

Bug Severity Classification in Software Projects using Natural Language Processing

by

Asset Kanatov

Submitted to the Department of Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

at the

NAZARBAYEV UNIVERSITY

June 2026

© Nazarbayev University 2026. All rights reserved.

Author
Department of Computer Science
11.04.2026

Certified by
Askar Boranbayev
Associate Professor, PhD
Thesis Supervisor

Certified by
Talgar Bayan
Postdoctoral Scholar, PhD
Thesis Supervisor

Accepted by
Yelyzaveta Arkhangelsky
Dean, School of Engineering and Digital Sciences

Bug Severity Classification in Software Projects using Natural Language Processing

by

Asset Kanatov

Submitted to the Department of Computer Science
on 11.04.2026, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science

Abstract

Identifying the severity and the further summary of the bug highly depends on the person performing this task. The manual job is time-consuming and error-prone. As this issue influences the software development pace, it can be fully automated, which this thesis tries to achieve. The goal of the research is to develop an NLP-driven application for bug severity classification. The study searches for a perfect solution by comparing traditional ML algorithms with LLMs. In terms of the LLMs, the choice fell on Meta's Llama 3 8B and on Alibaba's Qwen 2.5 7B. At the first stage of evaluation there were classic TF-IDF ML algorithms, which expectedly hit the ceiling of 50% in F1 score. Next, the zero-shot LLMs failed to overcome the traditional models. Finally, thanks to the Low-Rank Adaptation (LoRA), the fine-tuned LLMs managed to surpass the baseline and demonstrate the F1 score of 62%. As regards the one-to-one comparison, Llama 3 performed slightly better than Qwen 2.5. Apart from the benchmarks, there was developed a modern application with a decoupled LLM pipeline. The system operates with React on the frontend and FastAPI on the backend. The application allows users to upload bugs in various formats and export the analysis into Jira via CSV format. The main outcome is the practical applicability of locally hosted and domain-adapted LLMs for automating the bug triage process, improving the Agile workflows.

Keywords: Bug Classification, Bug Severity, Large Language Models, Natural Language Processing, LoRA, Software Engineering, Bug Triaging, Machine Learning

Thesis Supervisor: Askar Boranbayev

Title: Associate Professor, PhD

Thesis Supervisor: Talgar Bayan

Title: Postdoctoral Scholar, PhD

Acknowledgments

I would like to express my gratitude to Professor Askar Boranbayev, who supervised me throughout this research. Thanks to his guidance and revisions, I was able to deliver an academic paper and develop a project that I really strived for. Also, I am really grateful to Dr. Talgar Bayan, who acted as a co-supervisor for my thesis. Next, a big thanks to the faculty of Nazarbayev University. I am grateful to the School of Engineering and Digital Sciences that provided me with the valuable knowledge throughout 2 years. The university positively influenced my life and I am happy to be part of this community, graduating with a Master's degree. On a personal note, I would love to mention my parents, who believed in me throughout this academic journey. Also, my groupmates, who I shared these valuable years with. This achievement is partly thanks to their costly impact.

Contents

1	Introduction	11
1.1	Background and Motivation	11
1.2	Problem Statement	13
1.3	Research Objectives and Questions	15
1.4	The LLM Approach	17
2	Related works	21
2.1	Bug Triaging and Its Complexities	21
2.2	Difference in a Semantic Understanding of the Bug Reports	23
2.3	Data Noise and Class Imbalance Challenges	30
2.4	Domain Adaptation with a Fine-Tuning Method	32
2.5	Summary and Research Gap	34
3	Methodology	39
3.1	Introduction	39
3.2	Dataset Attributes and Preprocessing	40
3.3	Pipeline of Baseline Models	43
3.4	Pipeline of LLMs	45
3.5	Evaluation Framework	51
3.6	System Architecture	53
3.7	Reproducibility Tips	56
4	Results	57

4.1	Evaluation Setup	57
4.2	Performance of Traditional Models	59
4.3	Evaluation of Zero-Shot LLMs	63
4.4	Evaluation of Fine-Tuned LLMs	65
4.5	Computational Latency	68
4.6	Web Application and Importing into Jira	72
5	Discussion	77
5.1	Addressing Research Questions	77
5.1.1	Research Question 1	77
5.1.2	Research Question 2	78
5.1.3	Research Question 3	79
5.1.4	Research Question 4	80
5.2	Comparison of Llama and Qwen	81
5.3	Practical Application	82
5.4	System Limitations	85
6	Conclusion	87
6.1	Research Journey	87
6.2	Key Takeaways	88
6.3	Future Roadmap	89

List of Figures

1-1	Bug Triage Ticket. Source: [28].	13
1-2	Comparison of Traditional and AI Methods.	15
1-3	System Architecture at the High Level.	17
2-1	Architecture of LSTM in Text Classification. Source: [4].	26
2-2	Chain of Thought Prompting in Comparison. Source: [50].	29
2-3	Comparison of Zero-shot and Few-shot Models. Source: [19].	35
2-4	Evolution of the ML Models in Bug Classification.	36
2-5	Low-Rank Adaptation of LLMs in a Diagram. Source: [37].	37
3-1	Solution of a Data Imbalance Problem.	40
3-2	The process of TF-IDF Text Vectorization. Source: [34].	42
3-3	The Stack for LLM Fine Tuning.	45
3-4	Impact of LoRA hyperparameters on the LLM fine tuning	48
3-5	Structured Prompting in a JSON Format.	50
3-6	Training and Testing Data Split.	51
3-7	Decoupled System Design.	55
4-1	Dataset Distribution in Each Pipeline.	57
4-2	Formulas Calculating the Model Performance.	59
4-3	Baseline Model Performance.	60
4-4	Voting Ensemble Confusion Matrix Heatmap.	62
4-5	Comparison of Zero-Shot LLMs with Baselines.	64
4-6	Reasons of Zero-Shot LLMs Failure.	64

4-7	The Performance Comparison of Baselines, Zero-Shot and Fine-Tuned LLMs.	66
4-8	Class Distribution For Fine-Tuned Llama 3.	68
4-9	Comparison of Average Inference Time across Baselines and LLMs. .	69
4-10	Diagram of a Decoupled Architecture in the LLM Pipeline.	71
4-11	Bug Ingestion Interface.	73
4-12	Analysis of the Bug with Four Fields.	73
4-13	Bug History with an Export Function.	74
4-14	Import Process in Jira.	75
4-15	Successfully Imported Tasks in Jira.	75
5-1	Trade-Off in Accuracy and Inference Latency.	81
5-2	Flowchart of Automated Bug Triaging in a SDLC.	84

Chapter 1

Introduction

1.1 Background and Motivation

Nowadays, software has become an inalienable part of our lives. It is integrated everywhere, starting from digital payments, ending up with government services. As we use and rely on it every day, we need the applications to work perfectly. In order to reach this goal, the developers use specific methods for releasing updates more quickly. It is called continuous integration or continuous deployment (CI/CD). This CI/CD pipeline allows developers to send an update in one click, and if something does not work, the process stops and lists the errors. It creates a huge amount of data, which is valuable for the quality assurance engineers and project managers. They are able to extract those bug reports and identify the problems in the software, by further creating tasks in their backlog. As the amount of data grows, it turns into a big technical and organizational challenge for the software teams.

It is undeniable that software with bugs can lead to huge financial losses. The research made by Adhikari et al. states that the USA loses a striking \$2.08 trillion every single year because of the poor software quality [1]. The reason behind this is not only the arising bugs, but how inefficiently they are fixed or even prioritised. The human factor is major in this case, as after identifying the root cause of the bug, it all comes to the project manager to evaluate the priority, so-called bug triaging. In most IT companies, when the bug is reported, the system architects and analysts

work together, in order to read the report and understand how serious it is. At the end of the process, the responsible person, such as the project manager, manually assigns the priority and which team should fix it. As you can see, the chain is long and not everyone can be in the context, which leads to the mistakes in prioritization. Moreover, as the projects grow, humans are simply not able to keep up with this pace. As a result of that, more bugs get wrong priorities, and they get delayed in being fixed.

This problem is really noticeable in the high user base projects. They might receive thousands of fresh bug reports every day. As an example, you can take Eclipse IDE or Chromium browser; their bug reports are available open source. In the research made by Prerna et al., this case is described as a “scale and complexity problem” [40]. To simplify, as the application scales, the more data flows into the reports and it is harder to process it. As a negative outcome, it creates a massive backlog, where developers clear up the bugs, but inefficiently and in the wrong order. Inside an IT company, the time of the developer is golden. For that reason, classifying the tasks on severity is a significant process, as it dictates which issues will be fixed first and directly impacting the software quality.

This research aims to make the step of bug classification highly automated. Natural Language Processing (NLP) allows the machines to understand the textual data as an input. After applying NLP on bug reports, we can reach the automatic classification with the use of machine learning (ML). Potentially, it can reduce time and boost the accuracy significantly. The software teams can allocate more time on fixing, instead of sorting the bugs by severity. Apart from that, developers already extensively use AI tools for code generation, which increases their efficiency. The same question applies to the bug prioritization, why not apply AI here, in order to reach the highest accuracy in a short amount of time? With impressive news about advanced AI agents that work locally on the user’s device, this is where the industry is flowing.

Summary

Structured Features

Unstructured Features

Issue Type

Priority

Affected Version

Component

Environment

Observed Behavior

Expected Behavior

Test Cases

Stack Trace

Camel / CAMEL-5782

regression : invalid SetQueueAttributesRequest created, works on 2.10.1

Details

Type: Bug

Priority: Minor

Affects Version/s: 2.9.4, 2.10.2

Component/s: camel-aws

Labels: None

Environment: > \$ uname -a Linux pc-nc277 3.2.0-3-amd64 #1 SMP Mon Jul 23 02:45:17 UTC 2012 ...

Estimated Complexity: Unknown

Status: CLOSED

Resolution: Fixed

Fix Version/s: 2.9.5, 2.10.3, 2.11.0

Description

In org.apache.camel.component.aws.sqs.SqsEndpoint.updateQueueAttributes, if I don't have any configuration, the created SetQueueAttributesRequest contains a null attribute collection and AWS emit an error.

In 2.10.1, no problem.

Workaround in 2.10.2 : force the create SetQueueAttributesRequest to contain a valid attribute collection by defining a configuration in camel.

