited the coverage information from test cases to create a slice that corresponds to the statements covered in the test runs.

Nevertheless, all these techniques faced a common challenge of not being able to guarantee that the buggy statement and all its dependent statements reside in the constructed slice.

2.4.1.3 Spectrum-based Traditional Fault Localization Techniques

Spectrum-based techniques are statistical and probability techniques that analyze the behavior of program spectra or entities, like conditional branches, by recording their failing and passing frequency on test cases. Then uses the tracked frequencies to compute a similarity coefficient on covered statements, which is subsequently used to rank the statements on how likely they are the cause of a failure. Finally, the likelihood value, which in the spectrum-based parlance is referred to as the suspiciousness value of a statement, is utilized for sorting statements in descending order of fault proneness. The different suspiciousness value calculation techniques have spawned many different flavors of Spectrum-based fault localization techniques (SBFL).

Renieres & Reiss (2003) defined the concept of set spectra types using set operations. Specifically, the defined set spectra are modeled after the set operations of union and intersection on a failing test case, with the objective of determining which part of the code to focus on. In set union spectra, the focus is on the covered code statements that result from the difference between a failing test execution and the union of a set of

successful test runs. Whereas, set intersection spectra is concerned with covered code statements computed from the difference of the intersection of a set of passing test runs and a failing test. An additional set of differencing spectra from Renieres & Reiss (2003) is the nearest neighbor spectra. For this spectrum type, the underlying idea, as with most nearest neighbor algorithms, leverages distance metric to determine element classes by grouping close elements into the same class. Using this metric, the closest failing test to successful tests is identified, and the set difference of their covered code statements is used to locate a bug.

One of the most popular and earlier spectrum-based techniques proposed by Jones, J.A et el. (2001) is known as Tarantula. Tarantula uses the coverage information and the results of a test run (success or failure) to compute the suspiciousness value of each statement in a program. Tarantula pioneered the innovative data representation of program statements and test cases using a sparse binary matrix, later adopted by many ML techniques tackling fault localization.

Another variant of SBFL is the Mutation-based fault localization (MBFL) techniques designed to fill the gap left by earlier SBFL. Some of these limitations include buggy statements not being covered by any of the defined test cases, failed tests whose cause is not in any of the covered code statements, and cases where buggy statements produce successful test results.

The core idea in mutation testing is to learn if mutating source code statements will cause a passing test to fail. Thus, MBFL leverages the impact of each mutated code statement on test cases to identify the location of faults. Papadakis & Traon (2015) created the first MBFL technique, Metallaxis. Metallaxis separates code statement

coverage based on whether its mutants have any impact on the outcome of the test. Moon, S. et al. (2014) MUSE is another MBFL technique based on two similar intuitive assumptions: mutating a buggy statement may cause more failed tests to pass than mutating correct elements and mutating correct code statements may cause more passed tests to fail than mutating buggy statements.

In general, all the SBFL techniques are scored on a performance metric first proposed by Jones, J.A et el. (2001). This metric is based on the percentage of executed program statements that do not need to be examined. A statement examination determination depends on whether the suspiciousness value violates a predefined threshold. The higher the calculated percentage, the more performant the method is.

All the SBFL techniques are limited by their duplication of the suspiciousness values for the different code statements, regardless of whether the exercised test passed or failed. This weakness is mainly due to the way these values are computed for SBFL techniques.

2.4.2 Traditional Machine Learning-based Fault Localization Techniques

In traditional software development, a developer writes explicit step-by-step instructions as algorithms that guide the machine on how to accomplish a task; thereby predisposing it to a limitation of inability to adapt to new situations. Machine learning systems differ in that their algorithms are designed to imbue them with experience provided in the form of input data from which they learn patterns that map to the desired output.

Briand et al (2007) proposed a C4.5 decision tree method for easing fault localization during program debugging. They developed a C4.5 decision tree that leverages a technique called category partition (CP) to perform feature extraction. In CP parlance, there is a need to identify categories, and the choices associated with each category. Accordingly, applying CP to test specification involves identifying test function parameters characteristics, and each parameter's values, as the categories and choices, respectively. To ensure that the identified categories and choices impact the software behavior, CP facilitates the construction of test cases, covering various permutations of execution conditions of test functions with respect to their parameter inputs. In the constructed decision tree, with nodes as the categories and edges defined by the choices, the path from the root node to a leaf node is an induction of the rules governing the series of test instance inputs and outputs that lead to a test failure or success. For example, a test failure is predicted for a leaf node when the majority of failing test cases trace a path to it; otherwise, the test passes.

Motivated by a real-life production bug in Hadoop Distributed File System (HDFS), which took many months to diagnose and fix, Dobrowolski W. et al. (2022) developed several machine learning models for software fault localization. A key insight for taking this approach stemmed from their observation that the difficulty of providing an appropriate patch to the bug stems from the inability to realistically replay the reproduction steps that led to the failure. Other insights included the impossibility of reproducing the failure outside the production environment in which it occurred, and developers' expertise encapsulated in a unit test. These drove their novel undertaking of simulating reproduction logs from unit test runs in two stages: parsing of each unit test

suite produced execution log outputs into log events, and subsequent transformation of the log events into numeric vectors. Finally, two ensemble models, (Random Forest Classifier (RFC) and Gradient Boosting Classifier (GBC)) and a Naive Bayes Classifier (NBC) model, are trained on the labelled vector sequences and used to infer the matching test suite class(es). From the matching test suite classes, the source code line that produced the fault can be pinpointed.

2.4.3 Bug Localization with Neural Networks

Neural networks, or more formally, artificial neural networks (ANN), are an adaptation of biological neural networks. Architecturally, they are described as either deep or shallow and either wide or narrow. A deep or shallow network is a function of the number of layers, while a wide or narrow network is defined by the number of neurons. A typical shallow network has 3 layers: an input layer, a hidden layer, and an output layer. At a certain threshold of the number of hidden layers, the network may be characterized as a deep neural network.

Wong & Qi (2009) employed a back-propagation neural network (BPNN) in an innovative first-of-a-kind empirical study on the automation of fault localization. Using statement-level coverage data and execution results from exercising test cases against a software program, a BP network was trained. The trained BP networks' ability to approximate complex non-linear functions ensured it was able to learn functions that associate the relationship between the statement coverage data, and the execution result outputs to a suspiciousness value or likelihood of each of the statements containing a bug. During inference, feeding the trained BP network test coverage data generated per

program statement from a created virtual test results in a ranked list of statements, sorted in descending order of suspiciousness from highest to lowest. Such a ranked code statement assists developers in making decisions on which code statements (or components) to examine first and spend time on.

To address BPNN paralysis and local minima limitations, Wong, W.E. et al. (2012) trained a 3-layer, Gaussian basis function activated Radial basis function (RBF) neural network for fault localization using software program test cases, code coverage information, and execution results, as was done in BPNN. To adapt RBF network for fault localization, the authors came up with three innovations: representation of test cases, statement coverage, and execution result within a modified RBF network formalism, the development of an algorithm to simultaneously approximate the number of hidden neurons and their receptive field centers, and the definition of a weighted bit-comparison-based dissimilarity to estimate the distance between statement coverage vectors of two test cases. Also, as in BPNN, after training the network is evaluated on coverage data generated from a virtual test of a similar program, and its inferred result is a prediction of the likelihood of a bug for each program statement.

2.4.4 Deep Learning-based Fault Localization

Generally, most studies in bug localization literature trained DL models with a pair of bug reports and source code files. In these cases, the bug report represented a query into a source code repository, with the search results being the class file from which the defect surfaced. However, as was done in this project, DL-based bug localization also leverages a pair of log data and source code for locating defect points.

Wang, B. et al. (2022) trained a CNN network they named Multi-Dimension CNN (MD-CNN) using data from three different repositories: historical bug reports, API documents and source files. During model training, they performed two primary tasks: statistical feature extraction and bug report feature matrix construction. In the feature extraction tasks, the objective was to mine a set of features from the three raw data repositories that captured the various types of relationships between the bug reports and source files. To that end, they extracted five independent features: text similarity between bug report and source file, bug-source relationship by similarity of bug reports, bug-source relationship by recently fixed source files, bug-source relationship by class name similarity, and bug-source relationship by structural similarity. Afterwards, the other task uses the extracted features to construct each bug report representation as a feature matrix of dimension five by the number of source files. Through training, the model learns to fuse all five features and generate a query feature matrix. The generated query output is a ranked list of source files from which the reported defect emanated. So that, at inference time, when a new bug report is received, the model can generate its feature matrix and predict the ranked source file list potentially containing the defect. Evaluation of six eclipse projects, including Aspectj, Birt, Eclipse, JDT, SWT, and Tomcat, results in an average accuracy@5 of 0.67 and accuracy@20 of 0.81.

Xie, Y. et al. (2021) developed a log management framework, LogM, using a deep end-to-end neural network, they referred to as CAB net (Convolutional Neural Network (CNN) with attention-based Bi-directional Long Short-Term Memory) (Bi-LSTM) architecture and knowledge graph to learn the semantic and temporal dynamics from unstructured log data with the objective of failure prediction and diagnosis.

Intending to fill the gap of providing a real-time anomaly detection and anomaly root cause for a distributed multi-component system, they adopted Apache Hadoop, a distributed big-data processing multi-component system. Accordingly, the researchers collected and processed about 50 million log messages from 6 components of Hadoop (HDFS, Hive, MapReduce, HBase, Yarn, and ZooKeeper) in multiple steps.

In the first step, their one-pass logging invention was used to cluster the logs into anomalous and nominal log sequences. Next was a mapping of the sequence clusters by component and template categories. This step was followed by the aggregation of the templates based on timestamps, keywords, and labels. Lastly, a knowledge graph encapsulating the log data and prior knowledge was constructed. Afterwards, the CAB net was trained with the failure prediction log templates data and the failure diagnosis knowledge graph.

After training failure prediction with the CAB net involved determining the log group (nominal or anomalous) a test log message belonged to. Failure diagnosis, on the other hand, utilized both supervised and unsupervised techniques. In unsupervised failure diagnosis a log message vector needs to be correlated to similar root-cause labelled vectors in the knowledge graph, and the predicted root cause then being the majority root cause label for all correlated vectors. Supervised failure diagnosis could be achieved by training a Siamese LSTM network with a pair of labeled log message templates from the training data and knowledge graph, and the root cause inference is done with the trained network. Evaluating using the accuracy@k metric, which measures the models' accuracy on its ability to return at least one root cause message in requested k messages from the

knowledge graph shows that for k equal to 1, 2, or 3, the supervised model accuracy is 0.93 slightly better than that of the unsupervised model at 0.91.

2.4.5 Fault Classification

Defect or fault or bug categorization is the task of defining fault types based on certain attributes or features that the defect type expresses. Specifically for log categorization, the aim is to assign a defect type label to a log message or a sequence of log messages. Being able to classify bugs has several benefits. One of such benefits is the ability to quickly detect bugs, localize bug faults, and appropriately resolve them. A second benefit is in bug triaging for efficient management of software maintenance and bug resolution assignment. Another benefit of bug classification is that it aids in fault prediction and identification of the most error-prone components.

Gao, J. et al. (2019) research on defect classification delved into the 2 categories of defect classification – Manual defect classification and automatic defect classification. In manual classification, a human expert is required to define the taxonomy of defects and determine the defects that fall into defined classes.

2.4.5.1 Manual Fault Classification Techniques

Some manual defect classification methods described by Gao, J et al. (2019) include Orthogonal Defect Classification (ODC), IEEE standard defect classification, and Putnam classification.

ODC proposed by T.J. Watson, IBM, aims to classify defect types that are orthogonal and are all-encompassing. Orthogonality means the defect types are

independent, and the member defects are disjoint and non-overlapping. All-encompassing indicates, the categories should apply to all software products and be detailed.

The IEEE standard defect classification spans the entire SDLC and splits classification into defect classification and failure classification. Using one of these classes, exceptions can be classified into one of these eight categories: logic problems, computational problems, interface/timing problems, data problems, data processing problems, document problems, document quality problems, and reinforcement problems.

The Putnam classification groups defects based on the SDLC phase. Accordingly, types include requirement analysis defects, system design defects, algorithm defects, document defects, interface defects, and performance defects.

2.4.5.2 ML-based Automatic Fault Classification Methods

Tabassum, N. et al. (2023) conducted an empirical study focused on classifying cloud computing applications bugs using ML techniques. Ultimately, their objective was to cut down on or mitigate software security-related compromises, maintenance challenges, and reliability issues by automating bug finding through bug classification and prioritization. Their solution approach exploits both ML and NLP techniques.

With Synthetic Minority Oversampling Technique (SMOTE), they were able to address the data imbalance in their bug report training dataset, and subsequently vectorized the data via word2vec and TF-IDF. Afterwards, the resultant word vectors were used to train four different classical machine learning models: Naïve Bayes, Logistic Regression, Decision Tree, and Random Forest. The performance as measured

by accuracy was 0.88, 0.88, 0.91, and 0.88 for Naïve Bayes, Decision Tree, Random Forest, and Logistic Regression, respectively.

2.4.5.3 DL-based Automatic Fault Classification Methods

In general, DL-based bug classification models are either trained with bug reports or log data. Using bug reports, Du, X. et al. (2022) classified a class of bugs known as Mandelbugs. Their approach to tackling this challenge was to develop the DEEPSIM method, consisting of two parts: bug report data preparation and classifier training. For the data preparation part, they applied word2vec to train a DL-based skip-gram semantic model on over 2 million bug reports; and transformed each report into a 2-dimensional matrix, with the row dimension, the number of words, and the column dimension of the pretrained word embedding vector. In the second part of their solution, they trained a Convolutional Neural Network (CNN) on the data from part 1. Their evaluation analysis spanned several granularities. For example, on whether a bug report is bug/non-bug DEEPSIM F1 score was 0.897.

On the other hand, Ren, R. et al. (2018) in their research on the classification of system logs for distributed systems, developed a deep Convolutional Neural Network (CNN) to exploit the ability of DL-based networks to automatically learn patterns of bug signatures in the log data. The deep CNN employed for this task was built on the fault categories dictionary library created from the training log data. The dictionary's 13 fault categories for the log message events were manually created by human annotators. To train a classification model, a 130-dimensional semantic feature vector is constructed from a combination of each of the 13 categories' top 10 semantically relevant keywords'

similarity values. These similarity values are computed from the Levenshtein distance of their associated log messages. The best evaluation result for the trained model was on average 98.14%, 98.08%, and 98.11% for precision, recall, and F1, respectively.

2.5 Gaps in the Literature

This research focuses on using recent innovations in neural network architecture to handle fault classification and fault localization tasks.

For bug classification, the gaps revolve around the content and representation of the models' training data. Firstly, less attention has been paid to the effect that the preprocessing technique of the training data has on the performance of DL models. Another bug classification shortcoming in numerous studies of log-based software defect classification leveraging AI is the focus on using only the body or log text of the message for training data while neglecting the inclusion of information from the message header.

In numerous previous bug localization research exploiting log data to train DL models, the input feature vector into the model is constructed from a selection of the highest scoring tokens in a bug's stack trace. The scoring is generally performed using popular NLP techniques, like TF-IDF. Lessons from developers' success in manual defect localization might be a counter to this practice. Typically, in a bug's stack trace, there is a defect revealing log message generated from the bug location function/method in the logged trace. Generating the training feature vector from this bug revealing message provides several tangible benefits during inference. First, a knowledge base query using a bug revealing log message vector from an inference time bug's stack trace returns a ranked list with at least a matching log message vector that identifies the bug

location. Furthermore, the returned bug-identifying matches will be ranked more highly than other retrieved vectors. In contrast, when the query vector is constructed from highly ranked TF-IDF scored tokens, the retrieved ranked list might not include the vector representing the bug location or fault location vector might be ranked lower than other vectors in the list.

