

Kamal Raj N

N_Kamal_Raj_Mtech_Final

 Mtech Thesis

Document Details

Submission ID

trn:oid:::27535:142403801

Submission Date

Jun 10, 2026, 11:25 AM GMT+5:30

Download Date

Jun 10, 2026, 11:27 AM GMT+5:30

File Name

N_Kamal_Raj_Mtech_Final.pdf

File Size

3.1 MB

40 Pages

11,858 Words

64,431 Characters

9% Overall Similarity

The combined total of all matches, including overlapping sources, for each database.



Filtered from the Report

- Bibliography
- Submitted works

Match Groups

-  **69 Not Cited or Quoted 6%**
Matches with neither in-text citation nor quotation marks
-  **27 Missing Quotations 2%**
Matches that are still very similar to source material
-  **0 Missing Citation 0%**
Matches that have quotation marks, but no in-text citation
-  **0 Cited and Quoted 0%**
Matches with in-text citation present, but no quotation marks

Top Sources

- 6%  Internet sources
- 6%  Publications
- 0%  Submitted works (Student Papers)

Integrity Flags

0 Integrity Flags for Review

Our system's algorithms look deeply at a document for any inconsistencies that would set it apart from a normal submission. If we notice something strange, we flag it for you to review.

A Flag is not necessarily an indicator of a problem. However, we'd recommend you focus your attention there for further review.

Match Groups

- 69 Not Cited or Quoted 6%**
Matches with neither in-text citation nor quotation marks
- 27 Missing Quotations 2%**
Matches that are still very similar to source material
- 0 Missing Citation 0%**
Matches that have quotation marks, but no in-text citation
- 0 Cited and Quoted 0%**
Matches with in-text citation present, but no quotation marks

Top Sources

- 6% Internet sources
- 6% Publications
- 0% Submitted works (Student Papers)

Top Sources

The sources with the highest number of matches within the submission. Overlapping sources will not be displayed.

1	Internet	www.mdpi.com	<1%
2	Internet	dtu.ac.in	<1%
3	Internet	dspace.daffodilvarsity.edu.bd:8080	<1%
4	Internet	mdpi-res.com	<1%
5	Publication	Jaskaran Singh, Meenu Gupta, Rakesh Kumar. "Smart Technologies and Intelligen...	<1%
6	Publication	K. V. Sambasivarao, Anasuya Sesha Roopa Devi Bhima. "Artificial Intelligence, Co...	<1%
7	Publication	"Intelligent Systems and Applications", Springer Science and Business Media LLC,...	<1%
8	Publication	Md. Reshad Al Muttaki, Sadia Afrin, Alvi Ibn Amzad Anil, Shameem Hasan. "State-...	<1%
9	Internet	core.ac.uk	<1%
10	Internet	ijirt.org	<1%

11	Internet	portal.research.lu.se	<1%
12	Publication	Park, Hyeonsuk. "Improving Multiclass Classification for Software Issue Triage: a ...	<1%
13	Internet	arxiv.org	<1%
14	Internet	repository.uhamka.ac.id	<1%
15	Publication	Neha Sharma, Prashant Karwasra. "Suicidal Text Detection on Social Media for Su...	<1%
16	Internet	acikbilim.yok.gov.tr	<1%
17	Internet	www.ijfmr.com	<1%
18	Internet	www.researchgate.net	<1%
19	Publication	Ruslan Bondaruc, Davide Cazzetta, Marco Anisetti. "Advanced IoT Edge Architect...	<1%
20	Internet	www.gavinpublishers.com	<1%
21	Publication	Chandrasekar Vuppalapati. "Building Next-Gen ESG Platforms with IoT and AI for ...	<1%
22	Publication	Deepshikha Chhabra, Dr Raman Chadha. "Machine Learning Approaches in Conte...	<1%
23	Publication	Pengfei Wu, Zetong Wang, Sichen Lei, Xiaofan Wang, Zhenkun Tan, Jiao Wang, Yi...	<1%
24	Internet	eprints.qut.edu.au	<1%

25	Internet	eprints.utar.edu.my	<1%
26	Internet	hyuna-ham.github.io	<1%
27	Publication	Pushpa Choudhary, Sambit Satpathy, Arvind Dagur, Dharendra Kumar Shukla. "Re...	<1%
28	Internet	escholarship.org	<1%
29	Internet	thesai.org	<1%
30	Internet	archive.org	<1%
31	Internet	netman.aiops.org	<1%
32	Publication	"Artificial Intelligence in Education", Springer Science and Business Media LLC, 20...	<1%
33	Publication	Christian Y. Sy, Lany L. Maceda, Mary Joy P. Canon, Nancy M. Flores. "Beyond BER...	<1%
34	Publication	Yutaro Kashiwa. "RAPTOR: Release-Aware and Prioritized Bug-Fixing Task Assign...	<1%
35	Internet	cdn.techscience.cn	<1%
36	Internet	etheses.lib.ntust.edu.tw	<1%
37	Internet	www.diva-portal.org	<1%
38	Publication	Anandan Chinnalagu, Ashok Kumar Durairaj. "Comparative Analysis of BERT-base...	<1%

39	Publication	Jinfu Chen, Patrick Kwaku Kudjo, Solomon Mensah, Selasie Aformaley Brown, Geo...	<1%
40	Publication	Rongcun Wang, Xingyu Ji, Yuan Tian, Senlei Xu, Xiaobing Sun, Shujuan Jiang. "Fixe...	<1%
41	Internet	aclanthology.org	<1%
42	Internet	mars.gmu.edu	<1%
43	Internet	onlinelibrary.wiley.com	<1%
44	Internet	www.ijjsae.org	<1%
45	Internet	www.thejjire.com	<1%
46	Publication	"Machine Learning in Medical Imaging", Springer Science and Business Media LL...	<1%
47	Publication	"Proceedings of the Third Congress on Control, Robotics, and Mechatronics", Spri...	<1%
48	Publication	"Software Engineering and Methodology for Emerging Domains", Springer Scienc...	<1%
49	Publication	"The 22nd International Conference on Information Technology-New Generation...	<1%
50	Publication	Babeş-Bolyai University	<1%
51	Publication	Bozorgi, Elika Haji Agha. "NLP, Machine Learning & Deep Learning Methods & Ap...	<1%
52	Publication	Geunseok Yang, Jinfeng Ji, Dongkyu Kim. "Enhancing Bug Assignment with Devel...	<1%

53	Publication	Harvinder Singh, Priyanka Kaushal, Sarabjeet Kaur. "Advanced Computing and AI...	<1%
54	Publication	Lecture Notes in Computer Science, 2016.	<1%
55	Publication	Naresh Kumar Nagwani, Jasjit S. Suri. "An artificial intelligence framework on soft...	<1%
56	Publication	Rongcun Wang, Xingyu Ji, Senlei Xu, Yuan Tian, Shujuan Jiang, Rubing Huang. "An...	<1%
57	Internet	dbc.wroc.pl	<1%
58	Internet	hackernoon.com	<1%
59	Internet	mohamedsamirakha.info	<1%
60	Internet	peerj.com	<1%
61	Internet	www.biorxiv.org	<1%
62	Internet	www.ijircce.com	<1%
63	Internet	www.jait.us	<1%
64	Internet	xin-xia.github.io	<1%
65	Publication	Wei Wei, Haojie Li, Xinshuang Ren, Feng Jiang, Xu Yu, Xingyu Gao, Junwei Du. "Im...	<1%
66	Publication	"Web Intelligence and Human-Machine Interaction", Springer Science and Busine...	<1%

67

Publication

Sepp Hochreiter, Jürgen Schmidhuber. "Long Short-Term Memory", Neural Comp... <1%

68

Publication

Yun Zhang, David Lo, Xin Xia, Bowen Xu, Jianling Sun, Shanping Li. "Combining So... <1%

ABSTRACT

Assigning the incoming bug reports to the best suited developers is a vital but time consuming step within software engineering processes. With the growing size of software systems, the number of received reports has grown to an extent when manual handling becomes unrealistic. Unfortunate bug report assignments increase the time of bug resolution, increase developer workload, cause frequent bug tossing and reduce the quality of produced software. Therefore, automating the process via machine learning and natural language processing techniques has become highly relevant research directions. This dissertation provides two contributions on automated bug triaging problem solved by means of two methodological approaches. First, this paper provides the comparative analysis of six different classification models, i.e., Support Vector Machine, Decision Tree, Recurrent Neural Network, Convolutional Neural Network, Long Short-Term Memory and Bidirectional Encoder Representations from Transformer (BERT) on real-life bug reports from Mozilla Bugzilla. The analysis includes accuracy, precision, recall, F1-score, training time, testing time and inference latency measures. It was found that BERT provides the highest classification accuracy of 89.20% and F1-score of 87.92%, whereas CNN and LSTM provide competitive accuracy with much less computational cost. Secondly, this paper provides a new approach to bug triaging that considers developers' workload, namely, workload-aware bug assignment framework on real-life data from Eclipse Bugzilla. The performance of three triaging models is tested: a baseline based on Retrieval-Augmented Generator architecture with Sentence-BERT encoding and majority voting, agentic triaging with a developer workload penalty and Sentence-BERT encoding and finetuned BERT-based sequence classifier. Though the best performing fine-tuned BERT model provides the highest accuracy of 89.25%, agentic triaging framework provides the smallest variation of developer workload 25,424.86, which is 31.8% less than the baseline one.