For example:

```
from("aws-sqs://queue?amazonSQSCient=#amazonSQSCient&delay='pollCycle.getMillis()'*&maxMessagesPerPoll=10&deleteAfterRead=f")
```

-> works on 2.10.1, fail on 2.10.2

if I add an argument to my URI "defaultVisibilityTimeout=30"

-> works on 2.10.1, works on 2.10.2

Caused by: org.apache.camel.FailedToCreateRouteException: Failed to create route SQS-to-MongoDB-EVENTS: Route[[From[aws-sqs://EVENTS?amazonSQSCient=#amazonSQSCient... because of Failed to resolve endpoint: aws-sqs://EVENTS?amazonSQSCient=%23amazonSQSCient&delay=60000&deleteAfterRead=false&maxMessagesPerPoll=10 due to: The request must contain the parameter Attribute.Name.

at org.apache.camel.model.RouteDefinition.addRoutes(RouteDefinition.java:176) ~[camel-core-2.10.2.jar:2.10.2]
at org.apache.camel.impl.DefaultCamelContext.startRoute(DefaultCamelContext.java:722) ~[camel-core-2.10.2.jar:2.10.2]
at org.apache.camel.impl.DefaultCamelContext.startRouteDefinitions(DefaultCamelContext.java:1789) ~[camel-core-2.10.2.jar:2.10.2]

Figure 1-1: Bug Triage Ticket. Source: [28].

1.2 Problem Statement

The main purpose of bug triaging is to give each bug a severity. Even though it seems easy to assign a tag like major or minor, it is in fact difficult. There are many factors that influence the severity and later a priority in the backlog. Firstly, the bug reports are messy, as they contain logs and chunks of the code, taken out of context, as shown in Fig. 1-1. In order to understand it, the AI model has to know the technical language, not just be an algorithm trained on a specific dataset. Traditional models would fail in such a scenario, as they are just looking for specific keywords. Moreover, each software company has its own culture and pace, the model should be fine-tuned for such cases as well. To clarify, the same bug may be marked as major in one team and minor in another, solely based on their own constraints. Some teams may have a sprint full of tasks and tight deadlines, that is why that major bug will be moved to

the minor priority. Apart from that, it of course depends on the domain of each IT company and the functionality of their app. Additionally, the team may simply want to have an equal distribution of major or minor tasks, whereas other teams would like to have a 75-25 distribution in any direction. This process is context dependent, that is why the bug prioritization is a hard process.

In the majority of IT companies, this task is dedicated to humans. While it was an alternativeless method, it led to inconsistency and mistakes. Importantly, the people responsible for bug triaging always change and everyone has their own biases. Jahanshahi et al. discussed this issue and also mentioned that developers triaging the bugs often suffer from “myopia” [24]. It explains how the bug reports get a biased evaluation, by one person working in isolation. People working on the evaluation tend to forget the overall project goals or the current tasks in progress. Due to this reason, every person will deliver a different prioritization, made based on their own beliefs instead of the data driven approach. It was proven that 44% of bug reports struggle from an incorrect priority assignment [1]. The consequences of this problem are wild. Imagine if a critical issue was labeled as minor at the beginning, then it lies in the backlog for weeks. Meanwhile, the production server experiences crashes, dropping considerably in the quality rating.

In the scientific work organized by Hong et al. they discuss the reason for the simple automation failure, so called “organizational flux” [21]. It can be explained by the fact that the team members rotate and the goals inside of the projects always change. Imagine an automated tool, that was trained on one data and the next week the CEO decided to take a pivot, changing the prioritization completely. Such static rules inside of the model would be useless. Basically, this month one feature is prioritized, the next month it is already out of date and not needed. Subsequently, it leads to the conclusion that systems with defined rules will not be applicable to the agile teams in software development.

As previously mentioned, the traditional models cannot comprehend the technical language that well, resulting in a low accuracy in severity assignment. Also, the manual work done by humans lacks consistency and it is time-consuming. In Fig. 1-2

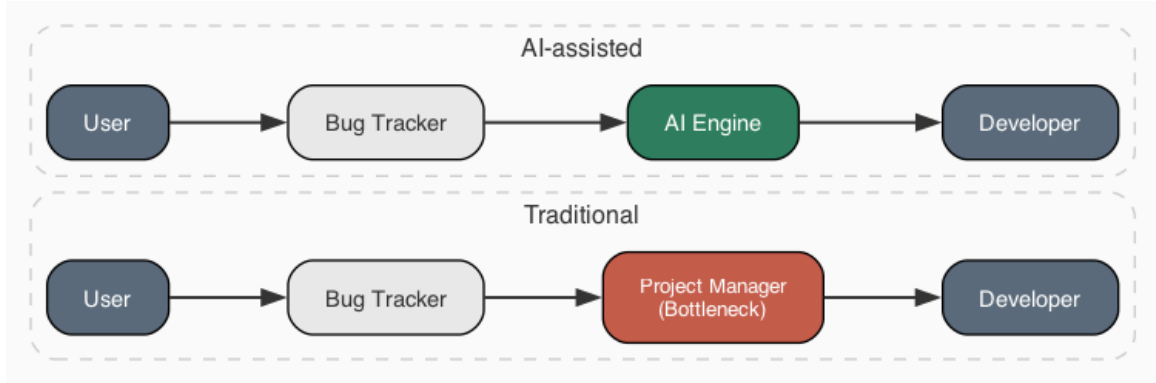


Figure 1-2: Comparison of Traditional and AI Methods.

you can witness the algorithm of bug triage and where it limps. There is always a high possibility of mistakes. The combination of these bottlenecks results in a research gap made in the IT industry. The software world needs a robust system that will not just search for specific marker words, but will be able to make a reasoning through the code logs. This advanced model would be comparable with a senior software engineer, saving millions of dollars for the company.

In today's world the engineers tend to check metadata more. Metadata is structured information about version, OS, database, file format, etc. Nevertheless, they miss a huge source of data hidden in the unstructured report. This thesis aims to design a tool that will be able to scan through the technical text and quickly analyze it on a high engineering level. If teams continue with the current approach, they will keep losing resources like time and money. It is clear that the current pace of the software development is extremely fast, AI is used in coding, leading to more frequent releases and more bug reports extracted each day. The software companies undoubtedly require advanced AI models to go through those reports with a deep analysis.

1.3 Research Objectives and Questions

Previous studies already tried to implement the algorithms that classify the bugs by severity and assign a priority for the task. However, they were based on traditional

models. Such models were only able to assign a tag like critical or minor. The biggest downside is that they could not provide the improved contextual representation. In case of this research, the modern transformers were used to handle the noisy bug reports from the open-source repositories.

The main objective of this research is to use Large Language Models (LLMs) on the bug severity classification. For that purpose there will be designed a User Interface (UI) that will accept the user's input, then evaluate the severity based on the probability. As you can see in Fig. 1-3, the system functions with a use of React on frontend and LLM connected via FastAPI. In terms of the LLM choice, two of the famous models were utilized. Llama 3 8B, which was designed by Meta, and Qwen 2.5 7B made by Alibaba Cloud. Hou et al. made a systematic review on LLMs in software engineering, where they emphasize that this is a distinct research field, capable of a lot more than static traditional models [22].

Turning to the research questions. After covering the main objective, it is now important to understand which questions does the thesis aim to answer. They are listed below:

1. What will be the difference in terms of the performance between the fine-tuned and original Llama 3 & Qwen 2.5, compared to the traditional and neural baselines, like Random Forest (RF), Naive Bayes (NB), Support Vector Machines (SVM), Logistic Regression (LR), and CodeBERT in bug severity classification?
2. Can the research achieve a significant boost in the classification accuracy with the use of Low-Rank Adaptation (LoRA) compared to zero-shot prompting methods?
3. Are the LLMs able to handle the noise inside of the dataset, such as stack traces and code snippets, and maintain the effectiveness in identifying critical system failures?
4. How does splitting the classification and summarization tasks impact the model's accuracy and output latency during a single ingestion and batch evaluation?

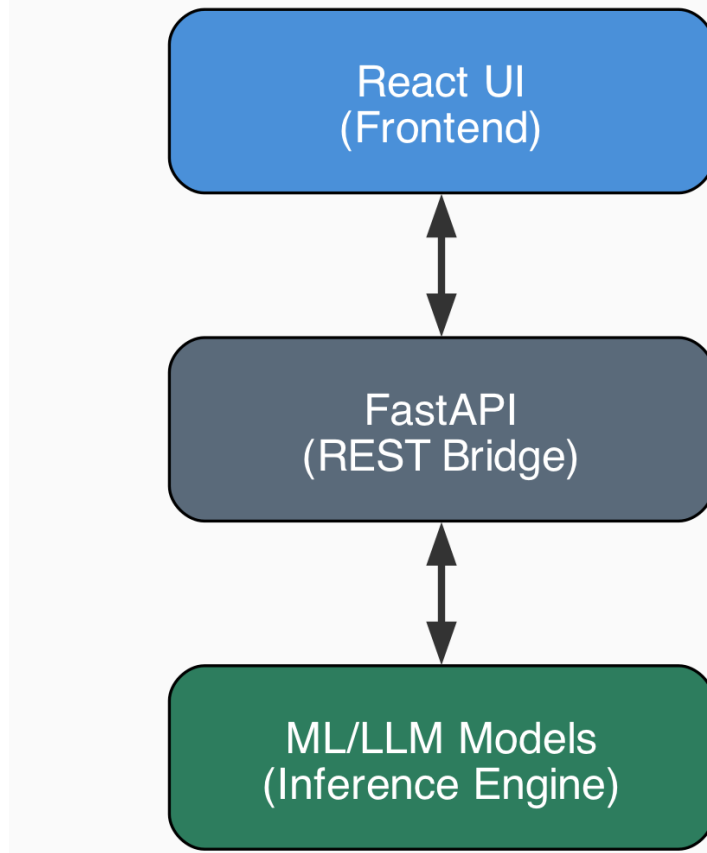


Figure 1-3: System Architecture at the High Level.

1.4 The LLM Approach

In order to tackle the disadvantages of the manual bug classification and traditional models this research proposes a smart framework. It is built on the engine of two models: Llama 3 8B & Qwen 2.5 7B. Llama and Qwen models themselves are able to understand the technical language and provide the reasoning, whereas the traditional models are very simple and just treat bug reports as a bag of words. Another research proves the shift from the metadata to the semantics-based approach as a primary source [8]. The model in such an advanced scenario understands the report and interprets the intent instead of solely counting specific words.

Moving to the model choice, as previously mentioned the research utilises Llama 3 8B. The choice fell on this model based on its technical factors. The model was released on April 18, 2024, which is still relatively fresh. As stated on the official

website of Meta, the model is computationally efficient, as well as really powerful in terms of reasoning capabilities [36]. The combination of these factors allows everyone to fine-tune it on accessible hardware. It plays a huge role, as the model can be retrained regularly with new datasets, keeping up with the company's priorities and changes in the development style.

Considering the fact that the first model was chosen, it was important to compare it with an equal LLM. Such a comparison brings the justification to this research that fine-tuning delivers better results in all scenarios. Qwen 2.5 7B is a powerful model, released on September 19, 2024 by Alibaba Cloud, is an excellent choice [42]. It is considered one of the best among the light models. What makes it special is the perfect ability to work with JSON structure, exactly what is needed for the bug classification.