2.6 Summary and Conclusion

The advent of Transformer-based generative AI has sparked a surge of innovation in deep learning networks. Generative AI's versatility and efficiency in text data processing make it suited for analyzing unstructured log data. This current revolution in deep learning is particularly timely for the task of automatic defect diagnosis from log data because, as software systems adopt a distributed architecture and get deployed in cloud environments, they get more complex. This means that the runtime status logged by these systems also balloons in size, velocity, and variety, providing the big data needed to train deep learning models. Additionally, the bug surface of these complex systems increases in size and complexity, exacerbating the diagnostics tasks of locating and categorizing faults. One reason for this diagnostic difficulty and challenges lies in the fact that the logged symptoms of these so-called distributed software bugs are propagated across multiple nodes.

As such, many different diagnostic strategies have been employed to tackle bug troubleshooting. These strategies are multi-pronged in that they can be utilized before, during, and after a fault has happened.

The proactive strategies applicable to source code, source code metadata or software metrics are mainly designed to predict fault proneness of a software component at different levels of granularity: component, methods/functions, statements, etc. Meanwhile, another form of proactive diagnostics strategy aims to improve how a software system logs its runtime status, with respect to where log messages are located and what is logged. Both pre-mortem fault diagnostics strategies leverage both traditional software and ML methods for implementation.

The real-time methods for software fault diagnosis focus on reporting anomalies as they happen. One very important objective of these methods is to aid in the provision of high availability for production software systems. Log data is the source of input information to these Just-In-Time (JIT) defect reporting techniques. JIT diagnostics methods have been quite successful in the use of all the implementation vehicles to deliver solutions; including traditional software methods, statistical and machine learning methods, and DL-based methods. Their recent success in DL-learning methods is inspiring the use of DL methods in post-mortem diagnostics.

The inevitability of software bugs necessitates that innovative strategies be sought to classify defects and locate the root cause of defects. Then, apply the appropriate patch to prevent recurrence. Developers debug software by following the trail of bug symptoms from log events of their occurrence to the source code line or component where the fault lies. The trail might be uneventful and just be isolated to a single node. It could also be distributed among many nodes. Automated post-mortem diagnosis efforts mimic this strategy of trailing from logs to source code. But sometimes the trailing is restricted to being log-based. While these efforts have mainly successfully employed traditional

software methods such as slice-based methods, spectrum-based methods, etc. Some implementation methods have utilized classical ML methods and DL-based defect diagnosis, especially for defect categorization. One intention of this research is to use the recent Transformer-based method for log-based fault classification and localization.

Chapter 3—Methodology

3.1 Introduction

This chapter aims to document the methods used to develop the deliverable of this praxis, with the aim to address the research questions and hypotheses posed. The research questions and hypotheses center around utilizing AI-driven log data analysis for software fault troubleshooting. Specifically, the question of the model -Transformer-based or LSTM-based or BiLSTM-based - most suitable for classifying and localizing defects in a software system leveraging log data is posed. Additionally, there is interest in understanding how the processing method(s) to which the training log data is subjected to affects the performance of a fault diagnosis model.

The rest of this chapter is organized as follows. Section 3.2 explores the processing methods leveraged to prepare the log data and the source code from which the log data is generated. Section 3.3 highlights the various methods and technologies, such as Transformer-based architecture, LSTM-based architecture, BiLSTM-based architecture, and RAG vector DB, uniquely adapted in this research for software fault diagnosis. Lastly, section 3.4 builds on the preceding section with the aim to present the specific steps that were undertaking to exploit the technologies for model training, validation and testing.

3.2 Data Preparation and Exploration

The data used for this empirical study is from the open-source HDFS system. HDFS is a component of the multi-component Hadoop distributed data processing

system. The function of HDFS in Hadoop is to provide file management and storage services to the other components. Its source code is publicly available in a GitHub repository.

3.2.1 HDFS Log Data

In Hadoop Distributed File System (HDFS), files are made up of blocks, which are the smallest units of data. Each of these blocks is stored independently on different DataNodes. Blocks, in turn, are replicated across DataNodes by the NameNodes to enforce data redundancy and facilitate fault recovery and data loss prevention.

In the event of a fault, a data block's fault diagnosis leverages its associated HDFS log dataset for troubleshooting. Depending on the content of a block's log messages, its state can be classified as either nominal or non-nominal. A nominal block is a block operating nominally, while a non-nominal block is one having some fault as reflected in at least one message in its log message sequence.

The log data utilized in this study was generated by Zhu, J. et al. (2015) in a private cloud environment using benchmark workloads and made publicly available on GitHub to facilitate log data research. Post-generation, the raw data was preprocessed by slicing based on block IDs into traces and creating manually handcrafted rules for labeling the log dataset points as either anomalous data points or nominal data points. An example of a nominal HDFS log message and a non-nominal HDFS log message is shown in Table 3-1. As illustrated by the example, log messages are usually freeform unstructured text or semi-structured text made up of a header (with fields like timestamp, process ID, etc.) and a body.

Table 3-1. Typical HDFS Log Message Structure

	DATE	TIME	PID	LEVEL	COMPONENT	CONTENT
NOMINAL	0811 09	2041 32	26	INFO	dfs.FSNames ystem	BLOCK* NameSystem…
ANOMALY	0811 09	2204 03	31	WARN	dfs.DataNod e$Dat…	10.251.107.227:50010: Got excepti…

3.2.2 HDFS Log Data Pre-processing for Fault Categorization

In most software systems, log data is generally imbalanced in favor of normalcy. From Table 3-2, it can be deduced that HDFS defect log messages are only about 3% of all the block log data.

Table 3-2. Count of Anomaly and Nominal Block Sequences

Anomalous Block Count	Nominal Block Count	Total Block Count
16838	558223	575061

The HDFS 16,838 anomalous block data comprises 26 defect types, of which just six defect types make up 15,346 (about 91%) of all anomalous blocks as seen in Table 3-3.

To demonstrate the severe imbalance in the HDFS log data, Figure 3-1 provides a visual representation of the fault label frequency. Note that the ‘Other’ label is a class for datasets IDs with examples count of between 1 to 34 block(s). To alleviate the imbalance

issue only labels with at least 900 examples are employed for model building. This includes the six labels with original IDs of 5, 31, 3, 0, 4, and 1 in order of decreasing block examples.

Table 3-3. Count of Fault Labels

ID	Block Frequency	Label
5	4167	Missing Block Info
31	3225	IO Failure
3	2950	Unexpected Early halt
0	2809	Invalid Set Deletion
4	1240	Orphan Block Request
1	953	Redundant Block Request
21	724	Serve Block Issue
7	476	Empty Packet Receipt
12	130	Replicate Block Issue
8	45	Replication Monitor Time-out
9,16,13,18,22,19, 20, 10, 27, 7,11,24, 25,28,30, 23	34,22,10,9,9,8,7, 3, 2, 3,3,2,2,1,1,1	Other

3.2.2.1 Fault Classification Data Preprocessing Algorithm Description

Using Python each line of the HDFS log file is read, clustered on block IDs, then each cluster is sorted on time of occurrence. Finally, each cluster is separated into an anomalous and nominal groups and written to files sorted_anomaly.log and sorted_normal.log, respectively. Also to simplify data labelling, each cluster block in the

anomalous group is duplicated in its own file with naming nomenclature,

<BlockID>_type_<typeID>.

Fault Classification Data Preprocessing Algorithm

Input: HDFS Log file, Log Types file

Output: Sorted anomaly/nominal logs files, sorted files of Block ids log messages

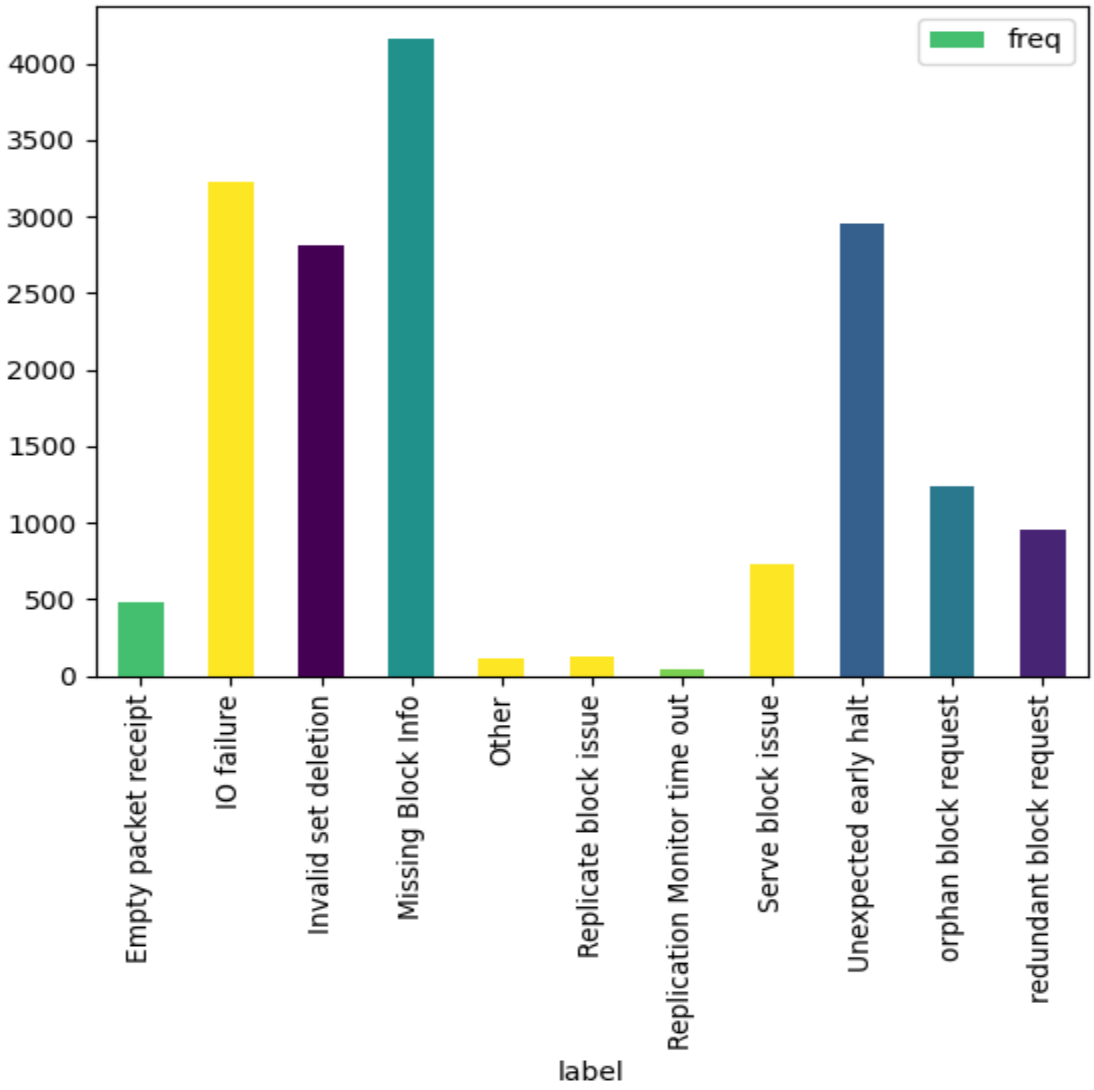

Figure 3-1. Bar plot of Defect Types Block Frequency

1. Read log type(normal/anomaly) file into Python dict mapping block IDs to label
2. Cluster log messages from HDFS log file on block ID
3. Sort each cluster on time field of its log message
4. Open sorted_anomaly.log and sorted_normal.log for write
5. For each block cluster
6. Get its normal/anomaly label from Python dictionary
7. If the label == ‘normal’
8. Write the cluster log messages to combined sorted_normal.log
9. Else if label == ‘anomaly’
10. Write the cluster messages to cluster file named, BlockID_type_typeID.log
11. Write the cluster messages to combined sorted_anomaly.log

The raw log messages that end up in these generated block files are not fed directly into the models as this would bias the models due to the presence of noisy data: numeric values like date type, IP addresses, variable values, etc., punctuation marks, and repeated log messages. Rather, the noise in the log data is removed either during deduplication operations (for repeated logs) or parsing. Parsing of the raw log data in the generated files is handled using the two methods depicted pictorially in Figure 3-2.

One of the methods is a custom parser written in Python. The Python code replaces semantically meaningless characters, such as punctuation marks, with white space and removes numeric values in each log message. It also splits camel-case formatted compound words into their constituent elements.

The other log parsing method, known as Drain (a fixed-depth tree-based online log parsing method), also written in Python, was pioneered for log processing in the

heavily studied anomaly log detection domain. He, P. et al. (2017) created Drain to speed up log parsing, and it is very performant in both online and offline processing of software logs. To achieve this accelerated parsing speed Drain makes use of a fixed-depth parse tree, which encodes specially designed parsing rules. Its output message templates retain the static part of the log message while replacing variables resolved at runtime with a placeholder. Just like in many other log message parsing techniques, messages are clustered into groups after having stripped out the variable part of the message to create the message template or event. In the case of Drain, regular expressions based on domain knowledge are exploited to create a message template for newly arrived log messages. Then the tree is traversed by the encoded rules to arrive at a leaf node (which in Drain is a message group) that message belongs to. Traversal could also result in the creation of a new leaf node for non-matching templates.

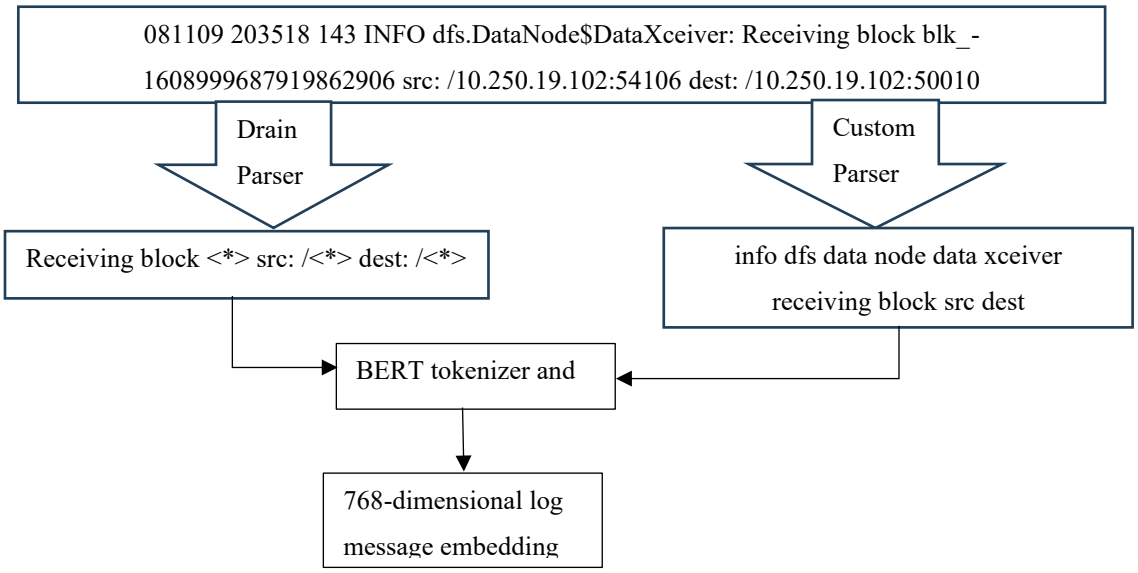

Figure 3-2. Log Message Parsing Output for Drain and Custom Parser

While both methods extract the static portion of log messages, the custom parser output includes additional contextual information that can potentially boost the models' diagnostics capabilities.