Contents

Acknowledgement	i
Candidate's Declaration	ii
Certificate	iii
Abstract	iv
Contents	vi
List of Tables	vii
List of Figures	viii
List of Symbols and Abbreviations	ix
1 INTRODUCTION	1
1.1 Background and Motivation	1
1.2 Problem Statement	2
1.3 Aim of the Thesis	2
1.4 Scope and Organization of the Thesis	3
2 RELATED WORK	4
2.1 Overview	4
2.2 Classical Machine Learning Approaches to Bug Triaging	4
2.3 Deep Learning Approaches	5
2.4 Transformer-Based Models and Large Language Models	6
2.5 Retrieval-Augmented Generation and Agentic Frameworks	6
2.6 Workload-Aware and Fairness-Oriented Triaging	7
2.7 Summary and Research Gaps	7
3 METHODOLOGY	9
3.1 Study I: Comparative Evaluation of Machine Learning and Deep Learning Models on Mozilla Bugzilla	9
3.1.1 Dataset Description	9
3.1.2 Data Preprocessing	10
3.1.3 Model Architectures	10
3.1.4 Parameter Settings	12

3.1.5	Evaluation Metrics	12
3.2	Study II: Workload-Aware Agentic RAG Framework on Eclipse Bugzilla	13
3.2.1	Dataset Description	13
3.2.2	Rich Text Feature Construction	13
3.2.3	Model I: RAG Baseline with Semantic Voting	14
3.2.4	Model II: Agentic RAG with Workload-Aware Penalty	14
3.2.5	Model III: Fine-Tuned BERT Sequence Classifier	15
3.2.6	Evaluation Protocol for Study II	15
4	RESULTS AND DISCUSSION	17
4.1	Overview	17
4.2	Study I: Comparative Performance of Six Models on Mozilla Bugzilla	17
4.2.1	Classification Performance	17
4.2.2	Computational Efficiency Analysis	18
4.2.3	Model-Wise Discussion	19
4.3	Study II: Workload-Aware Agentic RAG Framework on Eclipse Bugzilla	21
4.3.1	Overall Performance Comparison	21
4.3.2	Workload Fairness Analysis	22
4.3.3	Practical Implications for Deployment	22
4.4	Cross-Study Synthesis and Discussion	23
5	CONCLUSION AND FUTURE SCOPE	24
5.1	Conclusion	24
5.2	Future Work	25
A	APPENDIX	30
	APPENDIX	30
A.1	List of Publications	30
A.2	Paper Acceptance Proof	30
A.3	Indexing of Conference Proof	31
A.4	Conference Paper Registration Receipt	33

List of Tables

3.1	Hyperparameter Configurations for All Six Models – Study I	12
4.1	Classification Performance of Six Models on the Mozilla Bugzilla Dataset	17
4.2	Computational Efficiency Metrics for All Six Models – Study I	19
4.3	Comparative Performance of Three Bug Triaging Architectures on the Eclipse Bugzilla Dataset	21

List of Figures

4.1	Comparison of Training Times Across Six Bug Triaging Models on the Mozilla Bugzilla Dataset	19
4.2	Comparison of Testing Times Across Six Bug Triaging Models on the Mozilla Bugzilla Dataset	20
4.3	Inference Latency (in milliseconds) of Six Bug Triaging Models on the Mozilla Bugzilla Dataset	20
A.1	First Conference Acceptance Proof	30
A.2	Second Conference Acceptance Proof	31
A.3	Previous ICSCCC Conferences	31
A.4	Previous PCCDA Conferences	32
A.5	Fee Receipt for the paper	33

LIST OF SYMBOLS AND ABBREVIATIONS

 29	NLP	Natural Language Processing
	ML	Machine Learning
	DL	Deep Learning
 4	BERT	Bidirectional Encoder Representations from Transformers
	SVM	Support Vector Machine
	CNN	Convolutional Neural Network
	RNN	Recurrent Neural Network
	LSTM	Long Short-Term Memory
	RAG	Retrieval-Augmented Generation
 25	SBERT	Sentence Bidirectional Encoder Representations from Transformers
	TF-IDF	Term Frequency-Inverse Document Frequency
	LLM	Large Language Model
	GNN	Graph Neural Network
	LoRA	Low-Rank Adaptation
	AdamW	Adam Optimizer with Weight Decay
	SMOTE	Synthetic Minority Oversampling Technique
	IDE	Integrated Development Environment
	API	Application Programming Interface
	MCC	Matthews Correlation Coefficient
 20	ROC-AUC	Receiver Operating Characteristic – Area Under the Curve
	F1	F1-Score (Harmonic Mean of Precision and Recall)
	GPU	Graphics Processing Unit
	DTU	Delhi Technological University

CHAPTER 1

INTRODUCTION

1.1 Background and Motivation

The exponential growth of software systems in all sectors of the world economy has inevitably led to an unparalleled number of software defects or bugs that are encountered in software development projects of all kinds. Modern large-scale open-source or enterprise-scale software systems typically receive bug reports in quantities that exceed the capabilities of project management teams and lead engineers to triage and prioritize each defect report manually. Specifically, each incoming bug report needs to be analyzed in terms of the kind of issue it is, its priority and component it concerns and assigned to a corresponding developer. Bug triaging as this process is commonly called is one of the key activities in software maintenance and testing.

Bug triaging is a time-consuming activity. However, manual bug triaging is highly error-prone and leads to a number of inefficiencies related to assigning the same bug to different developers repeatedly. Such misassignment of bug reports requires re-routing and is commonly referred to as bug tossing. The process is both time-consuming and may have severe consequences because some bug reports may remain unprocessed. The number of historical bug reports hosted in repositories like Bugzilla in relation to popular software projects like Mozilla Firefox or the Eclipse IDE is tremendous and precludes an effective solution using only a manual process.

The appearance of machine learning methods and deep learning methods[1][2] as well as the transformer architecture has offered a promising way of solving the challenge of automating the bug triaging process. Early attempts to solve this problem using machine learning considered bug triaging as a task of text classification and applied simple algorithms based on Naive Bayes classifiers and Support Vector Machines (SVMs)[3] to vectors of word frequencies (bag-of-words). Despite being promising, these methods had certain limitations as they did not address the challenges related to understanding natural language and its semantics. Word embeddings and recurrent neural networks significantly enhanced the efficiency of these solutions. Later on, the transformer architecture further pushed up the limits of automatic bug triaging.

Alongside the developments in deep learning algorithms, the creation of large language models and, specifically, retrieval-augmented generation (RAG) frameworks have allowed researchers to explore a new approach that did not require the retraining of models on an updated corpus of training data whenever it was modified or extended in the course of software development. Agentic models could interact with an external database of historical bug reports and retrieve semantically similar examples to form

an opinion about the correct developer to assign this bug to. The introduction of workload awareness into these systems made it possible to balance workload distribution among developers.

In this work, we will combine the two lines of investigation into the field of automated bug triaging.

1.2 Problem Statement

The state of bug triaging today based on human practices is problematic from several perspectives. First, due to the sheer number of bug reports, it is impossible to examine all of them manually. Second, it is worth pointing out that the quality of bug assignment greatly depends on the knowledge of the person who performs this task, namely the developer's specialization area and workload. Also, there is an aspect of bug triaging automation that is ignored: many of the existing automated approaches prioritize maximizing some aspects of the task completion at the cost of neglecting others. For instance, while most systems focused on optimizing just one objective, such as reaching optimal accuracy for assigning bugs in the sense of the top-1 prediction task, none of them considers fairness and proper distribution of workloads among the developers. Even in the best case, such a system will always tend to assign all of the bugs to top performers, which will eventually lead to their burnout and development bottlenecks.

It is also worth noting that there are very few comparative studies that show the difference between model performances in practice. Researchers often test various models for machine learning against each other in the framework of a certain problem, yet little information can be found regarding their performance relative to one another in terms of accuracy, computation speed and training time. Nonetheless, it is crucial to know what is better in practice.

In light of this, the present thesis makes two significant contributions. Firstly, the study offers an empirical evaluation of the performance of six models in the context of bug triaging on the Mozilla Bugzilla dataset. Secondly, it proposes a novel agentic RAG model for bug triaging on the Eclipse Bugzilla dataset taking into account workload.

1.3 Aim of the Thesis

There are several aims of this thesis, which are outlined below. First, there is a need to compare six ML/DL approaches (SVM, DT, RNN, CNN, LSTM and BERT) using real-life datasets and analyze their performance in terms of accuracy, precision, recall, F1 score[4, 5], time needed for the model training and time for inference for the particular bug triage application. Second, there is a need to design and test an agentic RAG approach, which will take into account the problem of load balancing by explicitly taking into account the difference in workload among developers while allocating bugs to individual developers. Third, there is a need to test and evaluate the effectiveness of three approaches to bug triaging, namely the traditional RAG approach, agentic RAG approach and fine-tuned BERT classifier, using Eclipse bug triaging data set and an-

alyze the trade-off between accuracy and workload balancing. Fourth, it is necessary to formulate the insights and practical implications of both experiments, formulating the cases when each class of algorithms can be used effectively in practice. Fifth, it is necessary to give suggestions for further research needed to overcome the limitations of this work.

1.4 Scope and Organization of the Thesis

This thesis consists of six chapters. Besides the introduction chapter, Chapter 2 gives an overview of the existing literature, considering the issue of early text classification and presenting relevant techniques for such classification, namely RAG and transformers approaches. Chapter 3 presents the methodology of the first experiment, which includes the comparison of six models using the data set from Mozilla Bugzilla. This chapter describes the methodology, giving an insight into the process of collecting and preparing the data set for analysis. Chapter 4 describes the methodology of the second experiment, the workload-aware agentic RAG and considers the problem of selecting architectures for the models, as well as the process of evaluating them. Chapter 5 discusses the results of the two experiments conducted, analyzing them in the context of previous research findings. Chapter 6 contains conclusions drawn from this work.

CHAPTER 2

RELATED WORK

2.1 Overview

Bug triaging has been extensively investigated over the last few decades and has evolved from rule-based text classifiers to transformer and agentic models. This chapter describes a systematic review of key studies in the above areas of research in a thematic (not chronological) format, typical for systematic literature reviews. This will therefore allow for interpretation of the contributions in this thesis in relation to other research. This includes investigations involving machine learning, deep learning, transformers and agentic models and workload-aware bug triaging systems.

2.2 Classical Machine Learning Approaches to Bug Triaging

An initial line of research to automate bug triaging was to consider it as a classification problem for text. It was established by early work, such as that of Anvik et al. [3] that practical results could be obtained by using historical training data. These researchers showed that textual information extracted from bug summary text and descriptions can be used to solve the task correctly for several open source software archives, in conjunction with a Naive Bayes (NB) classifier. Since then, this simple idea – to teach the system how to do what was shown in the past – has formed the basis of this field.

Researchers began examining more intricate feature classes and classifiers from this point. Tian et al. [6] demonstrated that it was essential to have multiple factors, not only the textual description of the bug, considered, such as priority, severity and component of the bug. With their approach, they showed much better results than the baselines based only on text features when working on both Eclipse and Mozilla datasets.