Next, it is important to cover the concept of instruction tuning. It is a supervised fine-tuning method that allows to train LLMs with the specific datasets. In this case, if the dataset of bug reports is applied, the LLM will learn the technical language and the terms used by the specialists filling them up. As a result of that, the model will beat the human in terms of consistency and objectivity. From time to time, a human may forget the special rules or on a bad day, he can mistakenly assign a different priority. In the case of AI, it provides a score independent of those factors. To conclude, the newly trained model not only has a general knowledge about any domain, but also has a deep understanding of the company's dialect and jargon, it can act as an assistant in the bug classification. The LLM will go through countless reports and make sure that critical issues are not hiding in the backlog.

The contribution of this thesis is significant for the field of computer science. Firstly, it contributes to the field of LLM by developing a fine-tuned model specialized on the bug severity classification. The results of the accuracy will have a massive importance. Secondly, it contributes to the field of software development, as such tools could be used there on a daily basis. The proposed solution uses Llama 3 8B & Qwen 2.5 7B architecture for analyzing bug reports. With the use of helpful techniques like LoRA and 4 bit quantization, the research can solve the huge problem

for software companies. The model aims to reach peak performance, and at the same time balance it for the accessible hardware, so that the consumer can run the application locally. The main concept here is technology for everyone, meaning that the solution should be available to all types of hardware. By targeting a big audience in software development, the study can offer a great alternative to manual human classification.

Chapter 2

Related works

2.1 Bug Triaging and Its Complexities

Diving deeply into the history of software development lifecycle (SDLC), it was a long process of trying to automate the bug triaging. The final decision was always after the technology officer and the project manager, they would confer and come up with a priority. Such an approach was suitable for the small teams with relatively rare releases. Back in the day, the sprints were measured in months and the companies were not that demanding as now. Later, with the expansion of the application sizes and lines of code, with the tight deadlines, the disadvantages of manual bug triaging became noticeable like never before. In order to confront the limitations, the first attempts to automate this process were done more than 20 years ago, in 2004 to be exact. However, today many companies still follow the manual approach and one of the most famous problems that arose as a result of that is called bug tossing.

As previously mentioned, bug tossing is still a major problem in the SDLC. Jeong et al. discuss this issue in detail, it happens when a developer receives a bug out of his expertise and it is then reassigned to another developer, leading to the long chain and a waste of time [25]. Considering that a tossing can happen frequently within a team, it increases the metric of mean time to repair (MTTR). Also, the same research made by Jeong et al. highlights that exactly large projects have a high portion of bugs undergoing many reassignments before the correct developer receives

it. Consequently, this phenomenon requires a smart solution, where the task is not only prioritized, but also delegated to the right software engineer, minimizing the negative effect of a tossing loop.

As regards the automated classification of bugs, initially the idea came in place, when the developers realized that bug reports are chaotic and it becomes impossible to analyze them. Back in 2004, Cubranic invented a method for automatic bug triaging with a use of text classification [12]. In his work, he compares a bug report with a typical document. Therefore, if it can be treated as a document, a priority can be assigned to each bug as a classification label. Cubranic came to the conclusion that we can train a model using information retrieval method. Basically, the historical data is taken from the tasks, the model processes it and learns the patterns. Then, it is able to make predictions and provide accurate results. For this reason, his work is considered to be a basis in the application of ML for bug triaging.

During the next years, the researchers tried to work and develop that foundation made by Cubranic. In 2006 Anvik et al. established a new standard in the world of bug prioritization [5]. It was about applying the traditional models like SVM and NB on treating the active developers as the main ones. The idea behind this is that treating a report as a bag of words is incorrect and the model cannot make the right predictions. Moreover, developers usually rotate, they may be replaced, that is why the model has to pay attention to the recent contributions. Addressing this issue, Anvik et al. were able to prove that traditional models are pretty good at predicting the correct developer for a specific issue. This model achieved a fair accuracy, therefore it was a useful tool for the project managers.

Nevertheless, the technologies of that time were not enough for reaching a high accuracy. The automation was feasible, but it required stronger models that understand the semantics behind the report. The traditional models heavily rely on the keywords, the whole concept is to identify the patterns and match them inside of the text. Soon, with the scaling of the systems, the code base increases significantly and the keywords search becomes inefficient. The downsides of the information retrieval methods were addressed in the next era of bug triaging. In the next sections the

research will stop closely on the evolution of ML approaches and how they solve the challenges with the semantic and structural understanding.

2.2 Difference in a Semantic Understanding of the Bug Reports

Moving towards the first research question, each model has the limitations in the semantic and structural comprehension, restricting them from achieving the best performance. The bug classification issue requires the highest standards. For this reason, it was important to optimize the algorithms for automated bug triaging. Considering the fact that the feasibility was already proven, it was time to work on the accuracy. For that purpose, the researchers applied supervised ML using the models like SVM, NB and KNN. It was in the period from 2010 to 2017. Important to mention, the main goal in this era was to improve severity classification, unlike the older models that focused on the developer allocation. This shift can be explained by the fact that as the repositories grew, software teams started using Jira extensively. Each task had to receive a prioritization label and developers had to pick between a crash in the system, and a minor UI issue to be fixed firstly.

As the shift in the goal of bug triaging was already discussed, now it is time to mention who established the baselines. Lamkanfi et al. in his research evaluated the traditional algorithms on the open-source repositories Mozilla and Eclipse [32]. As a result, the most accurate and computationally efficient model turned out to be Naive Bayes. As you may know, NB is a simple algorithm that treats each word in the text independently, it is called feature independence. Even though the feature independence is usually treated as a downside of NB, in this case it was a decent solution. Lamkanfi et al. proved that we can treat the bug report analysis as a text mining problem, meaning that the amount of data there is very large. Even today, most of the works in the automated bug triaging still use NB as a baseline, comparing its metrics with the most advanced models, like LLMs.

As previously mentioned, NB treats each word separately, meaning that San Francisco would be two distinct words, rather than a name of the city. This simple categorization was proven to be inefficient in the long run. One research made an important statement, revealing that severity cannot be treated as categorical, it is rather ordinal [46]. To explain, categorical structure can be applied when two labels are not connected, for example tree and house. However, in case with the tasks, their severity is ordinal, just like the magnitude of earthquakes. The bugs classified as trivial should be treated as less severe than minor bugs, basically a grading scale is applied to them. The problem with the standard classifiers is that they treat the classes independently and they do not see the ranking connection between them. Tian et al. in his work noted a linguistic similarity between trivial and minor tasks, same as with major and critical. It means that the tasks ranked closely in the hierarchy have many common words. Subsequently, the old approach that utilized the bag of words was inappropriate, because it did not understand the concept of severity.

Turning to the next method, the simple information retrieval method had to be removed and there was a need for a more robust solution. The scientists came up with topic modelling that allows everyone to process the bug reports and extract the underlying themes out of it. Latent Dirichlet Allocation (LDA) was introduced in the paper by Yang et al. [53]. This algorithm functioned much better, as it was extracting multi-word topics instead of some keywords. The main idea of this method is to mark definite topics more severe than others. For example, take the topics like application crash and text typo, it is obvious that there is a drastic difference in the severity. When this model works on a bug report, it understands the context and the technical subject. The engineers then could map the subjects with each level of severity. Even though such a method boosted the accuracy, at the same time it required more computational power.

Moving forward, the next negative effect that was identified is bias. It was important to work on the algorithms to remove them. The traditional models like NB struggled with the scaling process, as the performance diminished when it faced unfamiliar words. The models like SVM were inefficient in terms of computational

power, as they struggled with high dimensional data. The researchers found a better solution, this method is called ensemble learning. It is essentially a combination of individual models that creates a stronger alternative. Johnson et al. in their research showed how well ensemble learning performs and surpasses the individual classifiers [27]. The reason for such an increase is that the model takes the best features out of them. Apart from that, the research also included the validation in multiple settings. Johnson applied the technique on industrial environments, like Ericsson. As a result, it became clear that the model was ready to be working on production, aimed for commercial use.

Classical ML was rapidly advancing up until the ensemble learning. However, each method has its own limit and the ceiling was hit here. One of the papers claimed that after a round of large experimenting on the models they could not reach any improvements [7]. As the amount of data increased, the results plummeted. Any changes in the feature selection process also had a negative impact. In the summary, classical models were stagnating, not being able to surpass a certain threshold, no matter what optimization technique was applied.

The answer of the stagnation question was hidden in the limitation of feature independence. As you already know, the classical models treat each word distinctly, which is a wrong approach. The context is lost and it results in a relatively low accuracy. Another disadvantage is ignoring cases with polysemy. The same word might have multiple meanings and the model should correctly identify them, as it is important for the context. Ambiguity is a well-known problem in NLP, when the same phrase can be interpreted differently, it plays a considerable role in linguistics. Several researchers paid attention to such limitations of classical ML. There was a clear need for the model that would be able to capture the sequence of words.

To sum up, the traditional ML models reached their possible limit and their time had come to an end. Software engineers decided to move to more advanced models. It was pointless to continue using classical models, as the complexity of the bug reports grew. The standards for the models increased, as they should read the text, understanding the semantics, instead of just counting some markers. The

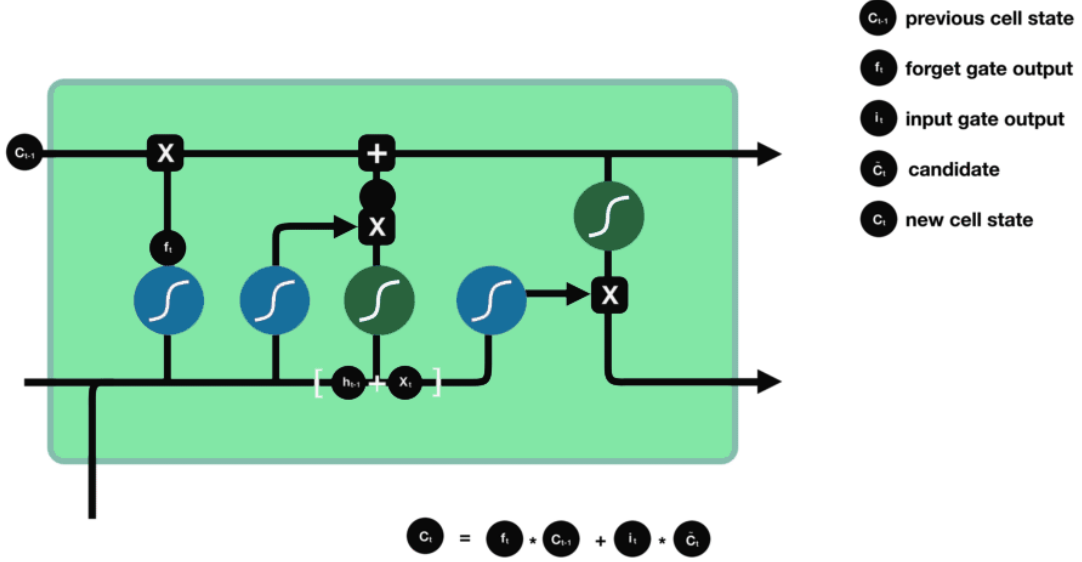


Figure 2-1: Architecture of LSTM in Text Classification. Source: [4].

advanced models that were mentioned include the techniques of deep learning and neural networks. They were better due to the fact that they captured local patterns in the text. Additionally, they could see the long-term dependencies. Gradually moving to the deep learning techniques, this shift was absolutely necessary to push over the mentioned limitations.