To get the data into a form usable by the models, the final step of data preprocessing creates a 768-dimensional embedding vector for each block log sequences with HuggingFaces pretrained Transformer-based BERT tokenizer and encoder. After the BERT tokenizer tokenizes each log message, which are restricted to the max word length of 512 words either through truncation for longer sequence or padding for shorter ones, the BERT model transforms the log message tokens into a 768d semantic vector embedding. Then the final vector of the log message is computed as the mean of all its 768d tokens. The algorithm below shows how this happens for all the blocks raw messages sequence.

Fault Classification Data Vector Embedding Algorithm

Input: HDFS anomalous block files
Output: Numpy compressed file of blocks log message and associated fault

1. For each block file
2. If block id is in [5, 31, 3, 0, 4, 1]
3. Process log messages either with drain or custom code and deduplicate
4. Create vector embedding with BERT for each message and sum the vectors for the log messages in a sequence.
5. Write the vector to a Numpy array
6. Persist the resultant Numpy array to a compressed Numpy file.

3.2.3 HDFS Log Data Vocabulary Analysis for Fault Localization

The fault localization models require both the anomaly block log messages and

their summary for training. Summarization of the log messages makes heavy use of Natural Language Processing (NLP) to extract the bug-revealing log message from a block's log sequence. Table 4-7 and Figure 4-7 present some descriptive statistics and visuals of the fault localization data set.

3.2.3.1 Fault Localization Preprocessing Algorithm Description

The fault localization log data preprocessing algorithms are written in Python. It is a multi-step processing algorithm that starts with splitting multi-delimited concatenated words into their component words using Python's Regular Expression. Next step does extractive summarization of a blocks' log sequence using Term Frequency-Inverse Document Frequency (TF-IDF) to score the terms in the fault-delineated vocabulary the block messages sequence belongs to. Then, the log message in the log sequence with the highest TF-IDF is selected as the log summary for two reasons: TF-IDF scoring is applied to a fault vocabulary as opposed to the global vocabulary and the summary fault-revealing log message contains more fault-specific words.

With the summary done, the next step is to determine the localized method name responsible for generating the log messages by attempting to match the method's extracted log messages to the most likely runtime incarnation written to the log file. This process of finding a suitable fault triggering method starts with a search for a method name in the fault's runtime log sequence guided by the extracted summary log message. The goal is to locate a class name/method name pair from the runtime logs that matches a class name/method name pair extracted from source code and the guiding summary. Even with a match found for a specific method log, the search space is expanded to include all

the other extracted method logs by scoring them against the extracted summary using Jaccard similarity. The extracted methods log messages meeting a threshold score of 0.5 or with the highest Jaccard score are chosen. With multiple retrievals, a second comparison of the retrieved method logs against the runtime logs, again using Jaccard, is done with the higher scoring method chosen.

After the fault related log message and its generating method is determined, the data needed for the encoder-decoder fault localization models is complete. The final data processing step just combines each block's file content, its extractive summary, and generating method name in a single Comma Separated Values (CSV) file.

3.2.4 HDFS Source Code Data

The log data from Zhu, J. et al. (2015) was created at runtime by the HDFS source code written in Java language. Given the evolvability in source code due to the need to fix bugs, add new features, etc. it is clear that a specific version of HDFS source code was responsible for this log data generation. Thus, it is essential that the version of the software that produced the log data be leveraged for defect troubleshooting. This version was determined to be Hadoop version 1.0.2 after a rigorous and careful search was undertaken to find a match between Zhu, J. et al. (2015)-provided log message text and various HDFS source code log statements arguments.

Following the pull of the source code of Hadoop 1.0.2 from GitHub, TreeSitter library was used to parse and extract pairs of methods and their associated log messages from just the HDFS component. From the 182 Java files, 2264 Java methods were extracted with just (271) 12% of all these methods containing logging statements.

3.3 Hypothesis Verification Methods and Tools

This section gives a brief highlight of the key empirical methods and tools that were leveraged at both the Exploratory Data Analysis stage and the model building stage. However, there are many other tools such as Google Bidirectional Encoder Representation from Transformers (BERT), TF-IDF, Stanford's Global Vectors for Word Representation (GloVe), etc. that were utilized but will not be highlighted here.

3.3.1 Encoder/Decoder Architecture

In machine learning, sequence-to-sequence tasks in which an output sequence is generated from an input sequence require models that leverage an encoder-decoder architecture. Examples of tasks meeting this criterion include machine translation, text summarization, question answering, text generation, etc. Whereas the encoder is responsible for generating an intermediate representation of the source sequence, the decoder's objective is to autoregressively generate the target sequence.

Using the encoded source sequence and previously predicted target sequence tokens, the target sequence is generated one token at a time by the decoder. Specifically, a token at the current position (say position i) for the target sequence is predicted using the encoded source sequence and tokens at the preceding positions (tokens at position 0 to position i-1).

This prescribed sub-models pairs training-inference life cycle guidance was followed by the developed encoder-decoder LSTM log message summarization model

and the encoder-decoder Transformer-based summarization neural nets to tackle fault localization.

3.3.2 Tree-sitter Parser Generator

Tree-sitter is a parser for generating concrete syntax tree from source code. Building a syntax tree representation of Hadoop HDFS Java source code involved feeding it into Tree-sitter. Then the resultant syntax tree was parsed in a dual process: methods definitions extraction from the syntax tree is followed by the retrieval of their log statements arguments static text.

3.3.3 Milvus Vector Database

Milvus developed by Zilliz is a vector database with both open-source and cloud service options available for users. The three deployment options are as a library, a single machine complete vector database, and an enterprise grade distributed vector database.

The library deployment of Milvus vector DB, distributed as Python Software Development Kit (SDK) known as PyMilvus, is utilized for storing and retrieving the output from the HDFS Java source code parser. Two main utilities leveraged from PyMilvus include MilvusClient (for creating and connecting to the DB, populating the DB, and retrieving matching records) and model.DefaultEmbeddingFunction (for generating text embeddings using Reimers & Gurevych (2019) sentence-transformer / paraphrase-albert-small-v2 model). Each record is persisted to the vector database and includes the fully qualified method name; the method logs static text and 768-dimensional vector representation of the runtime log message summary. Then, when a fault occurs, the

localization model's predicted summary for the faults' log message sequences is converted into a query vector to search the Milvus database for matching persisted summaries. The returned records are a clue to the faults' triggering methods.

3.3.4 Keras Deep Learning API

Keras is a Python-based multi-framework deep learning (DL) API that is built on top of JAX, TensorFlow, or Pytorch. It is used to develop modular components from its two main data structures: layers and models. The choice of Keras API utilized for building the classification and localization models is the flavor built on TensorFlow.

After a DL model architecture is defined by one of the three provided methods: Sequential Model, functional API and object orientation subclassing - pre-training configuration is performed on the model via its 'compile' method which takes a differentiable loss/objective function, an optimizer, and metrics. Then, model training is conducted using the 'fit' method. At a minimum 'fit' requires the following arguments: inputs and targets (or labels) to train on, the number of epochs or training, and the batch size to use (which can vary from one sample, a mini-batch or all the samples). Post-training a model enters the inference stage of its life cycle, achieved in Keras by calling 'predict' on it.

3.4 Model Training, Validation and Evaluation

All the models were built on Keras with Python in Google Colaboratory (or Colab). Colab is a free browser-based Jupyter notebook service that runs in the cloud making it accessible from anywhere internet (web) service is available. The service

provides CPU, GPU and TPU runtimes. As depicted in Table 3-4 below, of the original anomalous 15, 346 block log message sequences, 80% (12275 block log sequences) were allocated to training and 20% (3071) for model evaluation. The training data in turn underwent a 90-10 split to yield the final training-validation data.

Table 3-4. HDFS Data Allocation

Training	Validation	Evaluation
11,047	1228	3071

In general, all the models are generated through iterative training in multiple epochs, and each epoch consists of multiple mini-batch iterations. A model's training concludes after the validation phase, in which the trained model's performance is judged with respect to its accuracy on the validation dataset.

3.4.1 Fault Classification Models

The fault classification problem can be handled as a text classification task. A text classification task is a supervised learning task in which the intent is to predict the topic of the text. By analogy, the objective of fault classification is to predict the class of a bug based on the fault's log messages. To formalize fault classification, let $S = \{s_1, s_{2,,} \dots, s_{n_1}\}$ denote a sequence of HDFS blocks log message sequence where s_1 are log messages of block 1, s_2 are log messages of block 2, … s_n are log messages of block n; for all the blocks with fault. Also let $T = \{t_1, t_{2,,} \dots, t_{n_2}\}$ denote the fault type that describes the topic of text of S sequences. Then, fault classification is a supervised learning task that attempts to learn a prediction function, $f : S \to T$.

3.4.1.1 Classification Performance Metrics

The bug classifiers performance on predicting the right defect were graded on the standard classification evaluation metrics : precision, recall, F1 score and accuracy. Calculation for these metrics requires the initial computation of these 4 values: true positive (TP), false positive (FP), true negative (TN) and false negative (FN).

In the context of bug classification, TP indicates the number of anomalous log messages of a specific defect type that are correctly predicted as belonging to that defect class, FP are the number of anomalous log messages of a specific defect type incorrectly predicted as members of the different class, TN is the number of anomalous log messages for a defect type correctly labelled as not belonging to that defect type, and FN indicates the number of anomalous log messages of a defect type incorrectly predicted as belonging to a different defect class.

Having determined these values the formula for the metrics (see appendix A) can be deduced by specific combinations of the values.

3.4.1.2 Training, Validating, and Testing the Bug Classification LSTM Model

Figure 3-3 depicts the step-by-step training and testing process for the Bi-LSTM classifier built with Keras. The training starts off with the input layer receiving a 768d log message sequence embedding vector and goes through a series of transformations in a 2-layer 128-neuron Bi-LSTM, followed by the dropout layer whose dropout value is set at 30%, before finally reaching the output layer responsible for determining the fault type of the log message. This layer uses softmax to predict the probability of each of the six fault types being the class of an input log message. In the figure, emphasis on how a Keras

model is constructed from an input layer and an output layer is shown using the right brace to group components from the Input Layer and the output (Dense) layer. Also shown are the details of the bidirectional bug classification model pre-training configuration and training via the model's 'compile' method and 'fit' method, respectively.

After training the Bi-LSTM model performance is evaluated on originally set-aside 3071 log message blocks testing data. Evaluation results are a classification report produced from a comparison of the predicted faults relative to the actual classification labels.

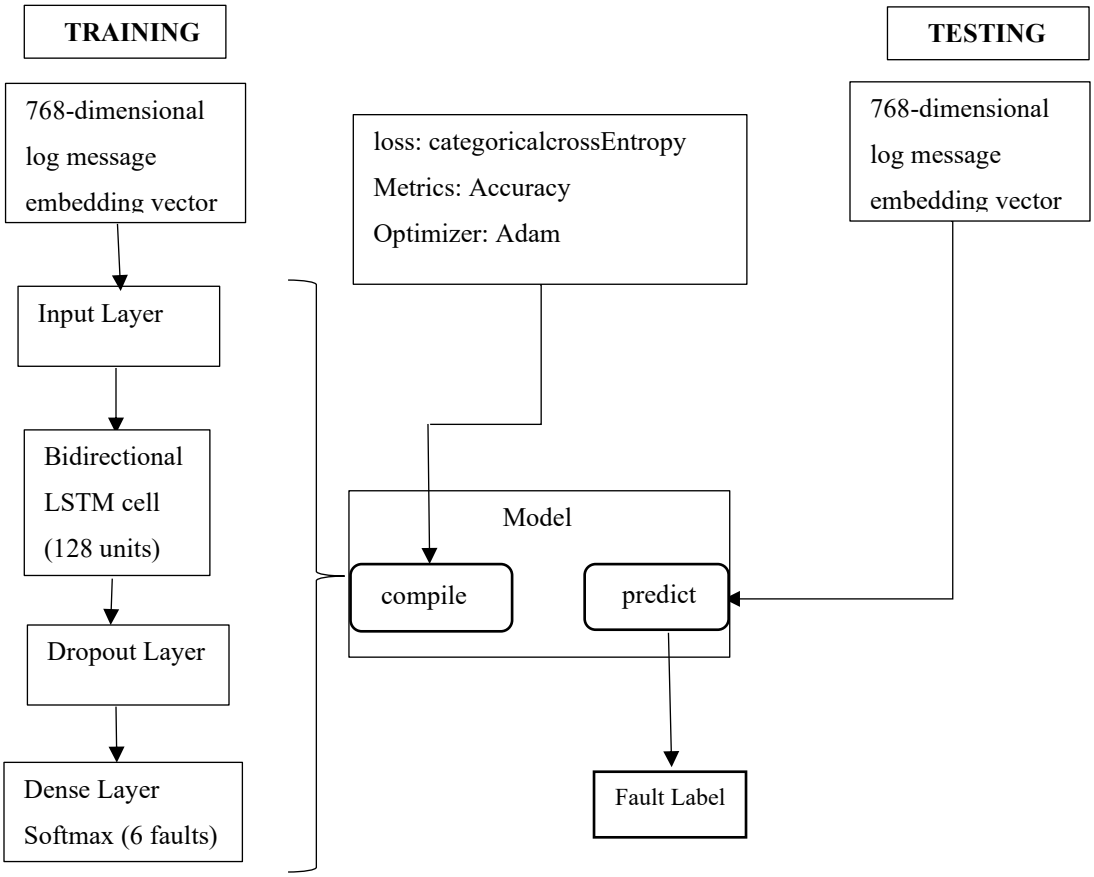

Figure 3-3. Classification Bi-LSTM Training Process

3.4.1.3 Training, Validating, and Testing the Bug Classification with Transformer

The Transformer encoder is a generic component that is immeasurably useful in text understanding tasks, like text classification task, which fault classification has been formally established to be. The architecture of the defect classification encoder (as in any

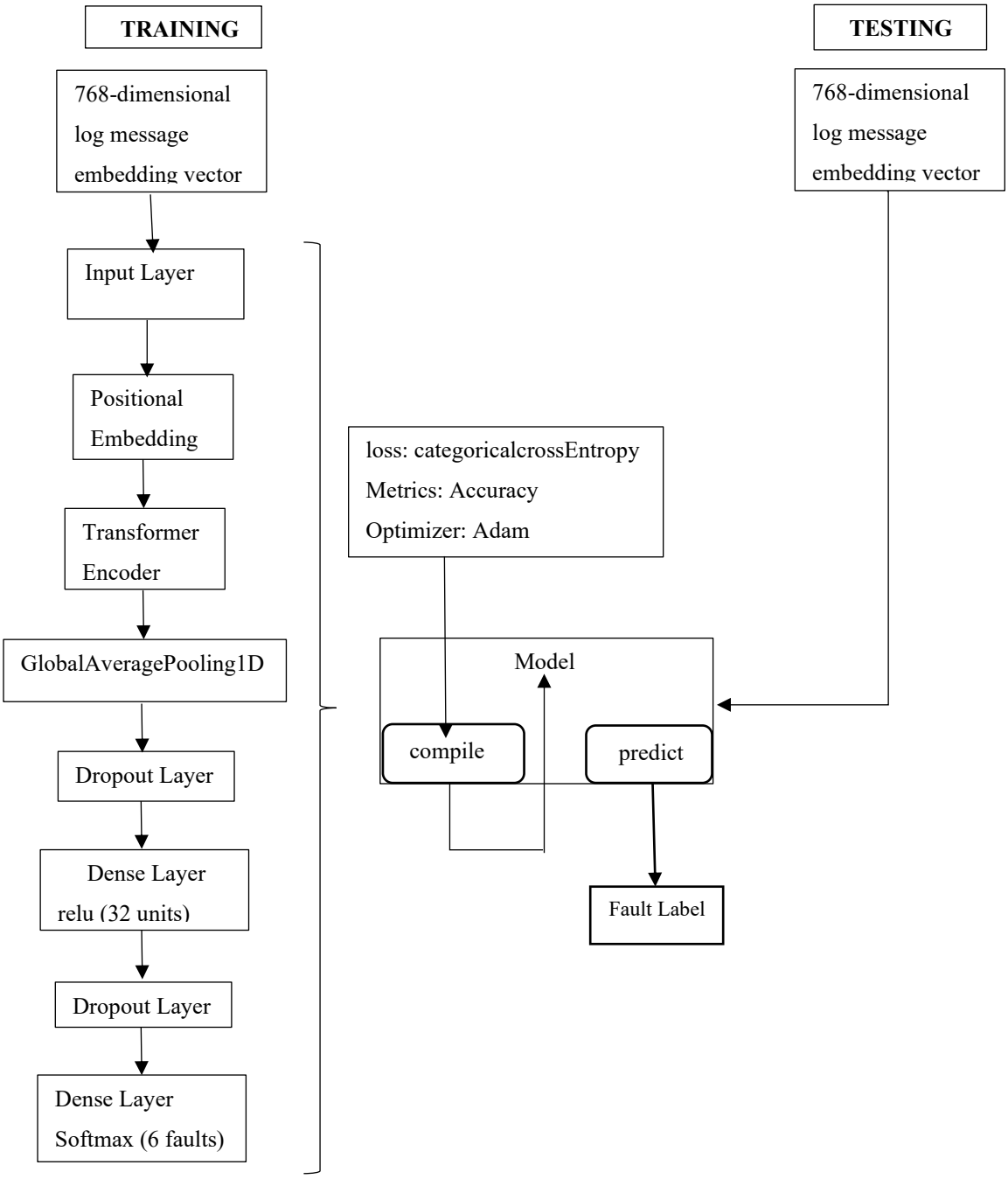

Figure 3-4. Classification Transformer Training Process

generic encoder) comprises of the multi-head attention layer, the layer normalization layer and two Dense layers.