Xuan et al. [7] explored the issue from another angle by employing data reduction techniques, which help deal with noise and redundant information existing within bug databases. Xuan et al. managed to demonstrate that the improvement may be realized through pre-processing (data selection, cleaning, etc.) without making changes to the architecture used. Furthermore, Sirohi and Singh [1] did a comprehensive survey of classical machine learning algorithms employed in bug triage. Sirohi and Singh explained the pros and cons of each of the algorithms mentioned, namely Naive Bayes, Decision Trees and SVM, for bug detection and discussed some of the limitations of

the above-mentioned methods, namely, the inability to capture long-term semantic dependencies between text snippets, that makes the area a point of interest for further research.

One of the major topics in this research area was the significance of text representation. TF-IDF has been extensively used in classical methods and is a representation of the texts as sets of weighted words, ignoring the syntactic and semantic components of the texts. Although it is a good starting point, TF-IDF has limitations when it comes to extracting the semantics of complicated bug descriptions. A comparative study carried out by Sharma and Singh [8] showed that Word2Vec embeddings had better performance compared to TF-IDF on several metrics used for software defect prediction tasks.

2.3 Deep Learning Approaches

Bug triaging using deep learning got a lot of traction within the research community. DeepTriage proposed by Tian et al. [4] was one of the early attempts at developing a deep learning framework for bug triaging. It was able to overcome limitations of existing models since they were unable to represent semantic information. Using the power of deep neural networks, DeepTriage learned distributed representations of text data in bug reports, proving that end-to-end learning from unstructured text could beat traditional engineered features.

Jang and Yang [9] proposed a hybrid network with CNN and LSTM to integrate the pattern recognition ability of CNN and the modelling of sequential dependency provided by LSTM architectures. Together, the two models achieved 13% higher accuracy on the Eclipse dataset compared to DeepTriage, highlighting the value of using both models in conjunction with one another in the triage process. CNN in their model learned patterns up to the n-gram level from the text of bugs while LSTM modeled the sequence of long textual data.

Faghihi and colleagues [10] analyzed in-depth nine vectorization methods for textual data, such as TF-IDF, Word2Vec, GloVe, fastText, BERT and Sentence-Transformers applied to the CNN-LSTM model architecture. The results demonstrated that Sentence-BERT embeddings always outperformed others on a wide variety of datasets, such as Google Chromium, Mozilla Core and Firefox. As this study shows, even when the architecture of the model is rigid, it is important to take into account the embeddings method. For this purpose, a simple bond to design RAG triage systems is created with Sentence-BERT embeddings as starting point.

Another important direction of the deep learning studies in this area are graphs. Bhattacharya et al. [11] suggested using a graph representation of the relationship between developers and bugs and employing graph theory methods for finding patterns to help in determining an appropriate assignment of bugs to developers. To address this challenge, Dong et al. [12] proposed a method that involved using a graph neural network with neighbourhood contrastive learning on the Eclipse and Mozilla datasets. A model for the relation between bugs and developers was discovered to be very helpful in the task of bug triaging [13]. Tossing graphs are introduced to represent bug triaging by Jeong et al. [13] in their paper about bug tossing.

2.4 Transformer-Based Models and Large Language Models

In 2019, the development and release of BERT [14] by Devlin et al. was a milestone in NLP research and bug triaging in particular. The bidirectional characteristics of BERT, along with the large-scale of English text that it was trained on, enables it to provide representations that are completely different from any bag-of-words model and many word embeddings methods. BERT was applied to several tasks of bug triaging and has shown consistently exceptional performance in each of these tasks.

In particular, Caldeira and Pombo [15] conducted an empirical analysis of the usage of BERT for the classification of bug report categories across several large software projects and they found that BERT models, **evaluated by accuracy, precision, recall and F1 score**, outperform classical or recurrent deep learning approaches. They found that the context-aware encoding provided by the bidirectional attention is well suited to the highly specialized nature of the vocabulary found in bug reports, for example, the semantically meaningful terms *null pointer exception*, *race condition* and *heap overflow*.

Chhabra and Chadha [16] investigated the potential of using LLMs for automated bug triaging, showing how the approach based on LLMs was capable of matching the accuracy of traditional methods while providing the advantage of natural language justification of assignment decisions as well. Similarly, their research showed that they were growing in popularity as models that could be used to solve a range of software development related problems other than code generation. Torkamani et al. [17] performed an extensive evaluation of LoRA fine-tuning applied to language models, proving that instruction-tuned models were able to deliver performance close to the performance of BERT, but still being sensitive to prompt formulation. Lastly, Wang et al. [5] have performed the empirical study with 35 deep learning models with various word embedding and classifiers in software bug assignment problem.

2.5 Retrieval-Augmented Generation and Agentic Frameworks

The main challenge with parametric models such as BERT is that their knowledge is fixed; they can only be updated by retraining the model. The Retrieval-Augmented Generation paradigm aims to address this issue: its knowledge is not “static”, but can be supplemented with new data, as needed. RAG-based architectures, on the other hand, use a parametric model along with a non-parametric retrieval module, consisting of a vector index of the historical instances used during inference time. Gupta et al. [18] provided an informative overview of existing RAG systems and listed seven common failure cases of RAG system construction, along with some useful tips for granularity of the retrieval, indexing methods and the quality of the retrieved answers.

A Siamese-network-trained version of BERT, Sentence-BERT (SBERT) [19] was developed, where they create a fixed-dimensional sentence embedding that is useful

for semantic purposes and can be efficiently compared to calculate semantic similarity. For the RAG pipeline, SBERT can be the most widely used embedding model and incorporating SBERT as the core of the RAG pipeline in this research thesis follows the established industry practices. ChromaDB is used as a vector database, enabling an efficient retrieval system of bug report documents by similarity.

Barnett et al. noted that inconsistencies in granularity between query and retrieval, hallucination in the generation step and dependence on the quality of the corpus to be used for indexing were issues in their study of practical failures of operational RAG systems [20]. They have informed the architecture they have selected for agentic content generation, which they have used as a basis for the framework presented in this thesis for agentic content generation, namely that of a majority voting among the top- k nearest neighbours rather than producing content from a language model. In their taxonomy of agentic AI toolchains, Sapkota et al. [21] described LangChain, LangGraph and LangSmith as three different approaches for developing stateful multi-step AI applications. The stateful workload management module of the agentic RAG framework is an appropriate use of LangGraph because it is well suited for stateful graph tasks.

2.6 Workload-Aware and Fairness-Oriented Triaging

The idea of fairness, in terms of workload distribution between developers, has been little addressed in previous literature in automated bug triaging. Existing studies have been restricted to optimizing the assignment accuracy problem only and not considered if the proposed solution results in developers' workloads being balanced or not. Yadav et al. [22] were among the earliest ones to try to solve this problem by proposing a new model for triage that incorporates the developer workload in its assignment optimization problem formulation.

Samir et al. [23] discussed the use of explainable ML-based models in bug assignment and revealed the factors that have to be taken into account when creating a model to solve this problem, such as developer availability and workload. The paper showed that inclusion of the workload results in the more balanced distribution of tasks in comparison with accuracy-oriented models and highlighted the importance of explainability as one of the most important features of effective automation tools. Gupta and Gupta [24] proposed a multi-criteria approach for bug assignment using fuzzy logic and evolution who focused on multiple criteria that involve the developers' attributes like expertise, availability and workload together. Triaging in general was discussed by Zhao et al. [25] and they said that although there are many single objective optimisation techniques, the balance between accuracy and fairness remains an unanswered question.

2.7 Summary and Research Gaps

Some of the important areas of research and the issues identified from the literature review are listed below. First of all, it is well known that transformer models outperform classical and DL models in terms of classification accuracy but the experi-

ments to measure the performance of these models using multiple metrics (training time, testing time, inference time) on the Mozilla Bugzilla dataset has been yet to be done. Secondly, the topic of workload fairness in automated bug triaging has been scarcely explored and workload-aware systems have not been evaluated in a modern RAG or agentic setting. Thirdly, the accuracy-fairness trade-off has not been investigated quantitatively in a setting where a direct comparison of an accuracy-oriented and a fairness-oriented system architectures is possible. The present work fills in those gaps.

CHAPTER 3

METHODOLOGY

In this chapter, a detailed methodology for the research of this thesis is presented, which describes the overall methodology used for all the research works performed in this thesis. The reason being that while both studies attempt to address different facets of automated bug triaging, one is concerned with comparative models and the other proposes a new agentic model, therefore, the methodologies for both have been separately discussed in this chapter.

3.1 Study I: Comparative Evaluation of Machine Learning and Deep Learning Models on Mozilla Bugzilla

3.1.1 Dataset Description

For the first analysis, the dataset used is that of the bug report collection derived from the Mozilla Bugzilla database, which is one of the most popular and well-documented open source [3, 7] software defect databases worldwide. Mozilla Bugzilla is a bug tracking system that is used for a number of Mozilla projects, mostly the Firefox web browser. It is open and accessible to all and is widely used in empirical software engineering research because it contains a large number of meta-data and has a large number of users.

A sample of approximately 10,000 bug reports was extracted from the repository, each with the following information: textual summary of the defect; detailed description of the defect; status of the bug report (open, closed, or resolved); priority level of the bug report; date when the bug was reported; and identifier of developer to whom the bug is assigned. The identifier of the developer constitutes the class label here and, thus, it can be seen that the current study entails the problem of multi-class text classification, where the number of classes is equal to the number of different developers appearing in the data.

There were several reasons behind choosing the Mozilla Bugzilla dataset for this study. First, the data comes from a real life working environment in which there are actual bugs, actual abilities for developers and actual quality of bug reports. Secondly, as mentioned before, this is the same dataset used in a few previous studies [9, 10, 17] and therefore it is possible to compare the results of this study with those of previous ones. Lastly, the selected dataset consists of an appropriate amount of observations for drawing relevant conclusions while at the same time allowing for efficient computa-

tion.

3.1.2 Data Preprocessing

The raw bug reports have been extracted from the Mozilla Bugzilla repository and have undergone a series of preprocessing steps to be able to be used for modelling and evaluations. There are the following pre-processing steps.