Classical ML could not improve the results anymore and it led to the advancements in the bug prioritization methods. As stated above, traditional models are limited by feature independence. The models were not able to catch the meaning behind the words, treating words like error and mistake as unrelated. The next era of ML granted the world with the adoption of deep learning techniques. From 2018 up until 2022 everyone could notice a rapid development, where developers started using dense word embeddings in a combination with neural architectures for the NLP purposes.

We could witness progress in this field at the moment when there was a move to word embeddings. They serve as numerical and vector-based representations of words. They help the model to understand the semantic meaning. Choquette-Choo et al. applied the most common Word2Vec and GloVe embeddings for the bug triaging [11]. As a result, they absolutely outperformed the models without the word

embeddings. The difference of the embeddings with the traditional vectors is that they use continuous vectors, meaning that words with similar meaning appear next to each other. “You shall know a word by the company it keeps”, as said by John Rupert Firth. Moreover, word embeddings allow developers to make mathematical operations on them. For example, if we subtract from the vector of king the vector of man, it would result in a vector of queen. This characteristic improved the generalization of the models, as they were able to link similar descriptions together using vectors and apply the same severity level. It worked successfully, even if the model did not see a certain word before. The previously defined problems of polysemy were resolved, resulting in a better performance.

As the issue with bag of words was solved, embeddings started to be integrated everywhere. Firstly, computer scientists tried to apply them for Convolutional Neural Networks (CNNs). Their architecture is based on mimicking the human visual cortex. Initially, they were intended for computer vision, but later were used in NLP as well. A newly designed CNN model was proposed for automated bug triaging by Umer et al. [47]. The concept behind it suggests treating the bug reports as 1D images. This method brought the revolutionary finding, where everyone could use convolutional filters for the detection of local patterns. To compare, the traditional models had to scan the whole document completely, whereas the CNNs focused on extracting n-grams, looking for specific phrases. The research includes the experimentation with the positive results of CNN application, achieving top scores in F1, compared to SVMs.

Although the CNNs were good at looking for n-grams, the bug reports are more than just a list of phrases. The bug report is a story, telling everything in a sequential order, where the beginning might be as important as the ending of a document. As regards CNNs, they experienced troubles at catching the dependencies, located far from each other. For instance, you have a debugging process mentioned at the start and somewhere at the end it states that it led to the crash in the system. In order to reach the interconnection between all the parts of a bug report, Qin and Sun first applied Long Short-Term Memory (LSTM) architecture inside of the bug triaging

model [41]. LSTM is a unique type of Recurrent Neural Network (RNN), which essentially remembers the necessary information for a long period of time. In Fig. 2-1 there is a visual representation of its architecture.

Moving forward, LSTMs allow the model to be built in such a way that it skims through the text and comprehends the logic of the report. Such systems would work exceptionally at remembering negations in the text and it would lead to the successful severity mapping. For example, if taking the models that used a bag of words approach, they would classify the sentence containing “it was not hacked” as severe, just because it saw the word hacked. In contrast, LSTMs would treat it as minor by correctly extracting the word together with negation. Hence, LSTMs perform very well on the long reports, with mixed vocabulary, as it is able to save the context in the memory throughout the whole iteration.

Turning to the next problem, RNNs and LSTMs received popularity, but they had their own negative sides as well. Considering their temporary success, they eventually lost their efficiency while processing extremely long reports. Such bug reports usually contain a lot of data, like stack traces or log dumps. Basically, what was happening is that the model would forget the data throughout the process, resulting in a vanishing gradient. As a solution, there was a DeepTriage model designed by Mani et al., which contained the attention mechanism [35]. This model achieved high optimization, as it learned to economize its memory, by assigning weights to each word. For example, special words that act as triggers for severity identification, like crash or leakage, would receive higher attention score, whereas general words would be ignored. Subsequently, the model excelled not only with its accuracy, but also with interpretability.

In conclusion, deep learning models undoubtedly reached better results than their ancestral traditional ML models. However, those results came with a much more demanding hardware, where the training process would be expensive for a general user. Moreover, RNNs were allocating resources inefficiently, by processing the text sequentially. It led to unwise use of GPUs. The deep learning models still struggled with ambiguity, as they were trained on static embeddings. The industry had to move to a more reliable solution, paving the way for Transformers and LLMs. These

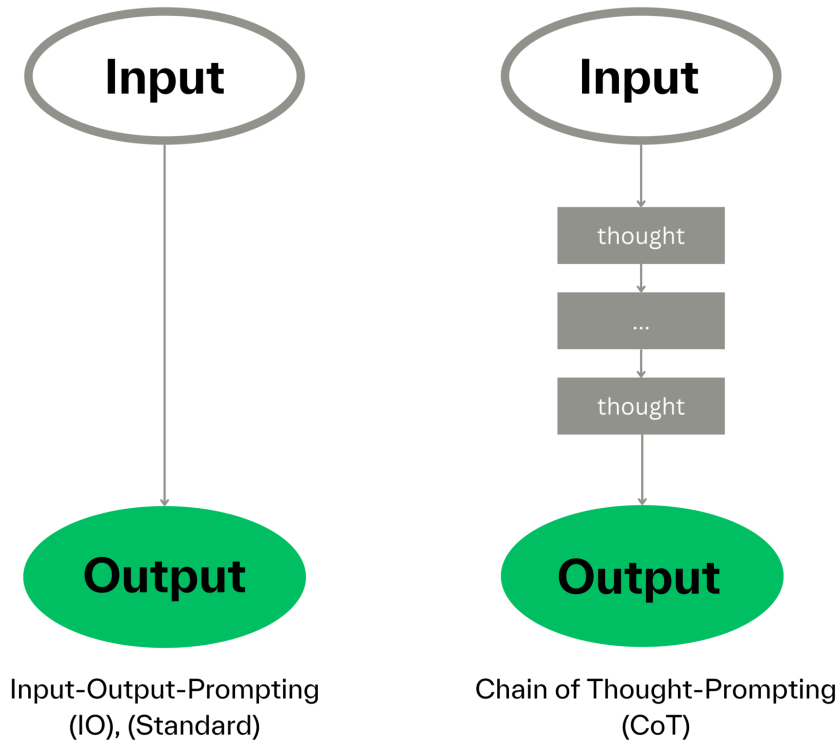


Figure 2-2: Chain of Thought Prompting in Comparison. Source: [50].

models fixed the issues with parallelization and of course, designed dynamic and context-aware embeddings.

The deep learning encoders had their own disadvantages in the semantic understanding, which led us to the key transition of generative transformer architectures. Transformers are a neural network that learns the context in the most efficient way, by tracking the relationships between sequence components. The main difference from the deep learning algorithms is that the sequence awareness was reached. To simplify, previous models of LSTMs and CNNs were just trying to map input features with specific output labels. Everyone experienced a gradual transition in software engineering from a predictive nature to generative [54]. Generative AI is able to produce new content, provide the reasoning, not just be a predictor. LLMs are much bigger than classifiers, due to great reasoning capabilities, and being able to process code with natural language at the same time.

The systematic literature review on LLMs was conducted by Hou et al. and it stated that LLMs can work and show its expertise on the tasks it did not face before [22]. Subsequently, the transformer models act differently in the bug prioritization, instead of just searching for a specific keyword, it analyzes a deeper meaning and looks for the semantic relationships. Thanks to this quality, such models reason about each bug and can competently prioritize them. This ability of reasoning is pretty similar to the cognitive process of a human. Undoubtedly, the LLMs do not replicate the human sense, but they provide a comprehensive step by step thinking. This process is depicted in Fig. 2-2. Meanwhile, the traditional models simply look for patterns, without explaining its choice in the end.

2.3 Data Noise and Class Imbalance Challenges

The evolution from traditional ML up until deep learning demonstrated an incredible progress in terms of pattern recognition. At the same time, the quality of the data played a significant role as well. A state-of-the-art model could be created, which can identify any pattern in the text, but that algorithm would not succeed without clean data. Unfortunately, bug reports are naturally unstructured and contain a lot of noisy data. Inside of the report you can find a whole mix of vocabulary intersected with the code snippets and logs. It results in a difficult process of separating keywords from the noise. Goyal and Sardana conducted a survey and identified that classical information retrieval methods cannot properly separate the misleading information from the actual severity triggers [17]. It appears to be one of the most significant problems at the training process, directly linked with the third research question of this thesis.

One of the main reasons why information retrieval fails is the huge concentration of unconnected lines of code and log data. The bug reports usually consist of thousands of such lines, meanwhile the important ones are less than ten. The model simply cannot assign each word the same weight, as it would result in waste of resources and low accuracy. Shivaji et al. decided to use aggressive feature selection in the bug

triaging domain, as it would solve the previous issue [44]. The experiments carried out in the research showed that after the removal of the unnecessary data and filtering out the tokens with numbers, the peak accuracy could be reached. The research highlighted that using only 10% of the most discriminative tokens as training data is much more efficient than trying to feed the model with the whole dataset.

Furthermore, the less compact the data is, the better the results are, as stated in the previous paragraph. Xuan et al. managed to design specific methods for data reduction, which allowed the models to prioritize the tasks correctly [52]. The paper mentions how noisy the bug reports are and that their size only keeps growing. Therefore, they struggle to keep the training time in the adequate range. In order to address this issue, the duplicate data has to be removed, as well as the noise that consists of unrelated information. By simply applying these techniques, the model can be trained 50% faster, as well as keeping our accuracy scores. Consequently, preprocessing is an integral part of bug triaging model training.

As regards the next challenge, the class imbalance can be faced. As you already know, the severity levels vary and there are at least four levels of them. Nobody assures an equal distribution of major and minor bugs. In fact, critical issues appear less frequently than the minor ones. It is a common case, when there is a dataset with 10:1 ratio in terms of minor and critical tasks. The underlying problem is that the model, which is trained on such a dataset, finds an easy way of achieving high accuracy. It simply throws a minor tag to any bug, achieving an accuracy close to 90%. This way, the model becomes biased, which is a negative outcome.

Interestingly, Xuan et al. carried out an experiment, trying to handle the class imbalance using two strategies [51]. Firstly, they tried to remove the minor bugs, equalizing the number of rows for each label. This process is called undersampling. Secondly, they applied oversampling, as you may already guess, artificially increasing the number of critical rows. The outcome was impressive, as the undersampling approach won. The model was able to learn the clear difference between minor and critical tasks, not being biased by noisy data in the minor reports. Nevertheless, both of these approaches are just temporary and there is a risk of losing a valuable context.