With the encoder created, the defect classifier can be built by combining the encoder with Input layer, a positional embedding layer, global average max pooling 1D layer, and a dense layer. As shown in the Transformer training process in Figure 3-4, training is realized in a step-by-step input transformation through all the layers. Then, post-training for any received input, its probability of being in any of the six fault types is inferred via SoftMax activation in the output layer.

3.4.2 Fault Localization with Sequence-to-Sequence Models

To pinpoint a fault location, the proposed fault localization encoder/decoder model aims to define the task of locating the defect inducing method name using sequence-to-sequence learning. Actualizing this proposal requires the pursuit of a supervised learning strategy that combines an encoder-decoder summarization and search in a technique denoted as search-enhanced summarization (SES). In SES context, the summary of a log messages sequence is the fault-revealing log message in the sequence that pinpoints the potential log point location in source code from which the fault originated.

In SES method, the idea is that at training time the encoder takes as input the log message sequences from the runtime log and outputs an intermediate encoding. The decoder, on the other hand, learns to generate a different sequence, a summary of the log message sequences, guided by a provided summary and the encoder log encodings. Then at inference the decoder predicted log summary is utilized as a query into a vector

database. Each record in the RAG-based Milvus vector DB is precomputed prior to training. These database records store information about source code methods that emit log messages. Specifically, the stored information includes the fully qualified method names, the methods extracted static log message or template, and the vector embeddings of the summary of the runtime log message generated from the method log template.

To retrieve records from the vector DB, the similarity score between the vector embeddings of the predicted log message summary and vector embeddings of summaries embedded in the DB record is computed. The returned search result is a ranked list of the K records with the highest score. A measure of how well the encoder-decoder models can localize faults from runtime log messages is predicated on whether the returned ranked records include at least one record with the buggy method that generated the runtime log.

Huo et al. (2016), formulated bug localization in terms of a set of source code, $C = \{c_1, c_{2,,} \dots, c_{n_1}\}$ and bug reports, $R = \{r_1, r_{2,,} \dots, r_{n_2}\}$ as a learning task which aims to learn a prediction function, $f = C\ X\ R \rightarrow y$ where $y_{ij} \in\ y = \{+1, -1\}$ indicates whether a source code file, $c_i \in C$ is relevant for a bug report, $r_j \in R$.

Taking inspiration from Huo et al. (2016), the proposed search-enhanced bug localization learning task leverages an encoder-decoder model to generate a query from a bug's log sequence utilized in a search to localize the source code method responsible for emitting the log sequence and therefore potentially triggered the bug. To formularize, let $S = \{s_1, s_{2,,} \dots, s_{n_1}\}$ denote a sequence of HDFS blocks log message sequence, $Q = \{q_1, q_{2,,} \dots, q_{n_2}\}$ denote the log sequence summary representing a query into a store of method name, summary embeddings, and log message template triplets. Let $M = \{m_1, m_{2,,} \dots, m_{n_3}\}$ denote a set of source code methods. Then, the function learnt by the

sequence-to-sequence fault localization model can be formulated as $f = S\ X\ Q \rightarrow y$ where $y_{ijk} \in\ y = \{m_1, m_{2_{,,}} \ldots, m_{n_3}\}$ indicates whether a log summary query, $q_i \in Q$ produced from a log sequence, $s_j \in S$ is relevant for retrieving buggy methods, $m_k \in M$.

3.4.2.1 Fault Localization Performance Metrics

Predicted summaries are scored with the accuracy metric known as accuracy@k. Accuracy@k measures the percentage of generated log sequences summary able to locate at least one buggy source code methods in the top k (k=1,3,5) ranked method names returned using the summary as a query into a knowledge base. Hence, a predicted summary is said to match the top k accuracy ranked list if it retrieves a top k query result that contains at least one correct source code method from which the summary, and thus the bug emanated. A higher accuracy@k value indicates better performance and effectiveness on bug localization by the model.

3.4.2.2 Building and Evaluating the Fault Localization LSTM Encoder-Decoder

The fault localization encoder is a 6-layer deep 128-units LSTM cell; each regularized via training dropout rate of 40%. In the layer's setup, the initial encoder LSTM cell consumes the input log message sequences while successive LSTM layers use as input the output of a preceding layer. Thus, in the localization LSTM, inputs are used for learning by passing through various representational states in the model's encoder layers before finally resulting in summary generation in the decoder layers. Figure 3-5 details the training and inference process of the LSTM summarization model.

At inference time, the log message summary is generated from scratch in several steps. First the inference log encoder model encodes the incoming log message

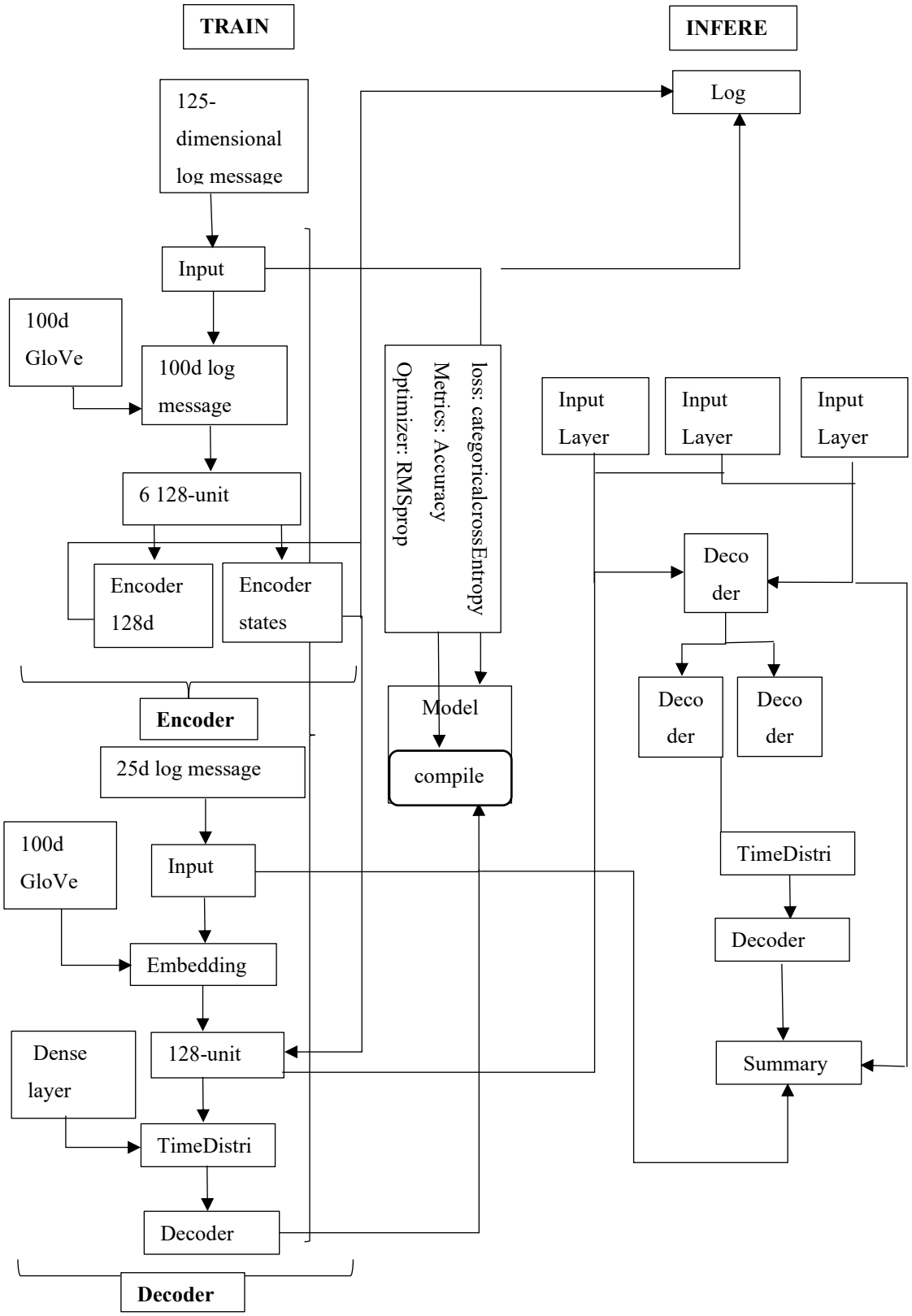

Figure 3-5. Log Message Summarization Encoder-Decoder LSTM Training Process

into an output sequence, hidden and cell state. Next the inferencing summary decoder model predicts the next token using these outputs and an initial seed token (referred to 'sostok'). Then each previously predicted token is iteratively fed back into the inferencing decoder to predict the next step token until either the end of sequence token (referred to 'eostok') is predicted or the number of tokens predicted thus far is one less than the max log summary length. The log summary is then a concatenation of the predicted tokens in the order of their prediction.

3.4.2.3 Bug Localization Encoder-Decoder Transformer Summarization Models

Google's T5 model is a sequence-to-sequence encoder-decoder Transformer model. Pretraining T5 on the Colossal Clean Crawled Corpus (C4) dataset endows it with the capability to handle many text-to-text tasks including question answering, machine translation, summarization, etc.

The defect localization summarization encoder-decoder Transformer models, unlike LSTM sub-models, were not built from scratch. But instead a T5 base model was retrieved from HuggingFace and finetuned on the log message sequences.

Prior to finetuning T5 on the log data, the data was loaded into a Keras Dataset and tokenized with T5 tokenizer. Next step involved the definition of the following required training parameters: output directory for intermediate and final training results and logging, number of epochs, training and evaluation batch sizes, warmup steps, weight decay, logging frequency, evaluation strategy, save strategy, reporting technique, learning rate, and data load workers.

To finally start training, the defined training arguments in addition to the downloaded t5-base T5 model, the tokenized training and evaluation datasets, and ‘rouge’ metrics function were passed as arguments to the T5 trainer. Training execution was performed in Google Colab using T4 GPU with High RAM. As configured, the trainer saved 2 of the best models in the specified output directory at training completion. Following fine-tuning, inference was done with the saved T5 models on the evaluation data, also loaded into a Keras Dataset.

Chapter 4—Results

4.1 Introduction

Defect diagnosis, whether done manually or automatically through software or ML is a difficult task, sometimes made infeasible to achieve with a restriction to perform using just log data. To overcome this challenge using ML the log data and its generating source code needed to undergo a series of transformations before a representative state that the models can learn from was reached. For fault classification which requires just the runtime log data, this data needed to go through several processing stages: removing its dynamic portions, breaking up compound words into their simple components, deduplication and transformation into embedding vectors representations. Fault localization, on the other hand, requires both the runtime log data and fault location data, which, as previously mentioned, were approximated by the log points method names in the generating source code. Similar to the classifier log data processing, the fault localization models runtime log data had dynamic portions removed, compound words unraveled, and duplicates dropped. However, its location information processing involved extracting the methods' log message templates while retaining only the static portion of the extracted pre-runtime log message template. Afterward, the runtime incarnation summary, static log template data and containing method names are persisted in a knowledge repository, Milvus vector DB to be used for facilitating learning in the fault diagnosis models.

For both the trained bug classifiers and bug localizers, how well they learnt from the data was gauged with several standard metrics already defined in section 3.4.1.1 and

3.4.2.1, respectively. The criteria for metric selection rested upon the simple requirement that it enabled evaluation that made it easy to grade the models on predictions accuracy. An accurate prediction has the effect of narrowing the defect search space. This of course meant that most of the time, the right bug class and correct top emanating bug method(s) were predicted by the defect classification and defect localization models, respectively.

4.2 Data Analysis Results

The fault classifiers fault-specific log data analysis details covered in the next six sections highlight each fault discriminating features. Such an insight might provide an interpretability and explainability lens into how the classification model works. Then starting in section 4.2.7, the corresponding descriptive statistics and charts of preprocessed log data and its generating methods for the fault localizers are presented.

4.2.1 Invalid Set Deletion Defect Data Analysis Results

With a vocabulary size of 48 words and an increasing word count as one navigates from a lower quartile towards an upper quartile, there seem to be no uniformity in the log messages content for the HDFS blocks expressing this defect type, as demonstrated by the statistics in Table 4-1. Additionally, the interquartile range (IQR) (112 - 96 = 16) confirms that both the minimum length log messages and maximum length log messages are not outliers, as they do not fall outside 1.5 times the IQR below quartile 1(Q1) or above quartile 3 (Q3), respectively. Also, the histogram on the right of figure 4-1 illustrates the bimodality of the invalid set deletion fault tokens word count while the word cloud on the left emphasizes by font size the frequency of each token.