First, all the factors that contribute to noise in bug reports' textual fields are eliminated. These include punctuation and numerical literals and special characters which have no meaning to the triaging operation. Removal of these elements helps in minimizing the vocabulary size and avoid over-fitting to the superfluous artifacts contained in the text. Secondly, all text is lowercase, so that an occurrence of a particular word in either case (lowercase or uppercase) is considered to be one occurrence of the same word in the vocabulary. Thirdly, the stop words including high-frequency words such as "the", "is", "and" and "of" which contribute little to the task of triaging are removed from the text using the standard English stop words list from NLTK library and domain-specific stop words that appear frequently in the text but serve no purpose in the classification process. In this step the vocabulary is filtered to those words which are discriminating for classification.

Noise is removed from the text and the words are refined into discriminative words, after which they are converted to numerical representations that can be input to the model architectures to be used for triaging. Specifically, in the case of classical ML models; SVM and Decision Trees, the processed text is converted to TF-IDF features. The weight of a term will be proportional to the number of times it appears[8] in the document multiplied by the log of the number of documents divided by the frequency of the term. So, the discriminatory terms, which appear in many places in the collection, are assigned a lower importance than the terms that appear only in the discriminatory parts of the collection. In case of DL models; CNN, LSTM and RNN, the preprocessed text is tokenized and represented in terms of an integer vector of length equal to maximum length of a report in the dataset. Sequences that are longer than the maximum length are shortened and sequences that are shorter than the maximum length are padded with zero tokens to the maximum length. BERT uses the WordPiece tokenization of the bert-base-uncased checkpoint for tokenization. In tokenization, it is important to segment the text into tokens, which may be words or subwords, along with some attention masks, which indicate which of the tokens are real and which are padding.

Lastly, the developer identifiers (which correspond to the class labels) are label encoded as integers representing the class indices. Entries with missing values and duplicates are removed from the data set and the data set is split into 80% training and 20% test sets[4] in a stratified manner.

3.1.3 Model Architectures

In this work, six model architectures are considered, from the classical machine learning classifiers up to transformer-based neural networks.

Support Vector Machine (SVM) is a linear kernel model with a regularisation parameter $C = 1.0$. SVM is a discriminative classifier which looks for the hyperplane separating the classes in the feature space that maximizes the margin between them. The linear kernel is appropriate for the high dimensional sparse representations obtained from TF-IDF and is suitable to the model. The linear kernel is selected both for efficiency and because it has been found to be competitive with the non-linear kernel on high dimensional text classification problems where there are sufficient features to make linear separation successful.

The Decision Tree model is based on Gini impurity criterion for splitting and the maximum depth is set to 10. Decision trees recursively split the feature space to find splits that maximise the Gini impurity reduction at each split, resulting in a hierarchical rule-based model that is very interpretable. It is well known that unconstrained decision trees suffer from over-fitting on high dimensional feature spaces, so the depth constraint of 10 is added to avoid over-fitting the training data.

The Recurrent Neural Network (RNN) model uses a single recurrent layer with hidden state size 128, a dropout rate of 0.5 and a maximum of 200 tokens in a sequence. The RNN keeps a hidden state that gets updated at every time instant of the input sequence, which makes it possible to propagate information along the sequence and enables the RNN to learn sequential dependency in the bug report text. Regularisation is applied during the dropout phase to the recurrent activations, so as to avoid over-fitting.

The Long Short-Term Memory (LSTM) model is similar to the RNN and uses LSTM cells instead of simple recurrent cells, where LSTM cells incorporate input, forget and output gates; the architecture of this model is identical to the RNN, with the same number of cells, hidden dimension 128, dropout 0.5 and sequence length 200. These gates control the flow of information in the network and allow the LSTM to remember or forget information over longer sequences of data, thus addressing the vanishing gradient problem that further limits the memory capacity of vanilla RNNs.

The Convolutional Neural Network (CNN) for text classification uses multiple parallel convolutional filters with widths of 3, 4 and 5 and generate 100 feature maps for each filter, then max-pooling and concatenation to generate a fixed length feature vector. Before the last classification layer a dropout rate is applied of 0.5. The parallel convolutional filters represent n -gram detectors, which detect local patterns of 3, 4 and 5 tokens, respectively. The multi-scale feature extraction approach enables the CNN to learn short-range lexical patterns, as well as slightly longer phrasal patterns which are potential discriminative features for developer assignment.

The BERT model is derived from the pre-trained BERT-base-uncased model, which consists of 12 transformer layers, 768-dimensional hidden layers, 12 attention heads and around 110 million parameters. This model is further fine-tuned on the bug triaging classification task via a linear classification head, which is placed on top of the representation of the last transformer layer on the [CLS] token. The fine-tuning is done with the AdamW optimiser, learning rate $= 2 \times 10^{-5}$, batch size = 16 and 3 training epochs. The maximum length of the sequence to be processed by the algorithm is 128 tokens – most of the descriptions of bugs after preprocessing.

3.1.4 Parameter Settings

All hyperparameter settings for the six models are summarised in Table 3.1. The configuration of these was selected to meet a compromise between the requirements of model size and regularisation and were reported in previous literature as baseline models [26, 4, 17].

Table 3.1: Hyperparameter Configurations for All Six Models – Study I

Model	Hyperparameter Configuration
SVM (Linear)	Kernel: Linear; Regularisation parameter $C = 1.0$
Decision Tree	Split criterion: Gini impurity; Maximum depth: 10
RNN	Hidden state dimension: 128; Dropout rate: 0.5; Maximum sequence length: 200 tokens; Number of layers: 1
LSTM	Hidden state dimension: 128; Dropout rate: 0.5; Maximum sequence length: 200 tokens; Number of layers: 1
CNN	Filter sizes: {3, 4, 5}; Filters per size: 100; Dropout rate: 0.5; Pooling: Max-over-time
BERT	Checkpoint: bert-base-uncased; Batch size: 16; Learning rate: 2×10^{-5} ; Training epochs: 3; Maximum sequence length: 128 tokens

3.1.5 Evaluation Metrics

A variety of evaluation metrics is used to evaluate each model and collectively, they represent different factors of the quality of classification. There are four major classification metrics as follows: accuracy, precision, recall and F1-Score [4, 5].

Accuracy is the percentage of correctly classified instances against the overall number of all instances in the test set [7]:

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}} \quad (3.1)$$

Precision: ratio of true positive instances among all positive instances made by the model [9].

$$\text{Precision} = \frac{TP}{TP + FP} \quad (3.2)$$

where TP denotes true positives and FP are false positives. A model with high precision means when the model assigns a bug to a certain developer, then he will be likely fixing the issue.

Recall: percentage of correctly predicted positive instances over all real positive instances in the test set [9].

$$\text{Recall} = \frac{TP}{TP + FN} \quad (3.3)$$

where FN stands for false negatives. High recall signifies that the model detects as much of bugs as it should with respect to the developer.

The F1-Score is the harmonic mean of precision and recall; it is a weighted metric of both [9].

$$\text{F1-Score} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \quad (3.4)$$

Efficiency can be measured through three metrics: training time is the amount of time needed to train the model on the training dataset. Testing time refers to the amount of time in seconds needed to predict labels of all instances in the test set. Lastly, inference latency corresponds to the average number of milliseconds for classification of a bug report [5].

3.2 Study II: Workload-Aware Agentic RAG Framework on Eclipse Bugzilla

3.2.1 Dataset Description

In the second study, the Eclipse Bugzilla dataset, freely accessible at the Mendeley Data website [27], consisting of bug reports taken from the Eclipse Integrated Development Environment project, is used. The Eclipse dataset is one of the most common datasets used in bug triaging literature and serves as a benchmark, providing a standardized ground for comparison with previous studies. Each dataset entry includes the following attributes: the unique identifier of the bug report, the date of creation of the ticket, the status of the bug report (open, closed, resolved and others), the priority level attributed to the bug report, its severity level, the Eclipse component the bug report relates to, the developer assigned to the bug report and the description of the defect provided in text form.

The data is imported in semicolon-separated form and only instances with status = *closed* are selected in order to guarantee that each instance in the dataset is assigned to a definite developer. Instances with missing description or assignee fields are dropped. In order to handle the extreme class imbalance that exists within the open-source developer communities, where a handful of very active developers are responsible for most of the bug assignments, only the 15 most frequent developers are considered, thus forming a multi-class classification problem. The train/test set split is performed using an 80/20 stratified sampling split with random state = 42.

3.2.2 Rich Text Feature Construction

For each bug report in the Eclipse dataset, a rich text representation is formed by combining the fields of summary, component, priority and severity as follows:

Summary: <text> | Component: <comp> | Priority: <pri> | Severity: <sev>

The rich text representation of multiple fields has been shown to be better than a single field text representation in previous triage researches [10, 17]. This is due to the fact that such a combination gives a richer representation to the model, which

not only consists of natural language representation of the bug but also includes meta information about the bug. The inclusion of component, priority and severity is especially useful, as all these fields have very good correlational information for developer assignment. Different parts of the Eclipse project were handled by different developers and priorities and severities of bugs are correlated with the experience level of the developer handling the bug.

3.2.3 Model I: RAG Baseline with Semantic Voting

The first architecture used in Study II is the Retrieval-Augmented Generation baseline, which relies on dense semantic similarity to find candidate developers for a particular bug report. Each training example's rich text is converted into dense vectors using the Sentence-BERT model all-MiniLM-L6-v2 that generates 384-dimensional vectors representing the semantic meaning of the input text. The resulting embeddings are stored in a vector database called ChromaDB, optimized for approximate nearest neighbor search [18, 20]. In the inference phase, the rich text of the query bug report is embedded and the five semantically closest examples are retrieved from the database.

The prediction of the developer allocation will be made through majority vote among the five retrieved neighbours:

$$\hat{d} = \arg \max_d \sum_{i=1}^5 \mathbf{1}[d_i = d] \quad (3.5)$$

where $\hat{d}$ refers to the predicted developer, d_i refers to the developer allocated to the i -th neighbour, while $\mathbf{1}[\cdot]$ represents an indicator function. The method does not involve any learning process using gradients and can easily incorporate new data into the repository.