The strongest solution would be a model that can read the text, no matter how much noisy data is inside of the report.

Talking from experience, data reduction was one of the main methods to handle data noise inside of the dataset, whereas LLMs solve this issue in a completely different manner. As discussed earlier, the noise inside of the data remains a complex challenge for the models. The ability of LLMs to reason through the semantics solves this matter. Before, software developers used to rely on classifiers that needed an accurate preprocessing, together with aggressive feature selection. The reason for this action is obviously the mixed nature of the bug reports, containing lots of stack traces. Later, there was a move to the models that plug in static word embeddings, assigning vectors to each term [11]. Even though the embeddings were quite useful, by improving the discovery of synonyms, yet they often failed with multiple contexts. For example, the word exception can appear in both positive and negative cases, requiring a dynamic vector identification. The vector of the word exception would be the same, which is a limitation.

Turning to LLMs, they are able to get around this constraint with a use of context aware embeddings. What strikes the most, LLMs can extract the most meaningful information from the countless logs and summarize them together [31]. The model performs wisely on avoiding the irrelevant information and focusing solely on the important features. This ability is possible thanks to the attention mechanisms. Moreover, if the model can personally prioritize the correct data, there is no need for the careful cleaning process. LLMs excel at seeing the variance between signal and noise during the inference process.

2.4 Domain Adaptation with a Fine-Tuning Method

Even though LLMs achieved great results at handling relatively light topics and filtering noise, they were a long way from solving complex threads and being used in corporate environments. They are trained on massive datasets that contain the data about every topic. In most of the cases, companies want a model that succeeds in a

specific domain. For example, a medical startup wants the AI model to know all the diseases in detail, whereas the knowledge about politics is redundant. A default LLM tries to predict the upcoming word based on the context and probability. However, for professional use there is a need for a model that provides an output in a specific way, adjusting to the standards of a project. With this in mind, the instruction tuning technique becomes necessary to develop a model that solves unique tasks, like bug classification. This chapter is connected to the second research question, related to the comparison of zero-shot and fine-tuned methods.

Kiashemshaki et al. researched the effect of instruction tuning on LLMs in the bug triaging domain [29]. The outcome showed that the models simply trained on a dataset are less effective than instruction tuned models. The reason lies in the prompt engineering, as the model learns to align its expertise with user intent. The experiments made in research proved that models with a defined task, like bug classification, delivered better performance and higher F1 scores than if they had a simple prompt to describe the bug. This hypothesis was proven by the Llama 3 model, which was designed by Meta [36]. Its architecture heavily relies on the instruction following, meaning that structured prompts are the best idea for Llama. The same happens with Qwen 2.5, which is considered the best choice for working with structured JSON output.

Moving forward, there is a dilemma in the IT community between the zero-shot and fine-tuned models. The researchers argue if the fine-tuning process is actually worth it, whether it delivers a significant boost in accuracy. Nowadays, there already exist giant models like Gemini 3 or GPT 4 that can operate in complex scenarios. Heo did a valuable comparison, demonstrating that zero-shot models fail because of their inconsistency [20]. Without fine tuning, the default model is not able to extract the domain rules of a project. As a result of that, hallucinations become common. For example, if there is a bug with the chat, where the media sending does not function. Generally, AI would think that it is a massive problem and mark it as critical. On the contrary, the project may have a low percentage of chat users and the team wants tasks with the chat to have a low priority. Only by fine tuning and setting instructions

this goal can be achieved. In Fig. 2-3 you can see a bar chart, representing the effect from a few-shot prompting in a classification task. To compare, the impact from fine tuning is even more extreme.

One of the potential solutions for zero-shot LLMs is usually a connected Retrieval Augmented Generation (RAG) framework. It allows the model to fetch the information from the Internet, comparing the universally accepted metrics and then generating a response. However, there is a drawback of latency and additional computational resources, which is not always necessary. Soudani et al. in his research tried to compare fine-tuning technique with a RAG AI framework, where they found that fine-tuning always wins in a classification task [45]. It can be explained by the fact that RAG only allows to extract the facts from the web, but it does not teach the model the internal culture of the company. Meanwhile, fine-tuning specifically instructs the model to prioritize certain features that affect the accuracy. All things considered, this thesis correctly gives a preference to fine tuning a smaller and efficient Llama 3 8B & Qwen 2.5 7B models, instead of trying to integrate a zero-shot computationally expensive LLM.

2.5 Summary and Research Gap

To conclude, natural language processing evolved immensely over the last three decades. As shown in Fig. 2-4, people were able to witness three eras, where deep learning served as a bridge between the traditional models and generative AI. Traditional models, first applied in 2004 on bug classification, clearly had their downsides and hit the ceiling. The bag of words approach was not efficient, due to this, in 2017 there was a shift towards RNNs and CNNs. Without a doubt, those models were more accurate, but they required a lot of computation and experienced problems with long-distance dependencies. Finally, in 2022 with the emergence of ChatGPT we arrived in the generative era. Boran et al. mentioned that bug triaging models started to focus on understanding semantics, rather than metadata, trying to interpret the intent [8].

As generative AI is still a developing field and there are new discoveries almost

Contact Classification with Zero and Few Shot Prompting

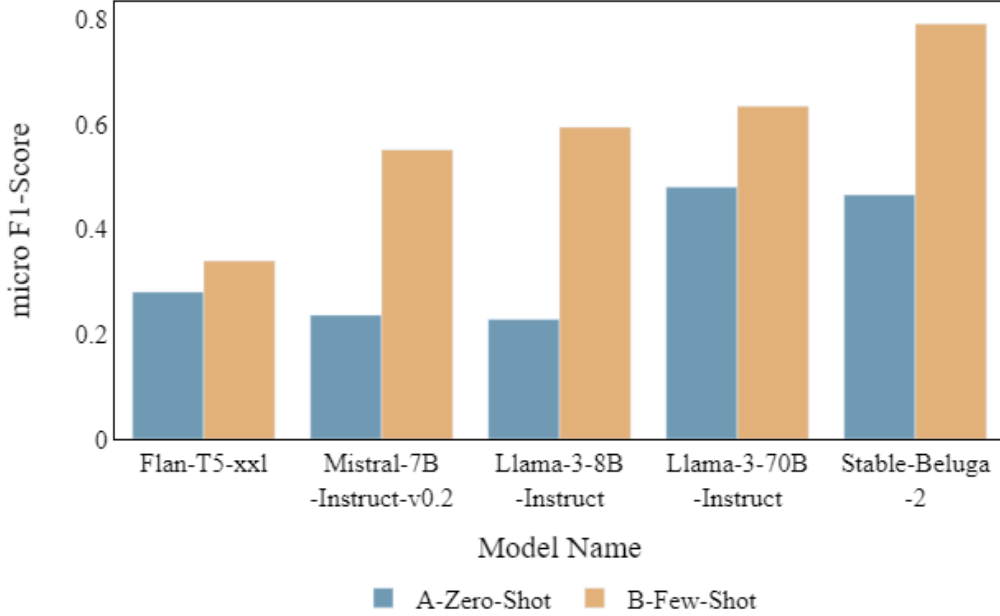


Figure 2-3: Comparison of Zero-shot and Few-shot Models. Source: [19].

every week, there is a significant research gap in how accessible and efficient LLMs are. Currently, most of the research relies on the powerful hardware, requiring not a single GPU, but a whole data center. The scientists work on the cloud servers provided by big corporations like Google and Microsoft. The regular startup does not have the graphical resources to develop their own model. Moreover, big corporations do not want to use external APIs and store their sensitive data in the cloud, addressing their privacy concerns. Sartaj et al. makes a prediction about the future of the generative era, saying that the best models will incorporate the reasoning capabilities together with efficient search based techniques [43]. However, the studies lack information on adapting open-source LLMs for daily usage, with great accuracy and on accessible hardware.

Therefore, the aim of this thesis is to bridge the gap by applying LoRA and quantization techniques, in order to achieve decent results. As demonstrated in Fig. 2-5, LoRA is an economic way to fine tune a LLM, where only 1% of the parameters are affected [23]. Dettmers et al. invented an efficient way of quantizing the model

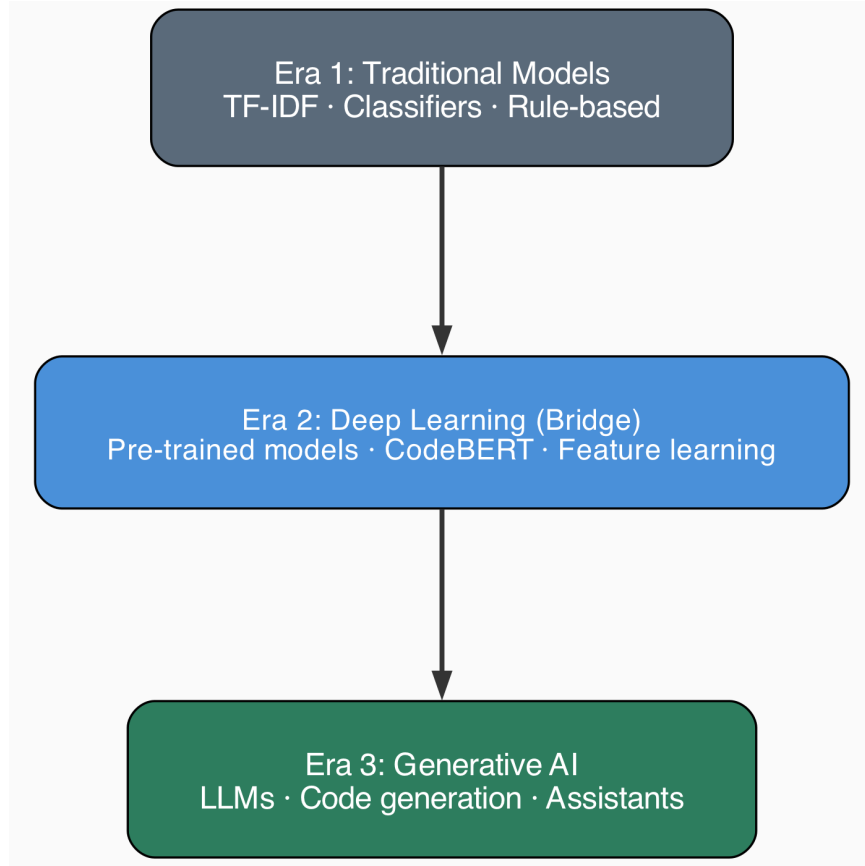


Figure 2-4: Evolution of the ML Models in Bug Classification.

to 4 bit architecture, achieving an optimized level for running on any hardware [14]. Essentially QLoRA is about compressing the model parameters, so that it could be installed on a consumer level computer. This thesis combines both of these methods with a goal of identifying if a quantized and fine-tuned model can achieve remarkable results. The conditions include a local environment, consisting of a single GPU, while guaranteeing privacy compared to the cloud solutions.