Table 4-1. Invalid Set Deletion Word Count Length Statistics

Descriptive Statistics	Value
Max Log Message Word Length	132
Min Log Message Word Length	85
Log Message Q1 Word Length	96
Log Message Q2 (Median) Word Length	103
Log Message Q3 Word Length	112
Log message Average Word Length	105
Log Messages Vocabulary Size	48
Count of Message Sequences	2810

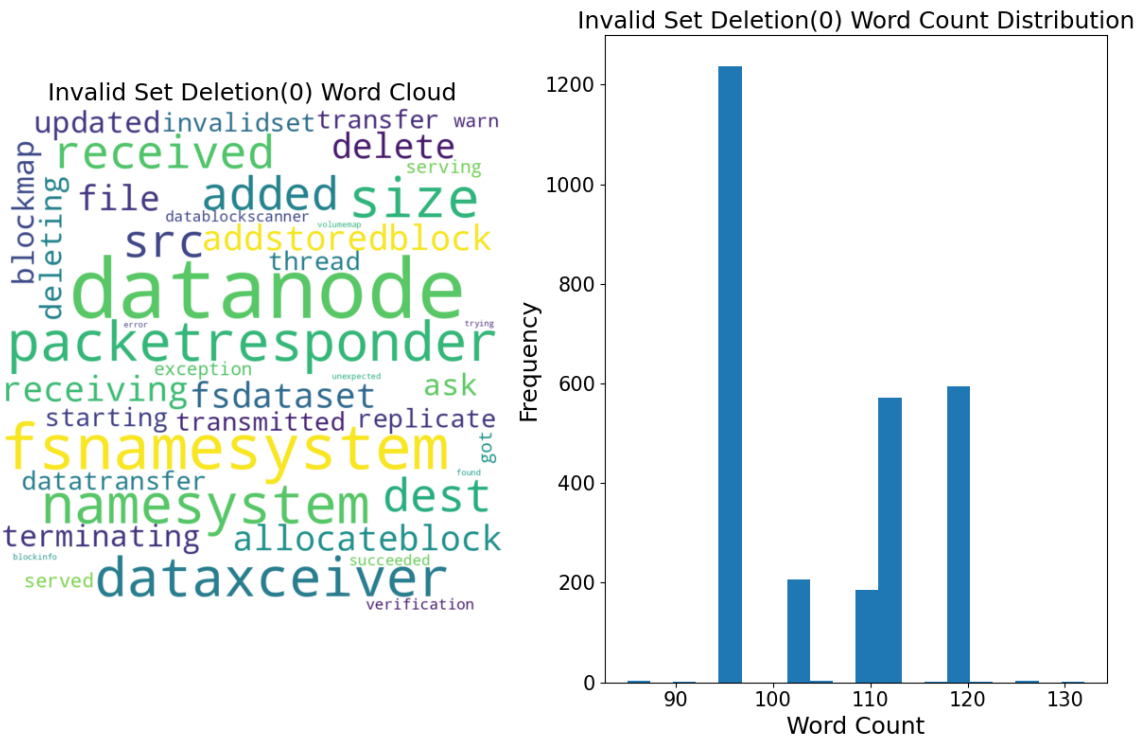

Figure 4-1. Invalid Set Deletion Word and Word Count Length Distributions

4.2.2 IO Failure Defect Data Analysis Results

The IO defect type statistics on Table 4-2 and visualizations in Figure 4-2

Table 4-2. IO Failure Word Count Length Statistics

Descriptive Statistics	Value
Max Log Message Word Length	365
Min Log Message Word Length	29
Log Message Q1 Word Length	29
Log Message Q2 (Median) Word Length	29
Log Message Q3 Word Length	29
Log message Average Word Length	29
Log Messages Vocabulary Size	68
Count of Message Sequences	3226

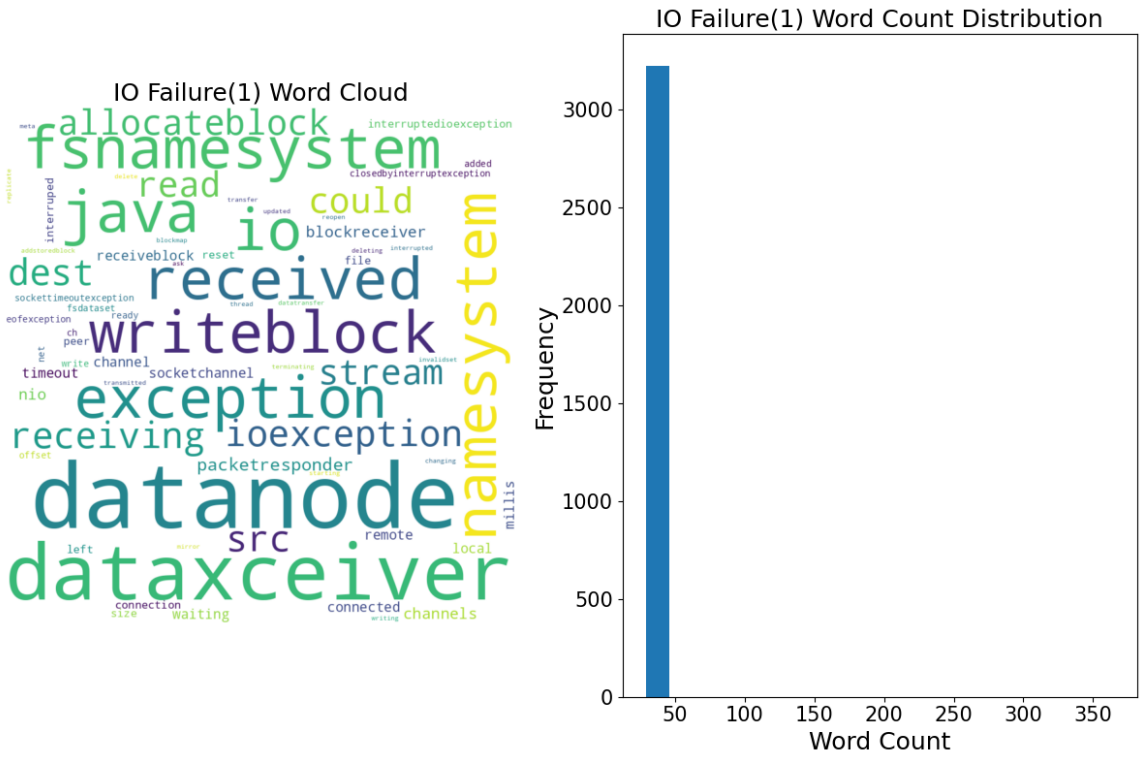

Figure 4-2. IO Failure Word and Word Count Length Distributions

describe a defect type having over 90% of blocks with a word count equal to the minimum and average word count. With a prominent min-max range of 336 (365-29) and an IQR of 0, this defect (statistically) has some outlier HDFS block(s). Also, the vocabulary size being more than twice the length of a majority of the messages means there is dissimilarity in the token composition between and within the blocks' log messages. Lastly, unique to this defect are tokens such as 'io', 'exception', and 'read' as seen on the word cloud on the left of Figure 4-2.

4.2.3 Missing Block Info Data Analysis Results

As presented in Table 4-3, missing block info bug vocabulary word count values for first, second and third quartiles are the same and equal to the minimum word count value.

Table 4-3. Missing Block Info Word Count Length Statistics

Descriptive Statistics	Value
Max Log Message Word length	132
Min Log Message Word length	74
Log Message Q1 Word Length	74
Log Message Q2 (Median) Word Length	74
Log Message Q3 Word Length	74
Log message Average Word Length	80
Log Messages Vocabulary Size	42
Count of Message Sequences	4167

Further, the histogram on the right of figure 4-3 establishes that about 75% of log messages in this bug type have a minimum word count of 74. On the other hand, the maximum word count of 132 seems to be shared by less than 25% of these defect log

messages given that the average word count value of 79 is quite close to the minimum word count. Also, its vocabulary size of 42 is almost about half its minimum word count value of 74, indicating that many log messages repeat the same tokens. From the word cloud on the left of Figure 4-3, ‘blockinfo’ token can be verified as been unique to this bug.

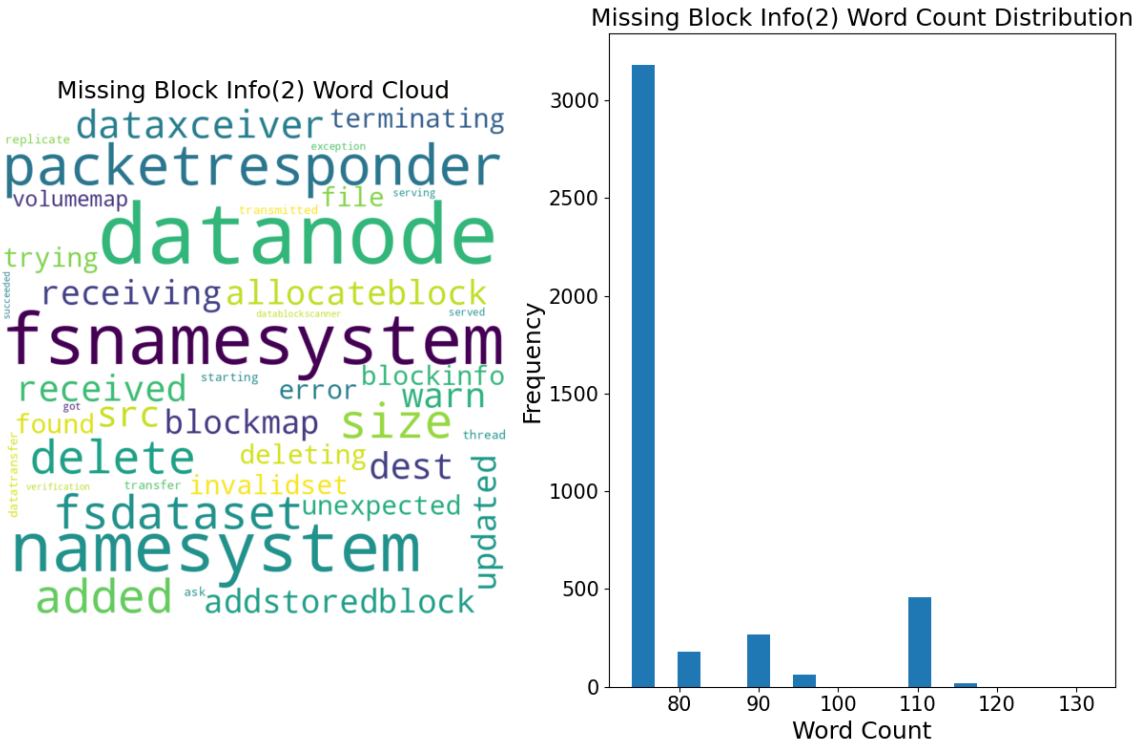

Figure 4-3. Missing Block Info Word and Word Count Length Distributions

4.2.4 Orphan Block Request Defect Data Analysis Results

For the orphan block request, its statistics on Table 4-4 show that 25%, 50% and 75% of the log messages have word counts less than 86, 96, and 96 respectively. Additionally, the histogram makes known that almost half of the blocks (500 of 1240) with this defect type have word count value close to the median. Lastly, its histogram in

Figure 4-4 shows its word count frequency distribution is bimodal and the word cloud on the right of this Figure 4-4 illustrates the frequency of its vocabulary tokens.

Table 4-4. Orphan Block Request Word Count Length Statistics

Descriptive Statistics	Value
Max Log Message Word Length	134
Min Log Message Word Length	69
Log Message Q1 Word Length	86
Log Message Q2 (Median) Word Length	96
Log Message Q3 Word Length	96
Log message Average Word Length	92
Log Messages Vocabulary Size	46
Count of Message Sequences	1240

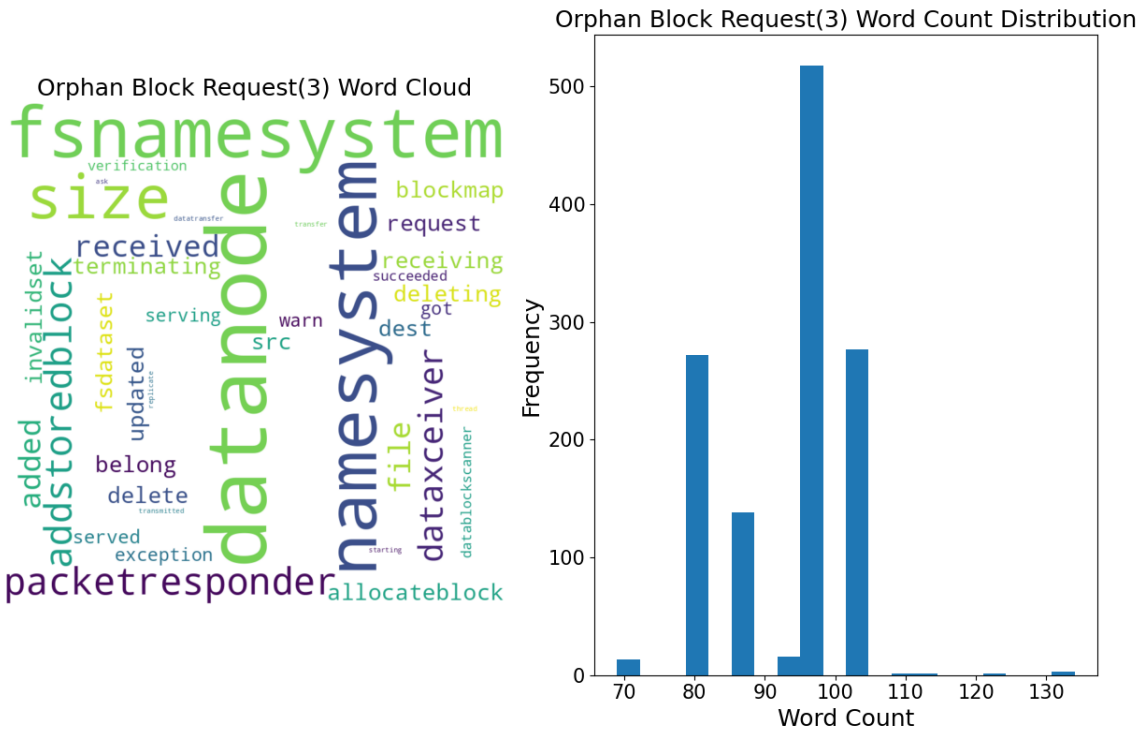

Figure 4-4. Orphan Block Request Word and Word Count Length Distributions

4.2.5 Redundant Block Request Defect Data Analysis Results

While the min-max range calculated (as the difference between the max (145) and min (56) log message word count) from statistics in Table 4-5 is 89 of this defect type

Table 4-5. Redundant Block Request Word Count Length Statistics

Descriptive Statistics	Value
Max Log Message Word Length	145
Min Log Message Word Length	56
Log Message Q1 Word Length	73
Log Message Q2 (Median) Word Length	73
Log Message Q3 Word Length	79
Log message Average Word Length	75
Log Messages Vocabulary Size	48
Count of Message Sequences	953

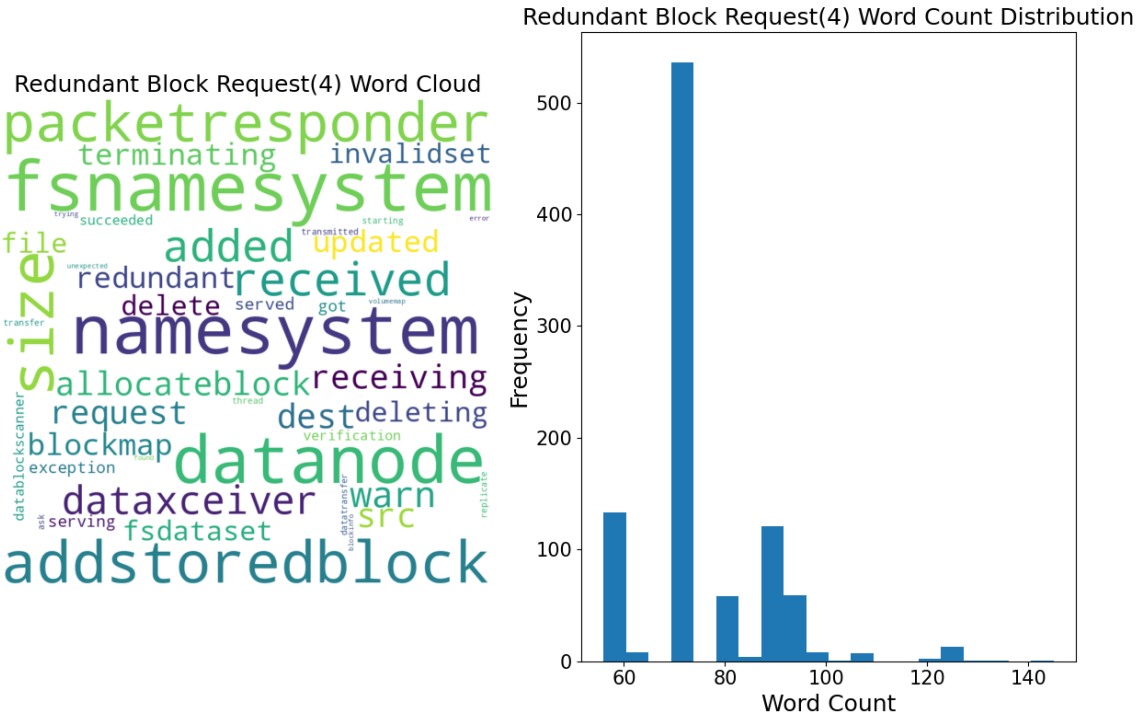

Figure 4-5. Redundant Block Request Word and Word Count Length Distributions

indicates that there is wide spread in the log message content, its IQR of just six indicates otherwise. More evidence on the probable correctness of the IQR is emphasized by the similarity of word count quartile values and the mean word count value. Also, the word count distribution on the right of Figure 4-5 is multi-modal, while its token frequency on the word cloud on the right displays the 'redundant' token.