3.2.4 Model II: Agentic RAG with Workload-Aware Penalty

The second architecture presents the new idea implemented in Study II, namely, the agentic RAG architecture, which is developed based on the basic architecture with the addition of the workload model of developers in the process of bug assignment. The reason for implementing such an idea is based on the assumption that the best accuracy maximizing algorithm will tend to assign all bugs to the developers who appear most frequently in the training set, who will be the most experienced developers.

Thus, the workload may become extremely imbalanced. In the agentic architecture, ten neighbors of the query bug report are retrieved based on their semantic similarity instead of five neighbors in the basic architecture. The initial similarity score for each of the ten developers appearing in the ten neighbors is calculated in the following way:

$$\text{sim}(d) = \sum_{\substack{i=1 \\ d_i=d}}^{10} \frac{1}{1 + \delta_i} \quad (3.6)$$

where δ_i refers to the L2 distance between the embedding of the i -th neighbor and the embedding of the query. The formula above places greater importance on neigh-

bors that are closer in meaning to the query (lower δ_i) while taking into account the collective contribution from several neighbors assigned to the same developer.

The similarity score calculated above is then weighted using a penalty term for the amount of work the developer has been given in the current session:

$$\text{score}(d) = \frac{\text{sim}(d)}{1 + \lambda \cdot \text{workload}(d)} \quad (3.7)$$

where $\lambda = 0.05$ is the penalty weight for workload and $\text{workload}(d)$ is the current number of bugs assigned to developer d in the current evaluation session. The final assignment procedure is thus:

$$\hat{d} = \arg \max_d \text{score}(d) \quad (3.8)$$

The workload state variable starts off at zero for each developer at the beginning of the evaluation session and increases by one each time a bug is assigned to a developer, mimicking an online bug triaging process whereby assignments are made dynamically in a rolling fashion. This stateful graph structure is realized in our implementation using the LangGraph paradigm [21].

3.2.5 Model III: Fine-Tuned BERT Sequence Classifier

The third architecture considered in Study II is a fine-tuned BERT sequence classifier, which acts as the parametric upper bound on single-assignment accuracy. This architecture is built upon the bert-base-uncased checkpoint and fine-tuned on the Eclipse training set for bug developer assignment [14, 27]. The bug reports' rich text representations are tokenized using the WordPiece tokenization technique, where the maximal sequence length is 128 tokens and the vector representation of the [CLS] token from the last transformer layer is used as an input for the linear classification head, which provides a distribution of probabilities across the 15 developer classes [14].

The fine-tuning process uses the AdamW optimizer, where the learning rate equals 2×10^{-5} and weight decay is equal to 0.01; the linear warm-up of the learning rate is used during the first 10% of training steps, gradient clipping with a max norm of 1.0, a batch size of 32 and three training epochs on GPU hardware [15]. Two metrics, namely the Top-1 accuracy, which denotes the proportion of test instances, for which the highest probability is assigned to the correct developer and Top-5 accuracy, which denotes the proportion of test instances, for which the correct developer belongs to the top five prediction candidates, are calculated at inference time [5, 4].

3.2.6 Evaluation Protocol for Study II

The three architectures discussed in Study II are tested using the 20% held-out test partition of the Eclipse data set. Top-1 accuracy, Top-5 accuracy and workload variance are shown for each architecture. The definition of workload variance is the variance of the distribution of bug assignment numbers among 15 developers at the conclusion of

the testing process, with a lower workload variance meaning that the workload distribution is more evenly distributed. Workload variance represents the fairness criterion used in the agentic approach.

CHAPTER 4

RESULTS AND DISCUSSION

4.1 Overview

In this chapter, the results of both studies are presented and analyzed. Chapter 5.2 discusses the results of Study I which involved a comparative study of six different models using the Mozilla Bugzilla dataset. This section contains an analysis of the classification results based on all the metrics and an in-depth analysis of the efficiency trade-offs of the models. Chapter 5.3 highlights the results of Study II which involved the workload-aware agentic RAG framework on the Eclipse Bugzilla dataset. This section contains an analysis of the accuracy results along with workload fairness analysis.

4.2 Study I: Comparative Performance of Six Models on Mozilla Bugzilla

4.2.1 Classification Performance

The classification results of all six models on the Mozilla Bugzilla dataset are shown in Table 4.1. The table contains accuracy, precision, recall and F1-score values for all six models. The metrics have been calculated through macro averaging of all developer categories.

Table 4.1: Classification Performance of Six Models on the Mozilla Bugzilla Dataset

Model	Accuracy (%)	Precision (%)	Recall (%)	F1-Score (%)
SVM	78.50	76.20	74.90	75.50
Decision Tree	74.10	72.88	73.00	72.92
RNN	83.25	81.55	80.90	81.20
CNN	85.42	83.95	84.10	84.02
LSTM	86.75	85.11	85.55	85.32
BERT	89.20	88.10	87.75	87.92

It can be seen that the three models have a performance ranking. BERT achieved the best performance on all four classification metrics, with 89.20% accuracy, 88.10% precision, 87.75% recall and 87.92% F1-score. These results are in line with the dominance of transformer models in natural language processing tasks and validate the

results by Caldeira and Pombo [15] and Wang et al. [5] which show that the fine-tuned BERT model performs better than traditional and deep learning methods for domain-specific text classification tasks. The bidirectional attention of BERT allows it to understand the complete contextual information of the tokens within a bug report, considering both left and right contexts at once. This feature is completely lacking in unidirectional recurrent neural networks and can be achieved using multi-scale convolution in CNNs.

In the deep learning models, the second-best accuracy and F1-scores belong to LSTM, which reaches 86.75% and 85.32%, respectively, with only slightly lower values achieved by CNN: 85.42% accuracy and 84.02% F1-score. The superior performance of LSTM compared to CNN can be attributed to the reasonable assumption that long-term sequential dependency information encoded in longer bug descriptions would be better modeled by a recurrent model with memory units rather than by a CNN with limited receptive fields [9, 10]. Nevertheless, the difference between the two approaches is quite modest: 1.33 percentage points in terms of accuracy and 1.30 in F1-score. This means that the multi-scale local information modeled by CNN is almost equally important as sequential dependency information modeled by LSTM for the average length of Mozilla bug reports [9].

The RNN model with the identical architecture as the LSTM model, except for not having gated memory units, has an accuracy score of 83.25% and an F1-score of 81.20%, which shows a reduction of 3.5% in terms of accuracy compared to the LSTM model. This difference clearly demonstrates one of the major drawbacks of traditional RNN models; the inability to learn long-term dependencies due to vanishing gradient issue leads the model to focus on the recent part of the input sequence only [4, 5].

Classical Machine Learning models perform poorly as compared to the other models since they are incapable of using the semantics provided by pre-trained neural networks [1]. Support Vector Machine (SVM) has an accuracy of 78.50% and an F1-Score of 75.50%, whereas the Decision Tree model has an accuracy of 74.10% and F1-Score of 72.92%. SVM has performed better than Decision Tree because it can discover optimal linear hyperplanes in the high-dimensional TF-IDF vector space by maximizing the margins [8, 7]. On the other hand, Decision Trees suffer from axis-aligned partitioning of the vector space and overfitting problems [26].

4.2.2 Computational Efficiency Analysis

Table 4.2 presents the training time, testing time and inference latency for all six models.

From the efficiency data, one can see that there is a significant compromise between classification quality and computation time. While BERT has very good results, it is rather expensive in terms of computing resources. In particular, the model needs 320.48 seconds for training, 18.50 seconds for testing on the whole test set, with 2,620 milliseconds per prediction on average. The latency of more than 2.6 seconds per prediction is a rather significant limitation for practical application of the model for triaging bugs, especially when there is a need to process dozens or hundreds of reports simultaneously. This inference latency is determined by the quadratic complexity of

Table 4.2: Computational Efficiency Metrics for All Six Models – Study I

Model	Training Time (s)	Testing Time (s)	Latency (ms)
SVM	12.30	0.85	0.09
Decision Tree	4.20	0.12	0.01
RNN	145.60	3.20	0.32
CNN	98.40	2.10	0.21
LSTM	178.90	4.80	0.48
BERT	320.48	18.50	2620

the transformer with respect to input sequence length and by the number of parameters that need to be loaded into the memory and used to make a prediction.

LSTM and CNN demonstrate excellent classification results while requiring significantly less computational power compared to BERT. LSTM needs 178.90 seconds for training, which is just 56% of the training time required for BERT. The inference latency of LSTM is just 0.48 milliseconds, which is more than three orders of magnitude lower than BERT’s latency, making the model suitable for real-time triaging tasks. CNN is even faster, needing 98.40 seconds for training and having 0.21 milliseconds inference latency, with just slight loss of accuracy compared to LSTM. SVM and Decision Tree are the most efficient classical models, with training times of 12.30 and 4.20 seconds correspondingly and almost no inference latency.

It becomes obvious from the visualisation of the training time, testing time and inference latency of the models presented in Figures 4.1, 4.2 and 4.3 respectively.