In addition, this research keeps in mind possible methodological flaws of previous studies. The majority of the work extensively focused on accuracy, instead of the objective validation with the use of statistics. In order to comply with the universal standards, this thesis considers the famous statistical framework created by Arcuri and Briand [6]. They proposed the Wilcoxon Signed-Rank Test, which plays an important role in validating whether the performance boost can be justified statistically. Also, the guidelines of Wohlin et al. were considered, resulting in a mitigation of va-

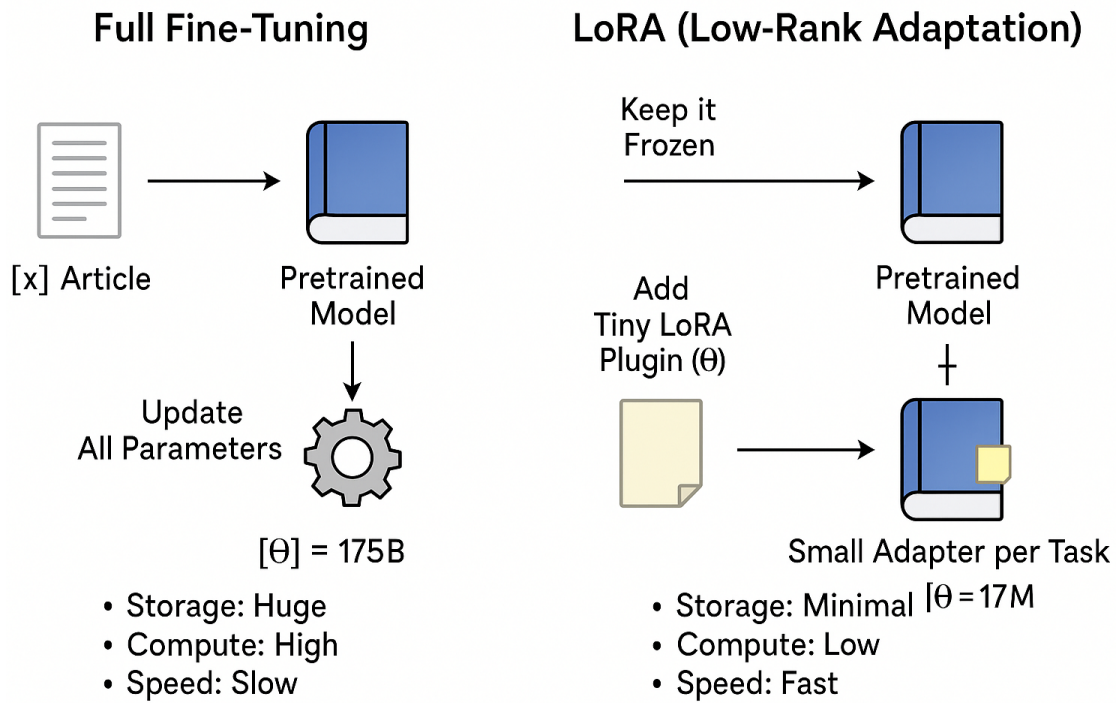


Figure 2-5: Low-Rank Adaptation of LLMs in a Diagram. Source: [37].

lidity threats [48]. This thesis ensures that all of the findings are true and can be applied in the industrial context.

Chapter 3

Methodology

3.1 Introduction

To kick off the methodology, this chapter is specifically dedicated to describing the architecture and implementation of the system. The system consists of a dual pipeline, focusing on the comparison of traditional ML baselines and LLMs. The ultimate goal is to evaluate the efficiency and accuracy of both approaches. Additionally, the comparison between default and fine-tuned LLMs plays a key role in the experiment.

The computational efficiency and semantic understanding are two of the key factors that differentiate the ML baselines with LLMs. In the case of traditional classifiers, they can execute the training and testing on a big dataset in a short amount of time, due to the heavy reliance on statistical feature extraction. However, generative LLMs have a way better contextual awareness that leads to better scores. Meanwhile, the constraints that arise from LLM implementation are hardware requirements and a long training time. In order to understand the details of both methods, this part of the thesis covers everything from the dataset preprocessing until the efficient fine-tuning.

Moreover, the objective of this research is to prove the implemented solution in practice with the use of a web application. The frontend part written in React is connected with a backend through the FastAPI, which demonstrates that the project is ready to serve as a decision support tool. The product itself could be used by software teams, as the system allows a dynamic switch between the models and a

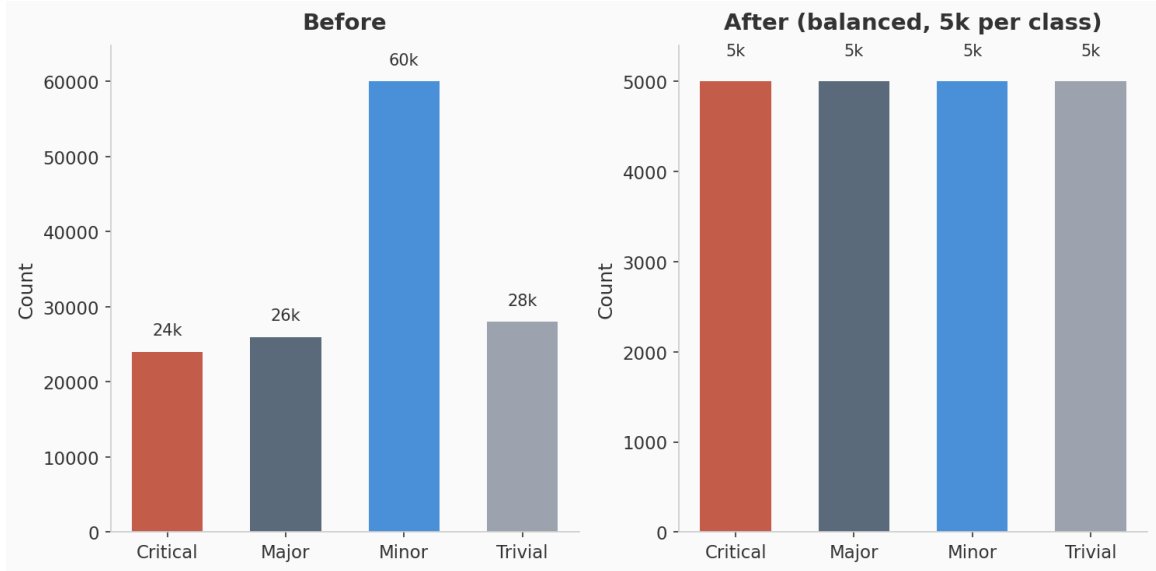


Figure 3-1: Solution of a Data Imbalance Problem.

separate tab for batch evaluation. In addition, the result of the bug analysis, done by LLM, could be transferred into Jira, which shows a direct value in modern software engineering.

3.2 Dataset Attributes and Preprocessing

The dataset selection is an important part of the methodology, as the results directly depend on it. It is without any doubt a foundation of the system, the knowledge used to train the models. It was decided to use Hugging Face repository `AndressaStefany/bug-reports`, which contains the dataset from open-source bug tracking systems like Mozilla or Eclipse, as they are a standard for the industry. One of the main advantages is that the reports have a large base of users, basically consisting of all team members. Moreover, some of the tasks came from the user feedback, that is why it is diverse and has a variable language. With that in mind, the dataset has a lot of noise that can hide the semantic meaning. The preprocessing step was identified as undoubtedly required to clean the data and prepare it for the training of the models. Firstly, the rows with missing severity were deleted, as they did not bring any value. Secondly, the tasks that contained enhancement label were deleted,

meaning that they were not bugs, rather functionality improvement. Thirdly, to make the dataset more predictable, the regular expressions were used, which remove the erratic elements. As a result, the final dataset contained only two columns of bug description and severity level.

- **Original raw dataset size:** 421,006 samples (100%)
- **Rows removed due to no severity label:** 282,916 samples (67.2%)
- **Rows removed due to enhancement label:** 54,309 samples (12.9%)
- **Final dataset size prior to balancing:** 83,781 samples (19.9%)

Even though the dataset was already cleaned from the noise, another challenge popped up. It was important to address the class imbalance, which is a common issue with such datasets. The problem lies in the distribution of minor or trivial bugs compared to the major or critical ones. As you can see in Fig. 3-1, it was close to 70/30 ratio and it is an extreme level of disparity. In ML, there is a famous problem called accuracy paradox, when the model simply predicts the majority class. Using this lazy way the model is able to achieve high accuracy, but in fact it does not have any success in detecting real bugs.

The goal was clearly to let the model learn rare severity types and to achieve that, the model had an introduced threshold. In short, the bugs with minor labels were capped to 50 thousand samples, allowing the ratio to improve. Apart from that, using the `pandas.sample (random_state=42)` method the dataset could be artificially rebalanced. Such a random undersampling method was also used in the research tackling the curse of imbalance datasets, proving the efficiency and validity of this technique [33]. Finally, each severity label was balanced in the number of samples. The distribution was equally 25% on each class. The total number of samples is equal to 20,000, whereas each class contains 5000 of them.

Important to mention, data leakage had to be prevented at the initial stage. For that cause, the dataset was split with an 80/20 train/test distribution. Apart from that, there is a random undersampling technique, which was applied to the dataset.

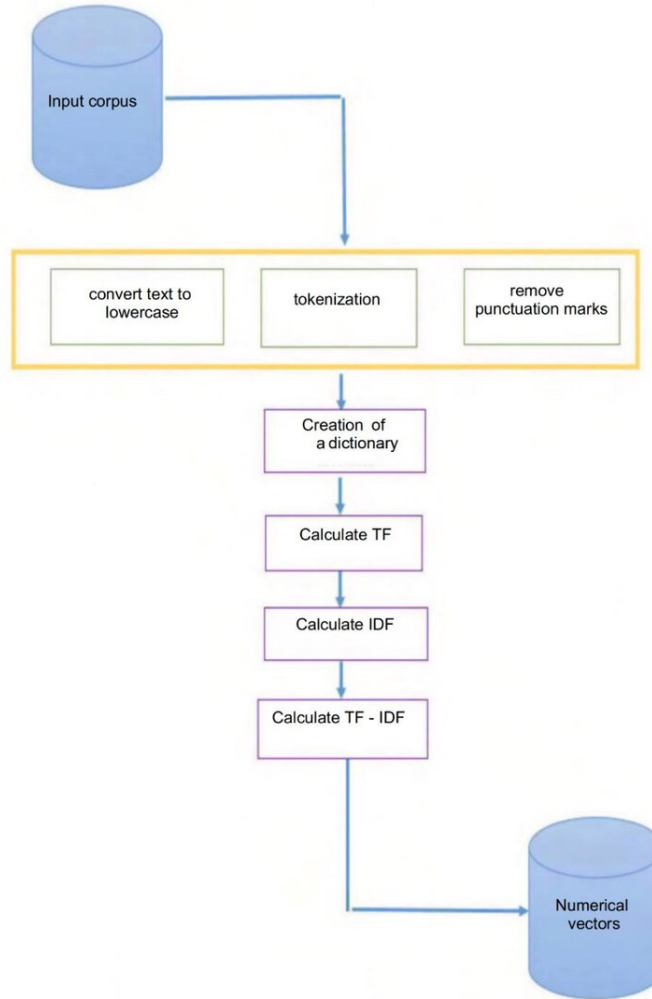


Figure 3-2: The process of TF-IDF Text Vectorization. Source: [34].