4.2.6 Unexpected Early Halt Defect Data Analysis Results

The unexpected early halt defect, whose statistics and visuals are shown in Table 4-6 and Figure 4-6 respectively, is a bug whose signature is typified by the tendency to stop an executing operation early and unexpectedly. Its statistics from Table 4-6 indicate that its message content might all be the same or similar given its min-max and IQR ranges are zero and all measured descriptive statistics values are the same. Such characteristics might make automatic or manual diagnosing of this defect type trivial.

Table 4-6. Unexpected Early Halt Word Count Length Statistics

Descriptive Statistics	Value
Max Log Message Word Length	14
Min Log Message Word Length	14
Log Message Q1 Word Length	14
Log Message Q2 (Median) Word Length	14
Log Message Q3 Word Length	14
Log Message Average Word Length	14
Log Messages Vocabulary Size	14
Count of Message Sequences	2950

This observation is further substantiated by the single bar histogram and the similarity of the token's font size in the word cloud depicting unexpected early halt fault vocabulary tokens in Figure 4-6.

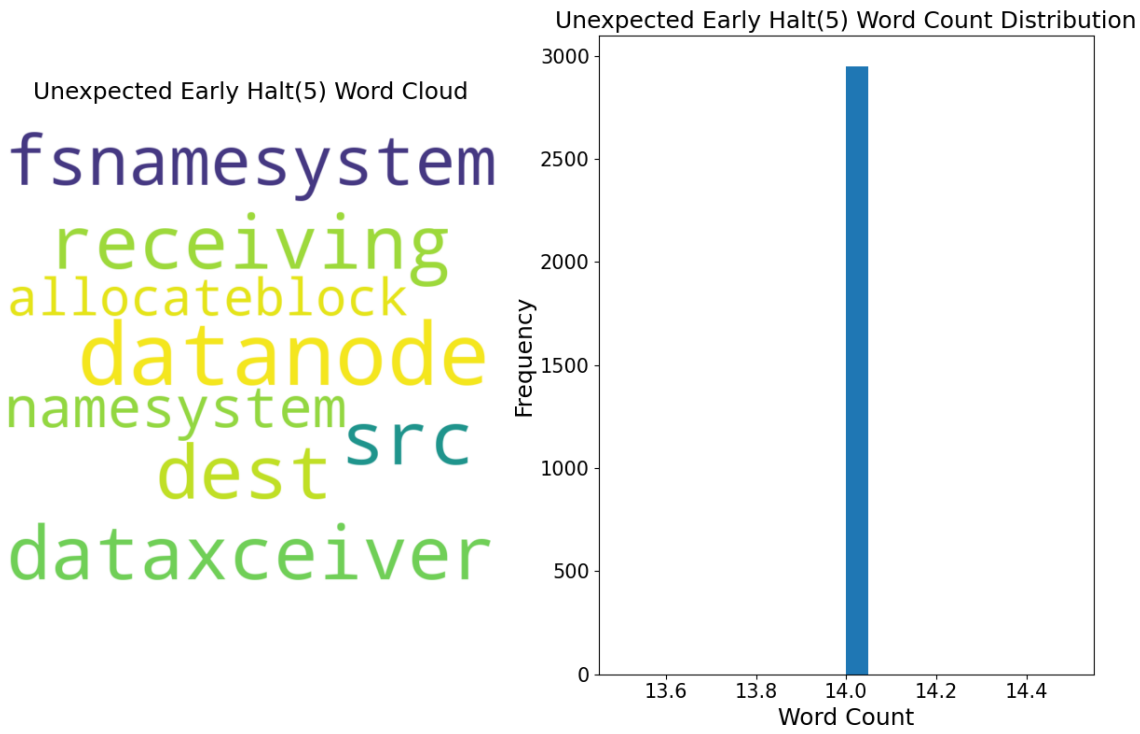

Figure 4-6. Unexpected Early Halt Word and Word Count Length Distributions

4.2.7 Fault Localization Log Data Analysis Result

Table 4-7 and Figure 4-7 are respectively the descriptive statistics and histogram plot for both the log data and log data summary descriptive statistics. Across all the measured statistics, the word count values for log data are at least 1.5 times that of their summary. On the histogram in Figure 4-7, the summary word count distribution has more modalities than its source log data. Meanwhile, Figure 4-8 demonstrates that the summary tokens are a subset of the actual runtime log tokens given they were extracted

from the runtime logs. These summary tokens include the fault-triggering method names, which are the fault locations pointed to by the localizers generated summary.

Table 4-7. Log Data and Summary Data Vocabulary Statistics

Descriptive Statistics	Value
Max Log Message Word Length	365
Max Log Message Summary Word Length	26
Min Log Message Word Length	14
Min Log Message Summary Word Length	10
Log Message Q1 Word Length	29
Log Message Summary Q1 Word Length	14
Log Message Q2 (Median) Word Length	74
Log Message Summary Q2 (Median) Word Length	17
Log Message Q3 Word Length	96
Log Message Summary Q3 Word Length	21
Log Messages Vocabulary Size	108
Log Message Summary Vocabulary Size	63

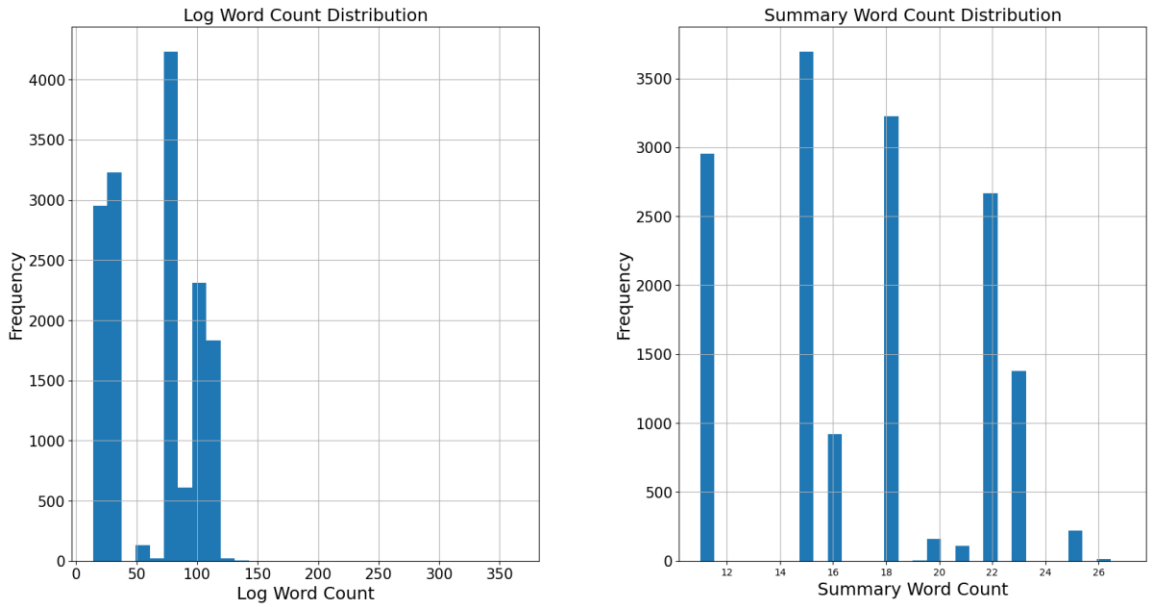

Figure 4-7. Log (Left) and Summary (Right) Word Count Length Distributions

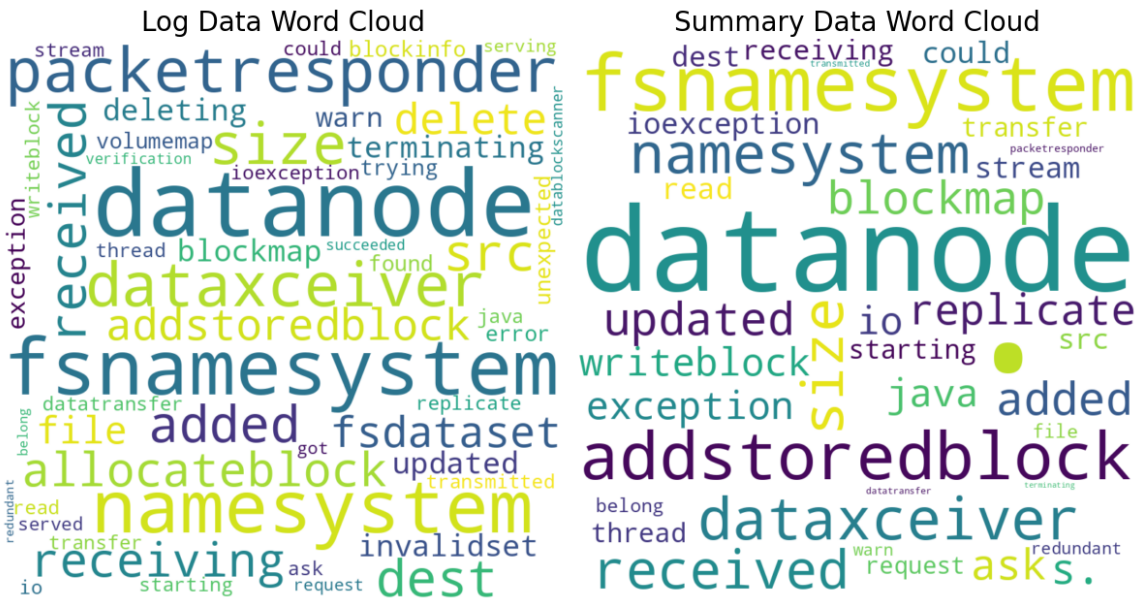

Figure 4-8. Defect Locator Runtime Log Data Tokens (Left) and Summary Data Tokens (Right)

4.2.9 Fault Localization Source Code Data Analysis Result

From section 3.2.4, it was stated that of the 2264 Java methods extracted from 182 HDFS Java source files only just (271) 12% of all these methods are responsible for generating the training runtime fault localizing log data. After this data was preprocessed it was learnt that only 10 (seen in Table 4-8) of these 271 methods include the fault triggering log points.

4.3 Fault Classifiers Performance Evaluation

Both the Transformer-based and the Bi-LSTM-based fault classification models were trained and validated with the preprocessed anomalous HDFS log data sequences transformed by BERT into 768-dimensional embedding vectors. There are two sets of the 12275 768d training log messages embeddings seen in Table 3-4: one preprocessed by

Table 4-8. Fault Log Points Fully Qualified Java Method Names

Fully Qualified Java Class Name	Method Name
org.apache.hadoop.hdfs.server.namenode.fsnamesystem	registerdatanode
org.apache.hadoop.hdfs.server.datanode.dataxceiver	writeblock
org.apache.hadoop.hdfs.server.datanode.datanode	updateblock
org.apache.hadoop.hdfs.server.datanode.fsdataset	Invalidate
org.apache.hadoop.hdfs.server.namenode.fsnamesystem	Addstoredblock
org.apache.hadoop.hdfs.server.datanode.datanode	Recoverblock
org.apache.hadoop.hdfs.server.datanode.datablockscanner	Verifyblock
org.apache.hadoop.hdfs.server.datanode.blockreceiver	Receivepacket
org.apache.hadoop.hdfs.server.datanode.blockreceiver.packetresponder	Run
org.apache.hadoop.hdfs.server.datanode.dataxceiver	Readblock

the custom parser and the other by the drain parser. The models were trained on these two sets to ascertain the effect of different preprocessing methodology on their prediction accuracy.

To train the models, a brute force hyperparameters tuning approach was implemented to determine the optimal number of epochs and train-validation-test splits. This involves arbitrarily setting the number of epochs to 30 with early stopping turned on for an 80-20 and 90-10 data splits. The observation from this exercise was that the best performance for both models' types happened within 20 epochs for an 80-20 data split. Additionally the models were fed mini batches of size 64 during all of their life cycle phases: training, validation, and evaluation.

An enumeration of the network (Keras) layers with their associated output shapes and parameters count is listed in Tables 4-9 and 4-10. From these tables it can be determined that while the Bi-LSTM classifier has 920,070 trainable parameters, the transformer classifier has 34 times (31,516,390) as more trainable parameters. Clearly, such a large number of parameters meant the transformer model took far longer to train relative to the Bi-LSTM model. Another observation from the tables is the similarity of the tensor shape of the input and output layers for both models. 'None' in the tensor dimension means it is calculated dynamically from the batch size and total number of training examples.

Table 4-9. Bi-LSTM Layers

Layer Type	Output Shape	Parameters Count
Input	(None, 255, 768)	0
Bidirectional	(None, 256)	918,528
Dropout	(None, 256)	0
Dense (output)	(None, 6)	1542

4.3.1 Defect Classifiers Evaluation Results

The shape of the tensor output of the classifiers dense output layer hints at the type of classification task and number of defect types. Given that each block's anomalous log message sequence belongs to one of six defects, the model is performing a multi-class classification task. Table 4-11 list the defect types and their associated label ID.

The model's metrics scores in Table 4-12 are from the classifiers that were trained with the output data from the custom parser. The results illustrate high prediction

accuracy from both models but with the Bi-LSTM unexpectedly demonstrating a better performance than the Transformer.

Table 4-10. Transformer Layers

Layer Type	Output Shape	Parameters Count
Input	(None, 225, 768)	0
Position Embedding	(None, 225, 768)	0
Transformer Block	(None, 225, 768)	31,491,584
Global Average Pooling 1D	(None, 768)	0
Dropout 1	(None, 768)	0
Dense 1	(None, 32)	24,608
Dropout 2	(None, 32)	0
Dense 2 (output)	(None, 6)	198

Table 4-11. Defect Types Label Name and ID

Label Name	Label ID
Invalid Set Deletion	0
IO Failure	1
Missing Block Info	2
Orphan Block Request	3
Redundant Block Request	4
Unexpected Early Halt	5

Using as training and test data, the output of the Drain parser results in classifiers that are equally performant at predicting the fault class of an anomalous log message

sequence as Table 4-13 depicts.