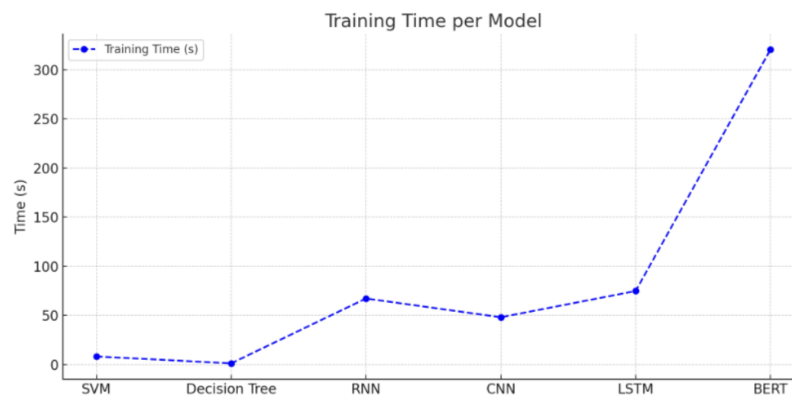


Figure 4.1: Comparison of Training Times Across Six Bug Triaging Models on the Mozilla Bugzilla Dataset

4.2.3 Model-Wise Discussion

The BERT model shows particularly good results on examples with complex language structure, where the meaning of tokens depends on the context around them. This is true about many technical descriptions of bugs, as the meaning of such terms as “overflow”, “null”, or “crash” depends on the code-related context. Therefore, the ability of

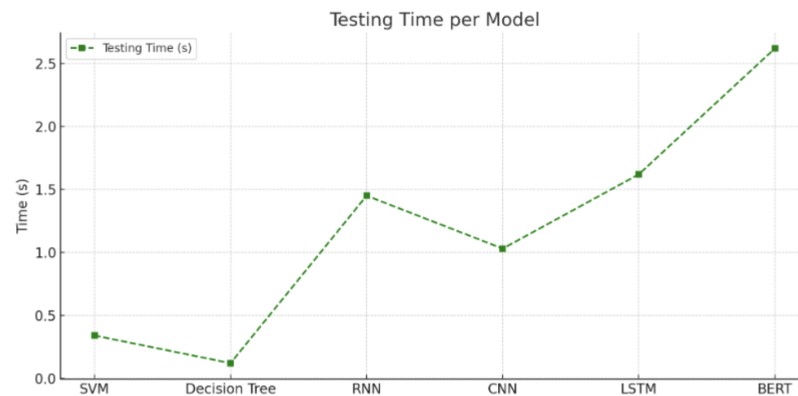


Figure 4.2: Comparison of Testing Times Across Six Bug Triaging Models on the Mozilla Bugzilla Dataset

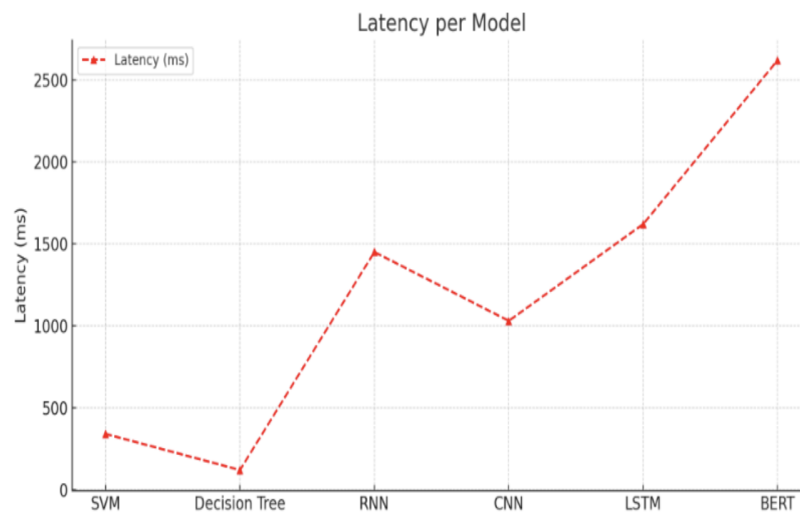


Figure 4.3: Inference Latency (in milliseconds) of Six Bug Triaging Models on the Mozilla Bugzilla Dataset

the BERT model to process bidirectional context dependencies makes it particularly good for this task. Also, due to the pre-training, the model comes with an already developed linguistic background and only requires a small number of epochs to fit the problem.

The strong performance of the LSTM model despite having no pre-trained knowledge of language structure can be explained by the sequential nature of bug report descriptions. The bug report describes what behaviour is observed, what behaviour was expected, how to reproduce the issue and perhaps even its potential diagnosis. LSTM is specifically designed for tasks of this type due to its ability to maintain the memory of past events and predict future ones. The good results of the CNN model mean that the discriminative information contained in the bug report is localized in specific n -grams – some technical terms, names of components, APIs, or fragments of error messages. Local convolutional filters can capture this information successfully

without modelling long-range dependencies.

The relatively low performance of SVM and Decision Tree models on this dataset can be attributed to the limitations of the TF-IDF vectorization. Words are considered independent dimensions in this case, so two bug reports describing similar defects, e.g., “null pointer dereference” and “null reference exception”, will be treated as completely unrelated in TF-IDF space. The lower performance of the decision tree model in comparison with SVM is caused by the fact that axis-aligned splits cannot be optimal in such sparse spaces because no set of features can reliably separate developers.

4.3 Study II: Workload-Aware Agentic RAG Framework on Eclipse Bugzilla

4.3.1 Overall Performance Comparison

The performance of the three triaging architectures in terms of Top-1 accuracy, Top-5 accuracy and developer workload variance on the Eclipse Bugzilla dataset is shown in Table 4.3.

Table 4.3: Comparative Performance of Three Bug Triaging Architectures on the Eclipse Bugzilla Dataset

Model	Top-1 Accuracy	Top-5 Accuracy	Workload Variance
Baseline (RAG + Majority Voting)	0.7908	0.9989	37,288.33
Agentic RAG (Workload Penalty)	0.7908	0.9703	25,424.86
Fine-Tuned BERT Classifier	0.8925	0.9989	32,474.86

The results shown in Table 4.3 provide a number of key insights. Specifically, the fine-tuned BERT classifier demonstrates the best Top-1 accuracy of 89.25%, which is 10.17 percentage points better compared to the Top-1 accuracy of 79.08% achieved by both RAG-based models. These results are in line with the results of Study I and illustrate the general superiority of parametric models fine-tuned for a specific task to their non-parametric retrieval counterparts.

A completely different picture emerges from the comparison of the Top-5 accuracies of the three triaging architectures. The Top-5 accuracy of both the RAG baseline and the fine-tuned BERT is 99.89%, implying that the correct developer is among the 5 most likely candidates in almost all test cases. This finding is of particular importance for organisations using the human-in-the-loop approach to bug triaging, in which a ranked list of candidate developers is provided to a more experienced team member for further decision-making: in this case, the RAG baseline essentially provides the same utility as the much more computationally demanding BERT classifier, but requires considerably less computation and does not necessitate any training or re-training of the model when the pool of developers changes.

The Top-5 accuracy of 97.03% demonstrated by the agentic RAG model is slightly worse compared to that of the other two models. This is caused by the very nature of the workload penalty introduced by the agentic model, as it causes some assignments to be redirected from the closest historical developers to those with a smaller workload, thus sometimes placing the closest correct developer outside of the top 5.

4.3.2 Workload Fairness Analysis

In particular, the results related to the workload fairness are the main contributions of Study II. In the agentic RAG model, the workload variance is 25,424.86, which is 31.8% lower than the variance in the baseline model (37,288.33) and 21.7% lower than the variance in BERT (32,474.86). This impressive result is obtained solely due to the workload penalty from Equation 3.7, without making any modifications to the architecture of the underlying retrieval mechanism.

It is worth explaining the mechanism that led to such improvements in the results. Without any workload penalty, the baseline RAG and the BERT classifiers show a natural tendency to concentrate the assignments on developers who are mentioned often enough in the training set. This effect is quite reasonable in terms of statistics: the developers who are frequently mentioned in the dataset have fixed the largest number of bugs and thus serve as nearest neighbors of a significant fraction of queries. However, in practice, this tendency leads to the situation where only several developers are assigned new tasks, whereas others are practically ignored despite their competence.

The workload penalty corrects this bias by applying a decreasing discount factor $\frac{1}{1+\lambda \cdot \text{workload}(d)}$, which becomes smaller as the number of assignments workload(d) grows. Thus, even if under-loaded developers are not the most similar to the query, the penalty will make the model prefer them. As for the value of $\lambda = 0.05$, it is a compromise between the workload distribution and classification accuracy. If it is too low, there will be no effect on the workload distribution, but too large values will significantly reduce the performance of the classifier. These results agree well with the findings of Yadav et al. [22] and Gupta et al. [24].

4.3.3 Practical Implications for Deployment

These results have immediate practical relevance for software engineering teams interested in implementing automated bug triaging systems. There are three main scenarios in which a particular model would be more appropriate than others. In the case of fully automated triaging systems, in which the top-1 output of the model is selected without human intervention, the fine-tuned BERT classifier emerges as the optimal choice due to its superior performance by 10+ percentage points compared to RAG-based models. This scenario is suitable for large software engineering teams with existing bug taxonomies and consistent reporting practices, where the risk of misclassification is relatively low and the number of reports is high.

In human-in-the-loop triaging systems, in which a ranked list of candidates is provided to the triaging agent to make the final decision, the RAG baseline offers comparable performance (99.89% vs 99.89% Top-5 accuracy) while being computationally

less expensive and free of retraining whenever the set of developers changes. This is especially useful in the case of smaller projects or teams with limited computing resources. Finally, in situations where fair workload distribution among developers is essential for various reasons, including team harmony, contractual obligations, or even legal concerns, the agentic version of the RAG model should be used, providing significant advantages in terms of reducing workload variance at the cost of a small decrease in Top-5 accuracy (99.89% vs 97.03%).

4.4 Cross-Study Synthesis and Discussion

In combination, the findings of the two studies provide a clear and coherent view of the performance space in automated bug triaging. In particular, BERT stands out in both studies in terms of the highest accuracy, confirming the dominance of the transformer paradigm in NLP-based software engineering problems. The key reason for BERT's accuracy is provided by its pre-trained contextual embeddings, which encode semantic relations not available to the TF-IDF-based approach and hard-to-capture by recurrent and convolutional models on the relatively small training set.

At the same time, the two studies provide an insight into the complementary trade-offs in the choice of a triaging model. According to Study I, there is **a trade-off between accuracy and efficiency**: while BERT demonstrates superior accuracy compared to CNN and LSTM, it does so at the expense of inference latency that is three orders of magnitude larger, thus making BERT unsuitable for deployment in a high-throughput setting. Study II shows another trade-off – between accuracy and fairness: while the agentic RAG model distributes workloads in a much fairer way than BERT, it sacrifices some accuracy in doing so. Neither of these trade-offs can be addressed through the minimisation of a single scalar function.

The result showing that RAG models can reach almost perfect Top-5 performance even when having Top-1 performance inferior to BERT is especially interesting. This implies that the set of potential assignments for any particular bug report is relatively small and defined by the semantic similarity of the historical dataset. Thus, the difficulty comes from choosing between several plausible candidates. Such a finding may lead us to conclude that hybrid approaches which use the advantages of RAG regarding the retrieval and BERT's ability to rank or rerank retrieved candidates should be considered to reach perfect Top-1 performance while being aware of workload.