The final result was extracted in a csv format, named as cleaned.csv. In terms of the imbalanced datasets, they pose a significant problem for the F1 performance. It was studied in detail by Kim and Park, which proved that skewed classes have a negative influence on the model performance [30]. Their work states that models with binary or multiple classes struggle with the imbalanced data, leading to the decreased mean calculation. As a result of that, it is necessary to prevent the majority class from overloading the dataset. Thus, this thesis wisely applies the cleaning techniques, maximizing the model performance and creating a fair and unbiased comparison.

3.3 Pipeline of Baseline Models

Moving forward, as the dataset is now cleaned from the noise and balanced in terms of the classes, the thesis had to focus on a strong control group. As the research needed to evaluate the positive effects of generative AI on bug classification, the baselines had to be clearly established, the minimal results that it should overcome. It is widely known that traditional models in ML are not able to process the natural language, they require the numerical format. In order to complete the conversion into the desired format, it was decided to use the TF-IDF vectorizer, as it excels at feature extraction. TF-IDF is specifically useful when it needs to evaluate each word in terms of mathematical relevance. It means that the specific words that bring a lot of meaning get higher weights compared to the common and repetitive ones, which just serve as a grammatical connection. In the Fig. 3-2 you can see the entire process of text vectorization in TF-IDF.

To compare, nowadays there exist other alternatives, like neural embeddings that are more advanced in the context comprehension. However, the choice of TF-IDF can be justified by the fact that the baselines had to be kept mostly traditional, inspired by the first papers in this field, demonstrating a large progress until the LLM era. In the same fashion, Ammu et al. recently produced a paper, where they still use TF-IDF technique for the ML baselines [3]. That research was able to prove that this technique is still relevant, especially in the domain of software engineering. The reason for that is that TF-IDF can accurately extract rare terms, like error codes or environment variables. In contrast, neural embeddings may skip them, while trying to create a dense vector space.

Turning to the baseline model selection, it was important to establish the standard and choose different types of algorithms, ensuring that every case was considered. Before that, the study was able to convert the text into vectorized matrices, which allowed it to successfully prepare the preliminary data. As regards the algorithms, they include models of deterministic, probabilistic and tree-based approaches. Considering that, the evaluation is objective and does not rely on a comprehensive architecture.

To begin, the first model that was added was logistic regression, which excels at the work with sparse matrices. Secondly, the SVM algorithm allowed the thesis to apply it for hyperplane separation. Next, one of the oldest algorithms, multinomial NB is the fastest model of all, providing the selection with a probabilistic standard. In terms of tree-based methods, there were RF and XGBoost that work efficiently with non-linear relationships. In addition, the voting mechanism was applied to extract the best prediction features from all individual models, allowing the voting ensemble to detect the severity with the highest accuracy.

Important to mention, later it was realized that there is a gap in terms of the transition from traditional models to LLMs. The neural models had to be taken into account, as they are the precedents of generative models. This study introduces CodeBERT among the baselines. This BERT encoder excels at understanding natural language and surpasses TF-IDF at following structural texts. CodeBERT contains 110 million parameters and it was further fine-tuned on 16k samples, boosting the accuracy to the maximum. This way, the thesis considers all possible baselines, starting from simple ones and ending up with deep neural encoders. Subsequently, the evaluation is objective and unbiased before we move to Llama 3 and Qwen 2.5.

As the list of baseline models was identified, it was time to move to the implementation part. The research had to ensure that all of the models are implemented, trained and tested at a high level, ensuring the software reliability. For that purpose, the famous Python library called scikit learn was used, which is an academic standard and peer-reviewed by the researchers [39]. Overall, the models from the library were not altered in any significant way, the hyperparameters were left unchanged. Important to mention, the minor changes were applied to ensure the methodological integrity. One of them is the parameter of `random_state = 42` that is aimed for strict reproducibility. Moreover, another change that was applied was the `max_iter` parameter, which had to be increased for linear models. The initial goal of this operation was to ensure that the optimization process is not terminated early, basically ensuring the mathematical convergence on the dataset with vectorized high-dimensional matrices.

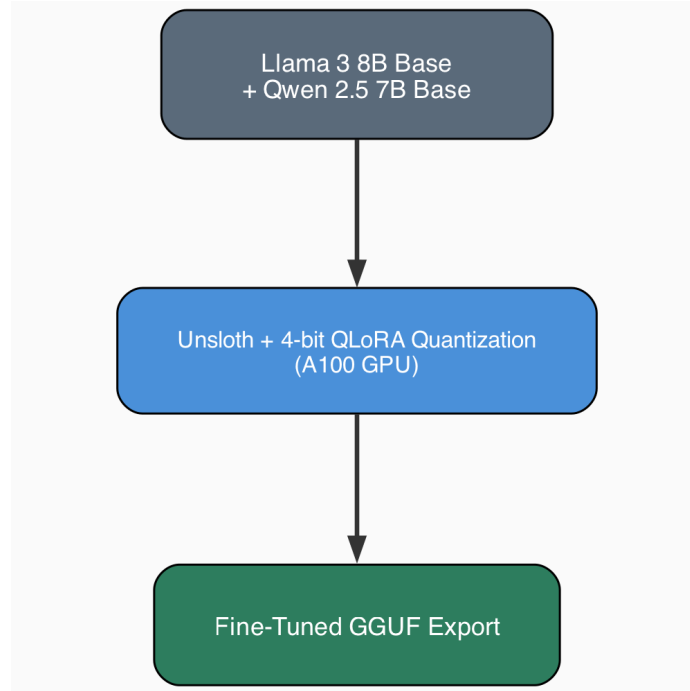


Figure 3-3: The Stack for LLM Fine Tuning.

3.4 Pipeline of LLMs

Although the traditional ML baselines and the CodeBERT encoder have a decent performance at an individual task of severity classification, they are strictly limited by the functionality in terms of the contextual representation. Meanwhile, generative AI bridges this gap and allows to analyze sophisticated defect reports. In this case, the open-source solutions were utilized, Llama 3 8B & Qwen 2.5 7B models. The goal was to seamlessly integrate the LLM into this project, for that purpose the study relied on the transformers library from Hugging Face, which is considered to be academically accepted for NLP architectures [49]. Finally, as this research focuses on improving the performance of the standard LLM, the pipeline was divided into two phases that includes the inference of zero-shot and fine-tuned models.

To cover the LLM comparison shortly, it was found to be necessary to weigh up two state-of-the-art models. The reason behind it is to see how efficiently fine-tuning works in both of the cases. Also, Llama 3 8B and Qwen 2.5 7B take part in the project not only due to the best benchmark results, but also thanks to the optimal

parameter sizes. Thanks to that, the thesis was able to fine-tune the models using 4-bit quantization and adhering to the strict VRAM limits (40 GB). Moreover, Qwen is considered a highly technical model, that is why it is fascinating to see whether it will outperform a globally recognized Llama in bug classification. In addition, the hyperparameters and fine-tuning techniques are identical, implemented with the use of LoRA. Subsequently, it allows the research to see the exact percentage gained from the fine-tuning. There is a move from a single lucky case to the trend and justification. The expectation from the fine-tuning in this study, prior to the training, was a significant improvement in the F1 score, as well as the decrease in inference latency. This happens because the model becomes optimized for the specific task of severity classification and it can process it quicker. As the training was applied on two models, it means that this approach is universally acceptable and can be replicated in different environments, not strictly depending on the architecture.

Initially, as the study evaluated the performance of a fine-tuned LLM, it would be silly to skip a zero-shot inference. It was important to compare the baselines of traditional models and LLMs, in order to understand the expectations and set the goals based on them. In this case, as the Llama 3 8B & Qwen 2.5 7B were used, the proposed system had the opportunity to connect them using the openly available Ollama API. Using that application, anyone can have a smooth user experience by texting with a chosen model and having a low latency. Additionally, it is well-known that base LLMs generate a lot of text instead of the concise answer. Considering this fact, a structured prompt was applied as an example for the model to keep the following format.

To go into detail, the prompt required the LLM to answer with just one word, containing a severity label like major or minor. There were four available options and the model just had to choose one of them, outputting it as a reply. Zhao et al. researched the prompt engineering and the correct way to utilize the zero-shot LLM, instructing it to give a desired output without any training [55]. In spite of that, foundational LLMs are still unpredictable and tend to give a long answer, without following the format. Covering this case, it was decided to play it safe and write a

script that automatically identifies the chosen severity label and parses it successfully. The exact prompt can be seen under this paragraph, provided for clarity and in case of the needs for reproducibility.

```
{"instruction": "You are a Senior QA Engineer. Classify the severity of the following bug report.\n\nSeverity definitions:\n- Critical: System crash, data loss, security vulnerability.\n- Major: Significant functionality is broken.\n- Minor: Small functional issue or inconvenience.\n- Trivial: Cosmetic issue or typo.\n\nOutput ONLY a valid JSON object with a single key:\n{\"severity\": \"<label>\"},\n\n\"input\": \"Build ID: @buildId@ (Latest head source)\n\nSteps To Reproduce:\n1. Creat a new plugin.\n2. Do a debug dump of all elements in the library.\n3. The content element associated with system created \"Customer Catergories\" has a wrong uri. It is pointed to \"plugin.xml\". Futhermore, the content element was not created before. This is also ain internal regression diff.\n\n\nMore information:\",\n\n\"output\": \"{\\\"severity\\\": \\\"Critical\\\"}\"}
```

Furthermore, zero-shot LLMs can solve basic tasks and provide general knowledge, but when tackling the domain-specific issues, it becomes complicated for it to achieve a decent accuracy. Here comes the selection of a fine-tuning method. The choice fell on the LoRA method, as it was scientifically backed by Agarwal and Madhavan that showed how LLMs tuned with LoRA win against the legacy models with encoder based architectures [2]. The reason behind it is the accuracy levels wisely balanced with the computational requirements, which leads to the optimal conditions of training and satisfying results.

As the LLM tuning requires a lot of resources, it was important to execute it efficiently, considering the hardware constraints. In this case, the research did not have a powerful machine, but there was an access to the relatively common GPU of NVIDIA A100, run in the Google Colab. As you can see in Fig. 3-3, this LLM training stack included the Unsloth library and QLoRA, specifically integrated into

LoRA Rank (r)	Alpha (α)	Trainable Parameters	Est. Training Time	Macro F1 Score
r=8	16	~21.2 Million	2h 58m 45s	58.5%
r=16	32	~42.4 Million	3h 00m 10s	60.8%
r=32	64	~84.9 Million	3h 01m 12s	63.2%

Figure 3-4: Impact of LoRA hyperparameters on the LLM fine tuning

the training script. Gholami et al. studied the positive sides of QLoRA in the work with hardware, where they found out that it tends to inject small rank decomposition matrices, essentially freezing the pre-trained weights [16]. The result is incredible, as the hardware overhead is not a threat anymore. As regards the Unsloth library, it helped with the cross entropy optimization, as well as with the memory management. Eventually, this training setup would never get out of memory.