4.4 Fault Localizers Performance Evaluation

Just as for the classifiers, the optimal number of training epochs for the localization LSTM was discovered by leveraging early stopping to be within the originally specified 30 epochs. Following a brute force hyperparameter tuning strategy, the LSTM encoder was determined to be 6 layers deep while the decoder was just 1 layer deep. Also, each layer's number of neurons and dropout were set at 128 and 0.4, respectively. The final LSTM encoder-decoder summarization model had a total 3,391,701 parameters with 3,374,401 trainable and the remaining 17, 300 not trainable. The non-trainable parameters are from the GloVe pretrained word embeddings.

The Google T5 base model finetuned on the log data has 12 layers with a total 247,577,856 trainable parameters: the encoder with 109,628,544 parameters and the decoder with 137,949,312 parameters. Fine-tuning was done for 10 epochs with a batch size of 64.

4.4.1 Defect Localization Models Evaluation Results

The bug localizer performance is evaluated using the accuracy@k metric. This metrics scores the model on its ability to generate a log sequence summary query that can be successfully used to rank the top k source code method(s) from which the fault potentially originated. Therefore, the goodness of a log sequence summary is determined by whether the bug being diagnosed originated directly or indirectly from the method that contains the log point from which the summary was emitted. Figure 4-9 shows a side-by-

side comparison of the LSTM and Transformer models for k = 1,3,5 retrievals. The Transformer log summary queries clearly result in retrievals containing the correct fault methods more times than the LSTM summary queries.

Table 4-12. Bi-LSTM and Transformer (on custom parser outputs) Classification Report

	Label	Precision	Recall	F1	Accuracy	Support Examples
Bi-LSTM	0	0.99	0.97	0.98	0.99	552
	1	1.00	1.00	1.00		636
	2	0.98	1.00	0.99		812
	3	1.00	0.99	0.99		244
	4	0.98	1.00	0.99		186
	5	1.00	1.00	1.00		578
Transformer	0	0.99	0.97	0.98	0.96	554
	1	1.00	1.00	1.00		631
	2	0.97	1.00	0.98		819
	3	0.87	0.77	0.82		243
	4	0.74	0.80	0.76		186
	5	1.00	1.00	1.00		575

Table 4-13. Bi-LSTM and Transformer (on drain parser outputs) Classification Report

	Label	Precision	Recall	F1	Accuracy	Support Examples
Bi-LSTM	0	0.62	0.67	0.65	0.76	547
	1	1.00	1.00	1.00		632
	2	0.60	0.89	0.72		817
	3	0.00	0.00	0.00		248
	4	0.00	0.00	0.00		190
	5	1.00	1.00	1.00		574
Transformer	0	0.61	0.71	0.66	0.76	555
	1	1.00	1.00	1.00		630
	2	0.60	0.85	0.70		812
	3	0.00	0.00	0.00		242
	4	0.00	0.00	0.00		186
	5	1.00	1.00	1.00		583

4.5 Findings from Evaluation Results

In this section the objective is to present some general findings with respect to the aforementioned results. These findings explore data trends effects on the performance of the bug classifiers and localizers and the relationship between the data processing strategies and prediction accuracy of the models.

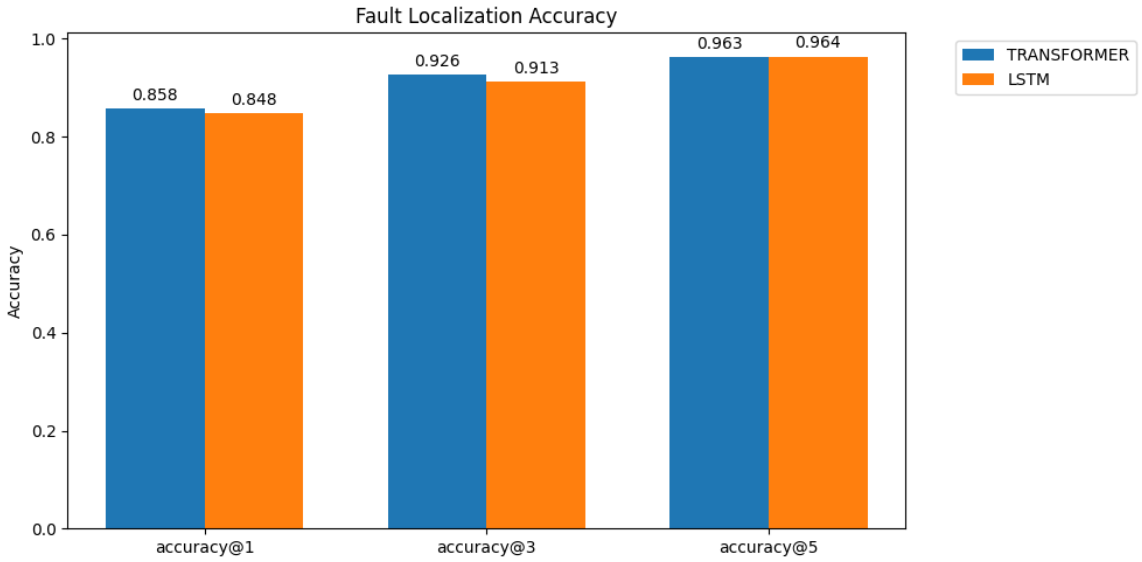

Figure 4-9. Defect Localizers Accuracy@k Report

4.5.1 Data Trends Effects on Model Performance

In the presented statistical results, word cloud and histogram visuals of the log data, there are several patterns that can be useful for fingerprinting certain fault types and identifying the defect location.

Firstly, with discriminating features on inputs, accuracy on predicting the defect type improves significantly for the classifier. A property of the data relevant for learning discriminating features is the number of the defect training examples. This is the case for defect labels 0,1,2, and 5 whose greater number of log message examples gave them the edge at establishing a defect-pattern correlation. The unreasonable advantage of data in model performance first reported by Halevy, A. et al. (2009) and has been corroborated in numerous empirical studies helps in elucidating the cause of the differing performance of the bug classifiers for each bug type. An observation from the evaluation result (for

this particular scenario) is that in the task of defect classification through log data, the Transformer prediction accuracy on a defect type will trail that of the Bi-LSTM if there is enough significant imbalance in the defects' training data relative to another defect type with sufficient training examples.

Secondly, results in Table 4-13 are markedly different from results in Table 4-12. While Table 4-13 shows the Bi-LSTM and Transformer classifiers with equal average performance, Table 4-12 show almost perfect prediction accuracy for the models. The one difference is that models in Table 4-13 trained on just the log message text, while the classifiers in Table 4-12 had additional log header info in their training data. Therefore, it would seem that this extra info was instrumental in providing more contextually relevant defect-delineating information.

Thirdly, the classifiers and localizers gained from the presence respectively of defect-specific tokens and defect location-revealing tokens. For the classifiers, tokens unique to a particular defect, such as "IO Exception" for the IO failure bug resulted in almost perfect prediction accuracy for bugs belonging to that defect class. The bug localizers on the other almost always located bugs whose symptom log messages summary included the generating method name tokens. For example "writeblock" method present in the summary of some block log message sequence is the location for the log point template for the message summary.

A fourth data trend discovery is that the defect localizer's highest frequency tokens tend to be the same as those of the defect classifiers in the runtime log data. On the other hand, discriminating tokens specific to a particular defect class such as

ioexception, invalidset, addstoredblock, etc. do not tend to be highlighted as very important in the context of bug localization.

Lastly, unsurprisingly the prominence of tokens is consistent with both the original runtime logs and their summary. For both the defect classifiers and localizers, the most common words are the names of packages and classes. This is followed by method names, then the actual log message text tokens.

4.5.2 Data Preprocessing Strategies Effects on Model Performance

Generation of embedding vector for a block's log message sequence involves several processing stages: first the embedding vector for each sequence log message is computed and the resulting vectors are combined to create the embedding vector for the blocks' log message sequence.

Creation of each sequence log message word embeddings involves standardization, followed by tokenization. Then each token is vectorized and final representation for the log message is computed by averaging all message token embedding vector.

The combination step that creates the final embedding vector capturing all (or at least an estimate of) the pertinent features relevant for gleaning insightful patterns by the classifier can be done either by adding or averaging each message vector. A crucial observation important for the classifier performance was that additive combination resulted in a more performant model as opposed to combination via averaging.

In the defect localization tasks, summarization was employed to disambiguate the defect-revealing message in the log message sequence. In practice summarization can be

performed by either extractive or abstractive means. While extractive summarization generates a new sentence by extracting sentence(s) from the original text, abstractive summarization generates a new sentence by paraphrasing the original text. For method-level localization it turns out that extractive summarization followed by search results in higher accuracy on defect localization than log message summary prediction using abstractive summarization succeeded by retrieval.

Unsurprising, for log message data summary, an LSTM trained on the data performs better on generating log messages summary than the Google T5 base model. But if base-T5 is finetuned on the training log data, its summary generative capability becomes mostly better relative to the LSTMs. Additionally, while the LSTM tends to memorize its training summary and subsequently regurgitate it as its prediction, the transformer tends to predict as its summary a selection of the most salient tokens from the defect-revealing summary it was trained with.

Using neural networks for defect troubleshooting can be an effective strategy to fast track and accurately address defect diagnostics. In particular, deep neural nets are especially suited for defect categorization, as they are able to learn the complex and non-linear relationship between the defect types based on the defect's symptom log messages.

Applying dropout to the defect diagnostics models during training produced a consistently more performant model relative to training the models without using dropout. However, there did not seem to be any notable difference in model performance for different dropout value applications.

4.6 Results Summary

The results presented in the preceding tables and figures were generated from evaluating the trained models on the set-aside test data. Evaluations were performed on models trained on the finetuned hyperparameters. Also, the reported results are the median of 25 rounds of evaluations of both the defect classifiers and defect localizers.

While the evaluation of the bug classifiers was based on standard classification metrics: precision, recall, F1, and accuracy, the bug localizers performance measurement metrics is accuracy@k. For both DL models the results demonstrate at least a 90^{th} percentile performance on the task of defect classification and localization, especially when the training data contains contextually relevant information.

Chapter 5—Discussion and Conclusions

5.1 Discussion

After software is formally released, it enters the maintenance phase of the Software Development Life Cycle (SDLC). During this stage, updates are normally made either due to bugs reported by end users or system upgrades/updates and the addition of new features. To address software bugs, whether pre- or post-release, an analysis of the bugs' symptoms logged to a file is indispensable in deciphering the type of fault and where in the software code the bug originated. The log analysis process performed while troubleshooting a software bug can be achieved through manual or automatic means.

With the trend towards complexity that software exhibits, it has become exorbitantly costly or even infeasible to manually analyze the resulting humongous log files of software bug stack traces for defect diagnosis. Meanwhile, the alternative of automatic log data analysis has been successfully applied in determining the class and location of software bugs, particularly with the use of deep learning networks.

This study is an empirical investigation of DL-based log analysis for the purpose of tackling two main defect diagnosis tasks: defect classification and defect localization. For both of these tasks, a Bi-LSTM classification network, an LSTM localization net and a Transformer network learnt applicable patterns from the HDFS log data. Subsequently, the trained models were evaluated using task-specific metrics to score and compare their performance on the tasks. While the defect classification task employed the standard metrics of accuracy, precision, recall, and F1-score as its evaluation metrics, accuracy@k was the measure of performance for the defect localization task. Afterwards, the

calculated metrics results are used to facilitate the goal of providing answers to the three posed research questions.

RQ1: Between Bi-LSTMs and Transformer-based models, which will be more accurate at software system defect categorization?

H1: A transformer-based model will be more accurate than a Bi-LSTM model for software system defect categorization.

Figure 5-1 is a chart of the size of log data sequences versus fault type for the total 3,071 and 11,047 log sequences training and test data, respectively. To ensure that each fault type was proportionally represented, a stratified data split was performed on the anomalous log data sequences for training and test data. The similarity of the shape of the data charts in Figure 5-1 clearly verifies this proportionality claim.

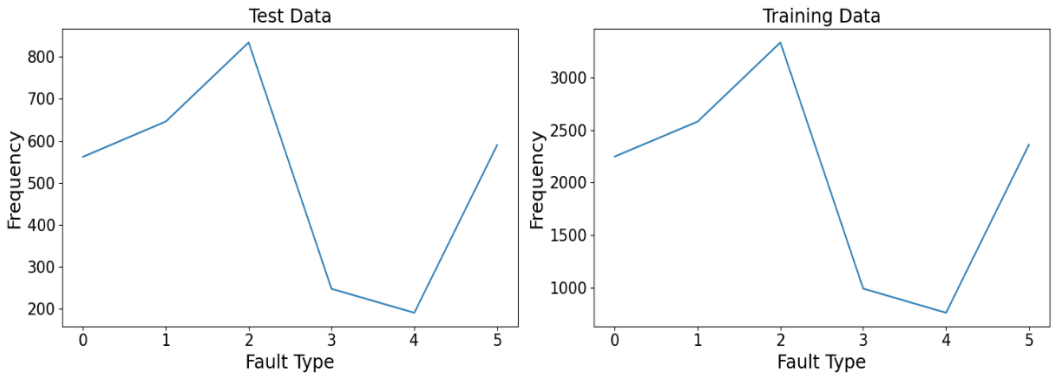

Figure 5-1. Classification Training and Evaluation Data Size by Fault Type

From Table 4-12, it can be deduced that the Bi-LSTM and Transformer models have almost equal perfect scores of at least 0.97 on precision, recall, and F1 for fault types 0, 1, 2, and 5. Also, Figure 5-1 demonstrates that this group of fault types has relatively more data to learn from. Meanwhile, bugs of types 3 and 4, whose training data size is about one-third the size of the fault type in the first (highly performant) group with

the smallest training data, are not readily classified. Although the Bi-LSTM scores on bugs 3 and 4 still match those in the first group, the Transformer scores, which are nonetheless above average, are comparatively inferior to those of the Bi-LSTM. Such a susceptibility to imbalance in the log data displayed by the Transformer might have caused its accuracy score of 0.96 to trail that of the Bi-LSTM accuracy of 0.99. Ultimately, resulting in a contradiction of the stated hypothesis.

RQ2: Does pre-processed log data using the Drain parser improve the diagnostic prediction accuracy of transformer-based and Bi-LSTM models as compared to raw log data?

H2: Diagnostic prediction accuracy will be better for a Bi-LSTM and Transformer-based model trained on raw log data relative to pre-processed data using the Drain parser.

Table 5-1. Classification Training and Evaluation Data Size by Fault Type

Fault Type	Training Data Size	Evaluation Data Size
0	2248	562
1	2580	646
2	3333	834
3	992	248
4	762	191
5	2360	590

As can be recalled and depicted in Figure 3-2, the output of the custom log message parser includes additional information from the header of the log message. This

additional information proffers a discriminative advantage in the faults' classification for both classifiers. In fact, this advantage gives the classifiers trained on the custom parser data up to a 20% lead in classification accuracy relative to classifiers trained on the output of the drain parser. This huge disparity in performance accuracy between the custom parser output trained classifiers and the drain parser output trained models is also observed in the results measurement metrics: precision, recall, and F1. Doing a bug-by-bug analysis of the custom results in Table 4-12 and the drain results in Table 4-13, the conclusion is that while both classifiers' scores on faults 1 and 5 are equal for both parsers' output, they differ considerably for the other bug types. Thus, these results confirm the hypothesis prediction for this research question.

Focusing on the results in Table 4-13, which correlated with fault types and data sizes in Table 5-1, it is evident that both models' results on the performance measurement metrics for the fault types tend to be commensurate with the data size. This behavior is an extrapolation from the patterns learnt in the training data. Hence, the classifiers' per-fault scores on precision, accuracy, and F1 form three clusters that might be attributable to training data sizes and content. There are the perfectly scored bug types: 1 and 5, the average to above average score cluster: 0 and 2, and the zero score group: 3 and 4.