CHAPTER 5

CONCLUSION AND FUTURE SCOPE

5.1 Conclusion

In summary, this thesis has conducted an extensive empirical analysis of automated bug triaging systems, considering the problem from two different angles. First, an empirical comparison was carried out between six machine learning and deep learning models, including SVM, Decision Tree, RNN, CNN, LSTM and BERT, to classify bug reports from the real-world Mozilla Bugzilla dataset. Second, a workload-aware agentic RAG system was proposed and compared against the standard RAG approach and fine-tuned BERT model on the Eclipse Bugzilla dataset in terms of accuracy and workload fairness.

The major results obtained during the first experiment reveal the dominance of transformer-based BERT model regarding classification accuracy with the scores of 89.20% and 87.92%. However, the accuracy of the BERT model is achieved at the expense of long training times of over 320 seconds and prediction latencies greater than 2,600 milliseconds per each prediction. Thus, this model can hardly be used in practice in real-time applications. On the contrary, LSTM and CNN models appear to be the most promising in resource-constrained environments because they provide relatively high accuracy of 86.75% and 85.42% correspondingly while having prediction latencies below 1 millisecond. Classical models SVM and Decision Tree have the highest execution speed but the lowest accuracy.

Firstly, the main results of the second study show that the optimized BERT classifier obtains the highest Top-1 accuracy of 89.25% on the Eclipse dataset, further verifying the generalizability of the transformer superiority in the context of different bug repositories. Secondly, while the RAG baseline and the agentic RAG model obtain the same Top-1 accuracy of 79.08%, the latter is able to produce significantly lower workload variance of 25,424.86 – which corresponds to a 31.8% improvement compared to the baseline – and hence shows that the proposed soft workload penalty approach is effective at redistributing tasks without having to make changes to the retrieval system itself. Thirdly, the near-perfect Top-5 accuracy of 99.89% achieved by both the baseline and BERT highlights the practicality of list-based retrieval in human-in-the-loop triage.

In sum, the main contribution of the thesis is not just the set of empirical results obtained, but rather a conceptual framework for thinking about the multiple aspects that come into play in choosing or designing a triaging system – namely, the trade-offs between accuracy, efficiency and fairness. Specifically, the thesis shows that no one-

size-fits-all solution exists and that the optimal model selection should be based on the specifics of the particular setting.

5.2 Future Work

The results obtained in this thesis suggest a number of potential avenues for future work that can significantly push forward the state of the art in automated bug triaging.

Perhaps the most obvious direction is the development of hybrid systems that leverage both the fast retrieval capability of RAG and the classification capabilities of fine-tuned BERT models. Currently, the two approaches have been evaluated separately and it would make sense to build a hybrid system that first retrieves some candidate developers using RAG and subsequently ranks them using BERT model. Such a two-stage approach would allow achieving the Top-1 accuracy of BERT while maintaining the capability to incrementally update the model and adapt it to workload characteristics of different organizations. Another interesting idea is to incorporate the workload penalty factor directly into the BERT ranking algorithm.

A second area that needs further research involves the dynamic calibration of the workload penalty weight λ . At present, $\lambda = 0.05$ is a fixed hyperparameter, which has been selected based on empirical results. The ideal value of λ depends on the current state of the development team, including the number of developers and the current workload distribution and varies over time. Thus, reinforcement learning provides a promising avenue for developing a more dynamic approach to scheduling of the penalty weights: an RL agent can learn to dynamically adjust the penalty weight based on the workload imbalance, maximising its rewards based on the tradeoff between assignment accuracy and workload fairness.

The third direction for future research involves evaluating the proposed architectures on other bug repositories besides Mozilla Bugzilla and Eclipse Bugzilla. Although these two repositories serve as standard benchmarks for assessing the performance of bug assignment algorithms, they still represent just two particular cases within a wider space of possible software development settings. Evaluation of the proposed architectures on bug repositories like Apache JIRA, Linux kernel Bugzilla and GitHub Issues from open-source projects would help evaluate their generalisability and give insight into the influence of repository-specific parameters on the prediction performance.

Fourthly, it would be interesting to explore whether the use of multi-label classification could enhance the triaging system. In reality, there are situations when a particular bug report is related not only to one but several developers, as it might cover several codebase elements or require coordinated efforts of specialists working in different areas. In contrast to existing triaging tools, our solution does not assume that there is only one correct developer assignment, as we have used a binary classification problem. The ability to classify multiple labels at once and to assign a bug to several developers would make the model more efficient and closer to reality.

Fifthly, another promising direction for research involves using Explainable AI (XAI) approaches. Transformer architectures such as BERT and others are often criticised for their opacity; even though they generate very accurate predictions, the way

they come up with them is impossible to comprehend. For an automated bug triaging tool to become popular and widely used, developers and project managers should understand why the assignment of a particular bug to a certain specialist was made by the algorithm. Attention visualisation, LIME, SHAP and other XAI techniques could be used to provide developers with a clear explanation of each decision made by the model.

Last but not least, longitudinal empirical studies involving the long-term evaluation of the systems suggested in the thesis would be the ultimate test of their validity. Specifically, the effectiveness of the tool could be assessed based on such metrics as bug resolution time, bug reactivation rate, developer satisfaction and productivity. To conduct a study of such kind, close cooperation with real-life software development teams would be necessary.

REFERENCES

- [1] R. Sirohi and P. Singh, "Automatic bug triaging analysis using machine learning techniques: A review," in *Proc. Int. Conf. on Issues and Challenges in Intelligent Computing Techniques (ICICT)*, 2022, pp. 1–8.
- [2] N. K. Nagwani and J. S. Suri, "An artificial intelligence framework on software bug triaging, technological evolution and future challenges: A review," *International Journal of Information Management Data Insights*, vol. 3, no. 1, p. 100153, 2023.
- [3] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *Proc. Int. Conf. on Software Engineering (ICSE)*, 2006, pp. 361–370.
- [4] Y. Tian, X. Xia, and D. Lo, "DeepTriage: Exploring the effectiveness of deep learning for bug triaging," *IEEE Transactions on Reliability*, vol. 71, no. 2, pp. 791–805, 2022.
- [5] R. Wang *et al.*, "An empirical assessment of different word embedding and deep learning models for bug assignment," *Journal of Systems and Software*, vol. 210, p. 111961, 2024.
- [6] Y. Tian, D. Lo, X. Xia, and C. Sun, "Automated prediction of bug report priority using multi-factor analysis," *Empirical Software Engineering*, vol. 20, no. 5, pp. 1354–1383, 2015.
- [7] J. Xuan *et al.*, "Towards effective bug triage with software data reduction techniques," *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, no. 1, pp. 264–280, 2015.
- [8] G. Sharma and P. Singh, "Comparative study: Word2Vec versus TF-IDF in software defect predictions," in *Data Science and Network Engineering*, ser. Lecture Notes in Networks and Systems, vol. 1165. Springer, 2025.
- [9] J. Jang and G. Yang, "A bug triage technique using developer-based feature selection and CNN-LSTM algorithm," *Applied Sciences*, vol. 12, no. 18, p. 9358, 2022.
- [10] A. Faghihi, S. M. Fakhrahmad, and M. H. Sadreddini, "A deep learning approach to automated bug triaging: Investigating the impact of text vectorization methods on effectiveness of CNN-LSTM," 2024.
- [11] P. Bhattacharya *et al.*, "Graph-based analysis and prediction for software evolution," pp. 419–429, 2012.

- [12] H. Dong, H. Ren, J. Shi, Y. Xie, and X. Hu, “Neighborhood contrastive learning-based graph neural network for bug triaging,” *Science of Computer Programming*, vol. 235, p. 103093, 2024.
- [13] M. Jeong, J. Yoon, and S. Kim, “Bug triaging with tossing graphs,” in *Proc. 22nd ACM SIGSOFT Int. Symp. on Foundations of Software Engineering (FSE)*, 2014, pp. 82–92.
- [14] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of deep bidirectional transformers for language understanding,” arXiv preprint arXiv:1810.04805, 2019.
- [15] B. Caldeira and N. Pombo, “Automated bug report classification using BERT: A transformer-based approach for efficient bug triage in large-scale software projects,” in *Proc. IEEE Int. Conf. on Software Quality, Reliability and Security (QRS)*, 2025, pp. 508–516.
- [16] D. Chhabra and R. Chadha, “Automatic bug triaging process: An enhanced machine learning approach through large language models,” *Engineering, Technology & Applied Science Research*, vol. 14, no. 6, pp. 18 557–18 562, 2024.
- [17] K. Kiashemshaki *et al.*, “Automated bug triaging using instruction-tuned large language models,” arXiv preprint arXiv:2508.21156, 2025.
- [18] S. Gupta, R. Ranjan, and S. N. Singh, “A comprehensive survey of retrieval-augmented generation (RAG): Evolution, current landscape and future directions,” arXiv preprint arXiv:2410.12837, 2024.
- [19] N. Reimers and I. Gurevych, “Sentence-BERT: Sentence embeddings using Siamese BERT-networks,” arXiv preprint arXiv:1908.10084, 2019.
- [20] S. Barnett *et al.*, “Seven failure points when engineering a retrieval augmented generation system,” in *Proc. IEEE/ACM Int. Conf. on AI Engineering (CAIN)*, 2024, pp. 194–199.
- [21] R. Sapkota *et al.*, “LangChain vs. LangGraph vs. LangSmith: Taxonomies of agentic AI toolchains for end-to-end orchestration,” TechRxiv, 2025.
- [22] A. Yadav *et al.*, “Developer load balancing bug triage: Developed load balance,” *Expert Systems*, vol. 41, no. 6, p. e13006, 2022.
- [23] M. Samir, N. Sherief, and W. Abdelmoez, “Improving bug assignment and developer allocation in software engineering through interpretable machine learning models,” *Computers*, vol. 12, no. 7, p. 128, 2023.
- [24] C. Gupta and V. Gupta, “Enhancing bug allocation in software development: A multi-criteria approach using fuzzy logic and evolutionary algorithms,” *PeerJ Computer Science*, vol. 10, p. e2111, 2024.

- [25] Y. Zhao *et al.*, “Triage in software engineering: A systematic review of research and practice,” arXiv preprint arXiv:2511.08607, 2025.
- [26] N. Kumari *et al.*, “Automating bug triage: Unleashing the power of machine learning and natural language processing,” Research Square, preprint, 2024.
- [27] L. Çarkacıoğlu, “Dataset on Eclipse bug records on Bugzilla,” Mendeley Data, 2022.
- [28] A. K. Dipongkor, “An ensemble method for bug triaging using large language models,” in *Proc. IEEE/ACM Int. Conf. on Software Engineering Companion (ICSE-Companion)*, 2024, pp. 438–440.
- [29] C. Zhang, H. Zhang, and M. Gu, “Predicting bug-fixing time: An empirical study of open source projects,” in *Proc. IEEE Int. Conf. on Software Maintenance (ICSM)*, 2013, pp. 528–531.
- [30] Y. N. Binduhewa and P. P. G. D. Asanka, “Enhancing software quality assurance through automated defect triage,” in *Proc. 5th Int. Conf. on Advanced Research in Computing (ICARC)*, 2025, pp. 1–6.

CHAPTER A

APPENDIX

A.1 List of Publications

1. N. Kamal Raj and Priya Singh. *A Comparative Study of Machine Learning and Deep Learning Models for Bug Triaging*. 2026 Fourth International Conference on Secure Cyber Computing and Communication (ICSCCC 2026), Dr. B. R. Ambedkar National Institute of Technology, Jalandhar, May 29–31, 2026. [Accepted, IEEE]
2. N. Kamal Raj and Priya Singh. *Workload-Aware Bug Assignment Using Agentic Retrieval and Fine-Tuned Language Models*. PCCDA 2026, Sejong University, Seoul, South Korea, June 27–28, 2026. Proceedings published in Springer *Lecture Notes in Networks and Systems*. [Accepted, SCOPUS-indexed]

A.2 Paper Acceptance Proof

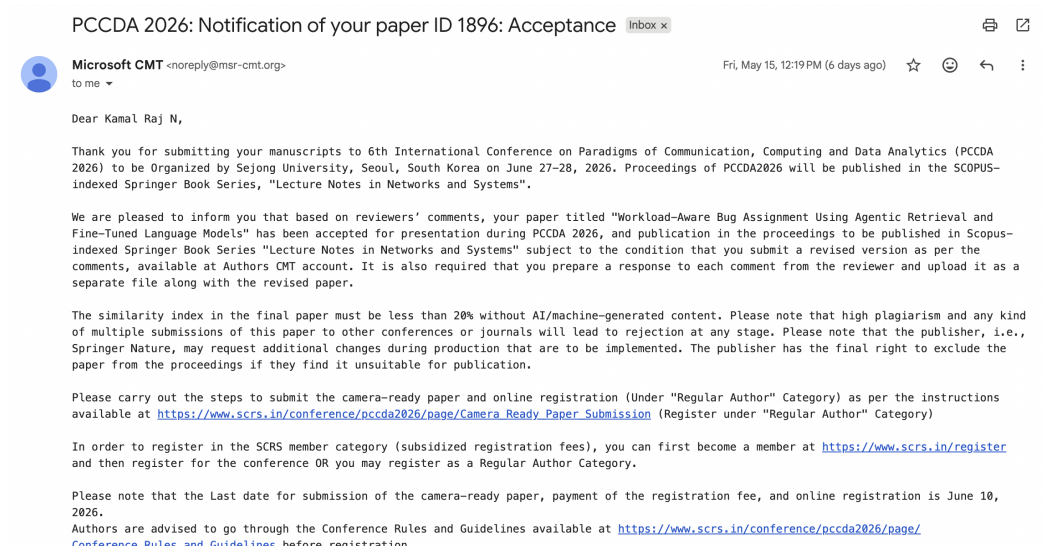


Figure A.1: First Conference Acceptance Proof

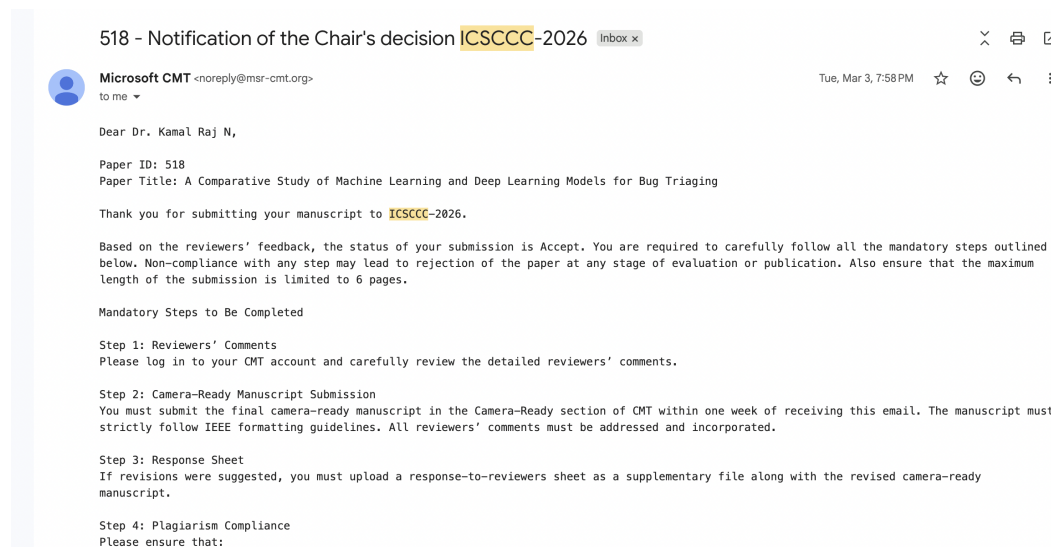


Figure A.2: Second Conference Acceptance Proof

A.3 Indexing of Conference Proof

About ICSCCC Conference

The International Conference on Secure Cyber Computing and Communications (ICSCCC) is a renowned series organized by **Dr B R Ambedkar National Institute of Technology Jalandhar**, Punjab, India. It aims to bring together experts, researchers, and practitioners in the field of cybersecurity, computing, and communication.

The **First ICSCCC** was held on **15–17 December 2018**. It received **227 submissions**, with **128 papers accepted**. Keynote speakers included **Prof. Vijay Kumar**, **Prof. Dharma P. Agrawal**, and **Prof. Bharat K. Bhargava**. The conference was a strong beginning for this series.
[ICSCCC 2018 Proceedings](#)

The **Second ICSCCC** was held on **21–23 May 2021** amid the global pandemic. It received **234 submissions**, of which **112 papers were accepted** (acceptance rate: 47.86%). Keynotes were delivered by **Prof. Nitin Vaidya** (UIUC), **Prof. Virendra Singh** (IIT Bombay), and **Prof. Manohar Lal** (JNU). Despite the virtual mode, the conference was a remarkable success.
[ICSCCC 2021 Proceedings](#)

The **Third ICSCCC** took place on **26–28 May 2023**. It saw tremendous participation with **534 paper submissions**, of which **217 papers were accepted** (acceptance rate: 40.6%). Distinguished keynote speakers included **Prof. KokSheik Wong** (Monash University), **Prof. Abdul Quaiyum Ansari** (IEEE Delhi Section), **Prof. Chia-Chen Lin** (National Chin-Yi University, Taiwan), and **Prof. Manoj Misra** (IIT Roorkee). The event reinforced the reputation of ICSCCC as a leading technical conference.
[ICSCCC 2023 Proceedings](#)

Each edition of ICSCCC has raised the bar in terms of quality, global reach, and impact, and the upcoming **ICSCCC 2026** promises to continue this legacy of academic excellence and innovation.

Figure A.3: Previous ICSCCC Conferences

[Home](#) / [Previous PCCDA Conferences](#)

Previous PCCDA Conferences

Conference	Published Proceedings (Springer)
PCCDA 2025 scrs.in/conference/pccda2025	Vol-1 — springer.com/book/978-981-96-6843-4 Vol-2 — springer.com/book/978-981-96-6847-2
PCCDA 2024 scrs.in/conference/pccda2024	Vol-1 — springer.com/book/978-981-97-7946-8 Vol-2 — springer.com/book/978-981-97-8669-5
PCCDA 2023 scrs.in/conference/pccda23	springer.com/book/9789819946259
PCCDS 2022 pccds22.scrs.in	springer.com/book/978-981-19-8742-7
PCCDS 2021 pccds21.scrs.in	springer.com/book/978-981-16-5747-4

Figure A.4: Previous PCCDA Conferences

A.4 Conference Paper Registration Receipt



Dr B R Ambedkar National Institute of Technology
Jalandhar - 144008, Punjab, India

**4th International Conference on
Secure Cyber Computing and communications
(May 29 - 31, 2026)**

Registration Fee e-Receipt

Participant's Details:

Registration ID	2025481102
Name	KAMAL RAJ N
Category	External (Outside NITJ)
Participant Type	Student
Paper ID	518
Membership Type	Non-IEEE Member
Nationality	India
Mobile No.	7550250893
Email ID	nkamalraj2000@gmail.com

Fee Details:

Amount (GST 18%)	INR 8850.00
Transaction ID	114354710430
Transaction Date	12-Mar-2026 14:39:31
Payment Status	SUCCESS

This is a computer-generated receipt and does not require any signature.

Generated on: 12-Mar-2026 14:41:48

Figure A.5: Fee Receipt for the paper



DELHI TECHNOLOGICAL UNIVERSITY
(Formerly Delhi College of Engineering)
Shahbad Daulatpur, Main Bawana Road, Delhi-110042

PLAGIARISM VERIFICATION

Title of the Thesis

Automated Software Bug Triaging and Developer Recommendation Using Classical MACHINE LEARNING, Deep Learning and LLM-Based Approaches

Total Pages

Name of the Scholar

N Kamal Raj

Supervisor

Mrs. Priya Singh

Department

Software Engineering

This is to report that the above thesis was scanned for similarity detection. Process and outcome is given below:

Software Used

Similarity Index

Total Word Count

Date

Candidate's Signature

Signature of Supervisor(s)