RQ3: Is the log message summarization query predicted by an LSTM model more effective at defect localization than that generated by a Transformer-based model?

H3: The log message summarization query predicted by a Transformer model will aid in a more effective defect localization than the log message summarization query predicted by an LSTM model.

As discussed in earlier sections, the summary of a fault's log message sequence represents the log message in the sequence produced by the method that potentially initiated the fault. Thus, such a summary generated by the models localizes the fault. Checking the accuracy of a generated log summary is done in two steps. First, the summary is used to fetch a ranked list of methods from a pre-constructed knowledge DB. Then, accuracy is computed as the percentage of the ranked retrievals with at least one (responsible) faulty method across all the evaluation faults. This accuracy metric is referred to as the accuracy@k, where k is the size of the ranked list.

Figure 4-9 illustrates the results of retrieving a ranked list of sizes 1, 3, and 5. Clearly, the Transformer summarizer outscores the LSTM summarizer on accuracy@1 and accuracy@3, while having an equal score for accuracy@5. Better performance on smaller sized retrievals is preferable as developers would only need to examine fewer methods to locate and fix faults. This better performance by the transformer supports the hypothesis claim.

5.2 Conclusions

It is inevitable that software complexity contributes significantly to software faults. Such complexity triggers faults, resulting in the expenditure of enormous corrective efforts. With the objective of alleviating costly efforts on software fault fixes, the aforementioned three research questions are an attempt to drive the investigation of a solution approach that best tackles defect classification and localization.

Firstly, there was the question of which of the two autoregressive models could classify the provided log data better based on fault type. While both models performed

superbly, the Bi-LSTM surprisingly outperformed the Transformer, contrary to the supposed research hypothesis.

The second question investigated whether a diagnostics model trained on the output of the Drain parser was more performant than one trained on the output of the custom parser. Remarkably, training a model with the output of the custom parser endowed it with superior defect diagnostics capabilities.

Lastly, the search enhanced summarizers based on Transformer architecture outperformed the one based on LSTM on the task of defect localization. The Transformer superior performance was particularly noteworthy because it was achieved on fewer retrievals from its predicted summary.

5.3 Contributions to Body of Knowledge

The dearth of attention paid to understanding the relationship between the log training data processing method and defect type prediction accuracy by a DL classification model opened a research gap that this empirical study attempted to fill. To remediate this shortcoming, a Bi-LSTM model and a Transformer model are trained and evaluated on the data preprocessed with two different parsers: an open-source Drain parser and a custom Python parser. The results from undertaking this empirical investigation establish that training data constructed from the combination of the log message body and extra header fields, such as logging level and log emitting component, results in a tremendous improvement to prediction accuracy by classification DL models.

This research approaches the task of bug localization differently from how it was previously tackled. The first insightful contribution was to leverage the defect-revealing

log message in a bug's stack trace for bug localization. Having selected a log message denoted as the summary of the fault's log message sequence, a second contribution offered a solution approach that defined the bug localization task as a summarization task. Thirdly, building on the summarization task, to locate bugs in source code, a technique known as Search Enhanced Summarization (SES) is presented. SES locates bug potential originating method(s) by using the model's predicted log message summary as a query into a Retrieval Augmented Generation (RAG) vector database. Lastly, the RAG vector database utilized for diagnostics knowledge storage is aimed at providing a characterization of the relationship between the extracted summary, the original log message template, and the summary-generating method.

Also, there are contributions with respect to HDFS log data defect types streamlining and source code mapping for the HDFS log data. For the classifiers, the anomalous HDFS log messages defect types were reduced to just six from the originally defined 21 defect types, while encompassing most of the messages. Finally, to facilitate defect localization, the popular HDFS log data used for various log analysis studies was paired with it generating HDFS source code through manual inspection and rigorous analysis. This log data-source code pair will certainly be invaluable for defect localization research.

5.4 Recommendations for Future Research

One observation in this empirical investigation was a 20 percent difference in classification accuracy between a bug classifier trained on custom parser output and one trained on drain parser output for the same log data. Such high disparity in favor of the

custom parser demands further investigation. Understanding the relevant header fields to combine with the log message text for training fault classifiers with high predictive accuracy will guide the design and development of better log data parsers.

Knowledge from developers many years in the trenches of software defect troubleshooting has provided some insights on defect localization. One such insights is that in large and complex software systems some failures that ultimately manifested from errors might have been as a result of multiple root-cause faults. The SES technique developed in this research assumes one-to-one relationship between errors and faults. Thus, it will not be performant in the cases of one-to-many or many-to-many error-fault relationships. Further research would need to be undertaking to adapt SES to these cases. Using abstractive summarizers as opposed to extractive summarizers is a possible research path. So, will leveraging fault location-specific extractive summarization i.e. extract multiple faults revealing log messages as a bug stack trace summary.

Lastly, this research was executed on a single software system log data. Therefore, another obvious and very important research direction is to train and evaluate the developed defect classification and localization models on log data from other software systems.

References

Amusuo, P. C., Sharma, A., Rao, S. R., Vincent, A., & Davis, J. C. (2022). Reflections on Software Failure Analysis. In Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering

Bader, J., Scott, A., Pradel, M., & Chandra, S. (2019). Getafix: learning to fix bugs automatically. ACM Programming Language, 3(OOPSLA)

Briand, L. C., Labiche, Y. & Liu, X. (2007). Using Machine Learning to Support Debugging with Tarantula. *IEEE* 18th International Symposium on Software Reliability (ISSRE '07).

Chen, A. R. (2019). An empirical study on leveraging logs for debugging production failures. *IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion),* pp. 126-128

Chen, Z., Kommrusch, S., Tufano, M., Pouchet, L.-N., Poshyvanyk, D., & Monperrus, M. (2021). Sequencer: Sequence-to-sequence learning for end-to-end program repair. IEEE Transactions on Software Engineering

Dobrowolski, W., Nikodem, M., Zawistowski, M., & Unold, O. (2022). Improved Software Reliability Through Failure Diagnosis Based on Clues from Test and Production Logs. In Proceedings of the International Conference on Dependability and Complex Systems, Springer pp. 42–49

Du, M., Li, F., Zheng, G., & Srikumar, V. (2017). Deeplog: Anomaly detection and diagnosis from system logs through deep learning. *ACM* SIGSAC conference on computer and communications security (pp. 1285-1298)

Du, X., Zheng, Z., Xiao, G., Zhou, Z. & Trivedi, K. S. (2022). DeepSIM: Deep Semantic Information-Based Automatic Mandelbug Classification. in IEEE Transactions on Reliability, vol. 71, no. 4, pp. 1540-1554

Gao, J., Zhang, L., Zhao, F. & Zhai, Y. (2019). Research on Software Defect Classification. *IEEE* 3rd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)

Ghanbari, A. & Marcus, A. (2022). Patch correctness assessment in automated program repair based on the impact of patches on production and test code. 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 654–665

Grottke, M. & Trivedi, K. S. (2005). A Classification of Software Faults. 16th International IEEE Symposium on Software Reliability Engineering, pp. 4.19–4.20

Gyimothy, T., Beszedes, A., & I Forgacs. (1999). An efficient relevant slicing method for debugging. *ACM Sigsoft Symposium* European Software Engineering Conference.

Halevy, A., Norvig, P. & Pereira, F. (2009). The Unreasonable Effectiveness of Data. In IEEE Intelligent Systems, vol. 24, no. 2, pp. 8-12

He P., Zhu, J., Zheng, Z. & Lyu, M. R., (2017). Drain: An Online Log Parsing Approach with Fixed Depth Tree. IEEE International Conference on Web Services (ICWS)

Huo, X., Li, M., & Zhou, Z.-H. (2016). Learning unified features from natural and programming languages for locating buggy source code. Proceeding of 25th International Joint Conference on Artificial Intelligent (pp. 1606–1612)

Institute of Electrical and Electronics Engineers. (2009). IEEE standard classification for software anomalies (Revision of IEEE Std 1044-1993) , vol., no., pp.1-23, , doi: 10.1109/IEEESTD.2010.5399061

Jones, J. A., Harrold, M. J., & Stasko, J. T. (2001). Visualization for fault localization. In Proceedings of ICSE Workshop on Software Visualization.

Krasner, H. (2022). The Cost of Poor Quality Software in the U.S. : A 2022 Report. Retrieved from https://www.it-cisq.org/the-cost-of-poor-quality-software-in-the-us-a-2022-report/

Krasner, H. (2021). The Cost of Poor Quality Software in the U.S. : A 2022 Report. Retrieved from https://www.it-cisq.org/the-cost-of-poor-software-quality-in-the-us-a-2020-report/

Landauer, M., Onder, S., Skopik, F., & Wurzenberger, M. (2023). Deep learning for anomaly detection in log data: A survey. *Machine Learning with Applications*, 12, 100470.

Laprie, J. -C. (1995). Dependable Computing and Fault Tolerance : Concepts and Terminology. Twenty-Fifth International Symposium on Fault-Tolerant Computing

Le, V. H., & Zhang, H. (2021). Log-based anomaly detection without log parsing. *IEEE/ACM* 36th International Conference on Automated Software Engineering (ASE) (pp. 492-504)

Esteves, G., Figueiredo, E., Veloso, A., Viggiato, M., & Ziviani, N. (2020). Understanding machine learning software defect predictions. Automated Software Engineering, 27(3), 369-392.

Minelli, R., Mocci, A., & Lanza, M. (2015). I Know What You Did Last Summer-An Investigation of How Developers Spend Their Time. In 2015 IEEE 23rd International Conference on Program Comprehension

Molina, F., Copia, J. M., & Gorla, A. (2024). Improving Patch Correctness Analysis via Random Testing and Large Language Models. IEEE Conference on Software Testing, Verification and Validation (ICST), pp. 317-328

Moon, S., Kim, Y., Kim, M., & Yoo. S. (2014). Ask the mutants: Mutating faulty programs for fault localization. *IEEE* Seventh International Conference on Software Testing, Verification and Validation,

Pandey, S.K., Mishra, R.B., & Tripathi A.K. (2020). Machine learning based methods for software fault prediction: A survey. Expert Systems Application (vol 172, article 114595)

Papadakis, M. & Traon, Y. L. (2012). Using mutants to locate unknown faults, *IEEE*

Papadakis, M. & Traon, Y. L. (2015). Metallaxis-FL: mutation-based fault localization. *IEEE* Software Testing, Verification and Validation (ICST)

Ren, R., Cheng, J., Yin, Y., Zhan, J., Wang, L., Li, J., & Luo, C. (2018). Deep convolutional neural networks for log event classification on distributed cluster systems. *IEEE* International Conference on Big Data (Big Data) (pp. 1639-1646).

Renieres, M., & Reiss, S. P. (2003). Fault localization with nearest neighbor queries. In 18th IEEE International Conference on Automated Software Engineering, (pp. 30-39).

Seliya, N., Khoshgoftaar, T. M., & Van Hulse, J. (2010). Predicting faults in high assurance software. In 2010 IEEE 12th international symposium on high assurance systems engineering (pp. 26-34).

Software Testing, Verification and Validation (ICST)

Tabassum, N., Namoun, A., Alyas, T., Tufail, A., Taqi, M., & Kim, K. -H. (2023). Classification of Bugs in Cloud Computing Applications Using Machine Learning Techniques. *Applied Sciences*, 13(5), 2880. Retrieved from https://doi.org/10.3390/app13052880

Thung, F., Lo, D. & Jiang, L. (2012). Automatic Defect Categorization. *IEEE* 19th Working Conference on Reverse Engineering

Tree-sitter parser generator. (2025) Wikipedia. https://en.wikipedia.org/wiki/Tree-sitter_(parser_generator)

Milvus-10. (2025) PyMilvus: Python SDK for Milvus Vector Database [Computer software]. Retrieved from https://milvus.io/docs/quickstart.md, https://huggingface.co/sentence-transformers/paraphrase-albert-small-v2

Reimers, N., & Gurevych, I. (2019). Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP), 3982-3992

Pymilvus. (2025) Pymilvus/Milvus Lite. https://milvus.io/docs/quickstart.md

Vaswani, A., Shazeer N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I., (2017) Attention is all you need. Advances in Neural

Information Processing Systems. Long Beach, CA, USA: 30th Annual Conference, 5998–6008

Wong, W. E., & Qi, Y. (2009). BP neural network-based effective fault localization. International Journal of Software Engineering and Knowledge Engineering, 19(04), 573-597

Wong, W. E., Gao, R., Li, Y., Abreu, R. & Wotawa, F. (2016). A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707-740.

Wong, W. E., Zhao, L., Qi, Y., Cai, K. Y., & Dong, J. (2007). Effective Fault Localization using BP Neural Networks. In SEKE (pp. 374-379)

Xia, B., Bai, Y., Yin, J., Li, Y., & Xu, J. (2020). LogGAN: A log-level generative adversarial network for anomaly detection using permutation event modeling. *Information Systems Frontiers*, 23, 285–298

Xie, Y., Yang, K., & Luo, P. (2021). LogM: Log Analysis for Multiple Components of Hadoop Platform. in IEEE Access, vol. 9, pp. 73522-73532,

Xiong, Y., Liu, X., Zeng, M., Zhang, L., & Huang, G. (2018). Identifying patch correctness in test-based program repair. ACM 40th International Conference on Software Engineering, ICSE 2018, pp. 789–799

Yuan, D., Mai, H., Xiong, W., Tan, L., Zhou, Y., & Pasupathy, S. (2010). SherLog: error diagnosis by connecting clues from run-time logs. *ACM SIGARCH Computer Architecture News*, Volume 38, Issue 1, pp. 143 – 154

Yuan, D., Park, S., Huang, P., Liu, Y., Lee, M. M., Tang, X., Zhou, Y., & Savage, S. (2012). Be conservative: enhancing failure diagnosis with proactive logging. 10th

USENIX conference on Operating Systems Design and Implementation, pp. 293 – 3

Yuan, D., Zheng, J., Park, S., Zhou, Y., & Savage, S. (2011). Improving software diagnosability via log enhancement. 16th international conference on Architectural support for programming languages and operating systems, pp. 3 – 14

Glass, R. L. (2001). Frequently forgotten fundamental facts about software engineering. IEEE software, 18(3), 112-111

Yuan, W., Lu, S., Sun, H., & Liu, X. (2020). How are distributed bugs diagnosed and fixed through system logs? Information and Software Technology vol. 119, pp. 106234

Zhang, X., Xu, Y., Lin, Q., Qiao, B., Zhang, H., Dang, Y., & Zhang, D. (2019). Robust log-based anomaly detection on unstable log data. *ACM* 27th joint meeting on European software engineering conference and symposium on the foundations of software engineering (pp. 807-817)

Zhu, J., He, P., Fu, Q., Zhang, H., Lyu, M. R., & Zhang, D. (2015). Learning to Log: Helping Developers Make Informed Logging Decisions. *IEEE/ACM 37th IEEE International Conference on Software Engineering*, pp. 415-425, doi: 10.1109/ICSE.2015.60

Zhu, J., He, S., He, P., Liu, J., & Lyu, M. R. (2023). Loghub: A Large Collection of System Log Datasets for AI-driven Log Analytics. IEEE International Symposium on Software Reliability Engineering (ISSRE).

Appendix A

$$\frac{TP}{TP + FP}$$

Equation A-1, Precision

$$\frac{TP}{TP + FN}$$

Equation A-2, Recall

$$\frac{TP + FP}{TP + FP + TN + FN}$$

Equation A-3, Accuracy

$$\frac{2 * precision * recall}{precision + recall}$$

Equation A-4, F1-Score