In order to choose the correct LoRA hyperparameters for LLM fine-tuning, there was a special comparison process. As you can see in the Fig. 3-4, there is a comparison between the ranks of 8, 16 and 32. The tests were executed on the training dataset, containing 16,000 samples. The difference in the performance is marginal, where $r=32$ leads with the highest macro F1 score. Moreover, the variety in the trainable parameters should be noted, as the rank of 32 requires approximately 85 million parameters. Considering the results of the experiments, the decision was to follow this setup for the fine-tuning process:

- **Rank (r):** 32
- **Alpha (α):** 16
- **Dropout:** 0.05
- **Learning Rate:** $2e-4$
- **Epochs:** 3

Moreover, QLoRA is also considered to be efficient thanks to the NF4 quantization. In case of neural networks, they usually have a normal distribution. Meanwhile,

NF4 is a type of data that compresses the memory of the model up to 75%. Gholami et al. state that it allows the model to maintain the high level of accuracy, while decreasing the VRAM requirement from 16 GB up to 6 GB [16]. Considering the stated, it is a huge saving in terms of the computational power, allowing the eco-friendly type of training.

Some people may treat the quantization sceptically, unless the real experiment is provided. In this case, for the demonstrative purposes the study utilized 16 bit fine-tuning. The LLM was loaded using Unsloth library with the native *bfloat16* instead of the 4 bit compression. The technical environment is identical, the experiment was conducted on A100 GPU using a 32 bit optimizer, called *paged_adamw_32bit*. Unexpectedly, the training process without quantization took 35 minutes less than with it. The reason behind it is the efficient backpropagation, without 4 bit to 16 bit dequantization.

Nevertheless, the GGUF model extracted from the Google Colab was too big for the local deployment. Firstly, the model weighted four times more than the quantized counterpart, accounting for 16GB. Subsequently, such a huge model could not run properly during inference testing. The processor that it was running on was Apple M1 Pro, which only has 16GB of RAM. The LLM exhausted all physical RAM and system freezes became really noticeable. Secondly, the model used BF16 tensors, which were unquantized. As a result of that, there was a mismatch with the MacOS framework. It leads to the conclusion that 4 bit QLoRA architecture is necessary, due to the boundaries that we faced. As one of the goals of the research is to deliver the application, runnable on accessible hardware, this quantization process is a must-have.

The LLM needs a specific data format in order to learn from it. The most popular way to do so is to convert the data into JSON file. The dataset should contain the columns in a structured and easy to read way. In this case, initially the thesis needed only the severity label, but it was decided to go over the basics, outputting the reasoning, summary and description of the bug. The desired result of a JSON output is shown in Fig. 3-5. This way, there can be an improvement over the baselines,

```

25 # LLM is asked to output: reasoning, summary, description, severity
26 JSON_RESPONSE_SCHEMA = {
27     "reasoning": "Brief explanation of why this severity was chosen",
28     "summary": "One-line summary of the bug",
29     "description": "Short description of the issue",
30     "severity": "Critical | Major | Minor | Trivial",
31 }
32
33 SYSTEM_MESSAGE = (
34     "You are a Senior QA Engineer. Analyze the bug report and classify its Severity. "
35     "Output ONLY a JSON object with keys: 'severity' and 'reason'. "
36     "Severity options: Critical, Major, Minor, Trivial."
37 )
38 PROMPT_TEMPLATE = (
39     "<|begin_of_text|><|start_header_id|>system<|end_header_id|>\n\n"
40     "{system_message}<|eot_id|>"
41     "<|start_header_id|>user<|end_header_id|>\n\n"
42     "{bug_report}<|eot_id|>"
43     "<|start_header_id|>assistant<|end_header_id|>\n\n"
44 )

```

Figure 3-5: Structured Prompting in a JSON Format.

meanwhile structuring the fields in the UI and transferring them right into Jira, which brings value to the potential user. Moreover, when the model has to go through the certain phases to reach an answer, it is engaged in chain of thought (CoT) processing [15]. The CoT prompting allows to train the model more efficiently, compared with the simple prompting that has a direct result.

The final steps of the fine-tuning included the successful extraction of the model from Google Colab environment. As the training epochs came to the end, the LoRA weights were merged into the base model. It allowed to export the fine-tuned model using the GGUF file extension. Then, the model was saved on Google Drive and it could be downloaded directly on the machine. As a result of that, the model was successfully exported and run on this CPU architecture. In the end, the GGUF file was connected to the project using the FastAPI on the backend, effectively running the fine-tuned LLM in a local environment.

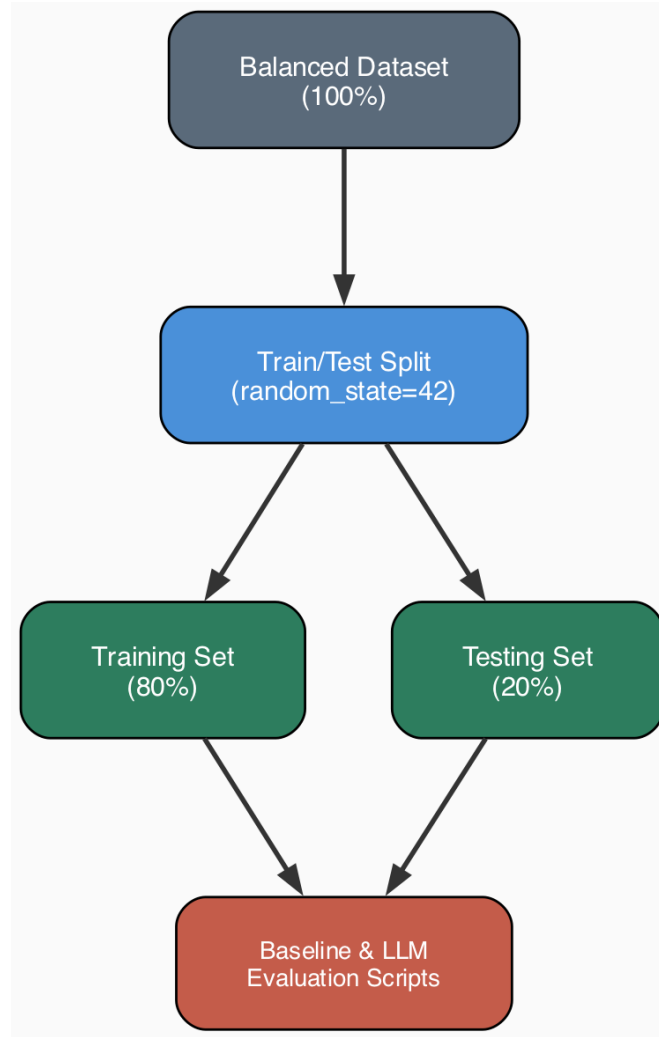


Figure 3-6: Training and Testing Data Split.

3.5 Evaluation Framework

The next phase of the experiment is evaluation. It happens when both of the pipelines are ready for training. At this stage, it was important to ensure that the data would not be leaked during the testing phase, meaning that there is a strict split of the training and testing data. This approach guarantees the academic validity of the experiment. In this case, the thesis made sure that the samples during the testing were not leaked before. For that purpose, an 80/20 split of training and testing samples was established, as can be seen in Fig. 3-6. Apart from that, the random generator was introduced, `random_state = 42`, which mixes the samples between each other

inside of the evaluation script. The following logic perfectly assures that baseline and LLM models were tested in the same data environment, but without any information leakage. The number of samples during the training of both baseline models and LLMs was 16,000. Meanwhile, the testing of baselines was executed on 4000 samples and on 2500 for LLMs, due to the fact that local LLMs have high computational latency. Nevertheless, the research made sure that LLMs were evaluated on the same dataset, using the deterministic indexing for subsampling.

Turning to the metrics, this framework included four of the most common ones in the model evaluation. They are F1, accuracy, precision and recall. The study had a combination of them, as relying on a single metric is not the right way in ML. Moreover, if the accuracy is simply taken, it will be misleading, as 80% of the dataset are minor bugs. The model can simply learn the pattern and keep using minor for all of the bugs. Multiple studies proved that accuracy is not the most significant metric and it can even confuse the scientists in the highly imbalanced datasets [10]. In our case, the model should learn to accurately predict all of the severity types, as the cost of the mistake with critical or major bugs is really high. Subsequently, the research needs a metric that comprises the accuracy of all classes, guaranteeing the harmonic mean. It was decided that macro F1 is the most important criteria. The higher the F1 score is, the more successful the model is, simple as that. It would ensure the model is able to identify all of the bugs, even the rare ones.

To conclude, this evaluation framework excels mathematically and it aligns with the modern trends in the LLM benchmarking [13]. The modern approach is about multi-dimensional benchmarking, where the study combines the scores of reasoning, fairness and robustness for the objective comparison. In this case, the study used not only the F1 score together with recall, but also the duration of the inference was analyzed. The computational latency is an important factor, as it demonstrates the practical feasibility of the solution. Nowadays, companies want fast and robust technologies integrated into the CI/CD pipelines for automation. Finally, the LLMs are evaluated on their structuring and reasoning abilities. As previously mentioned, the fine-tuned model is required to fill the summary, description and the name of the

task, in order to be exported into Jira. This operation is done by advanced semantic feature extraction and proper formatting into the JSON structure. Such a rigorous methodology is inspired by the survey made by Chen et al., where they evaluate LLMs in code generation tasks [9].

3.6 System Architecture

The initial priority of this research was on the NLP part, focusing more on the baselines and LLMs. The UI was just a supplementary part, required for the convenient testing. That is why at the start of this research a monolithic architecture was created, using Streamlit. This approach served fine as a prototype, where the developer could easily deploy the changes and test them using the browser. However, it was quickly realized that it limits the project in terms of the functionality and future scalability. The interface of Streamlit was hard to alter and it was impossible to introduce specific pages for this thesis project. The next decision was to move to a fully decoupled architecture. It means that a separate frontend UI was implemented and connected with the backend via FastAPI. For a better understanding you can refer to the diagram depicted in Fig. 3-7. The files related to the server are located in the api folder. The main file is called server.py and it contains BaseEngine interface, serving as a core model registry. Thanks to this architectural design, the system was able to have flexibility in terms of the model switching inside of the application. It means that LLMs and baselines can be swapped seamlessly by the user, without any delay or extra commands. The pivot that was taken from the initial Streamlit implementation was inspired by the decoupled infrastructure made by Jin et al. [26]. They perfectly integrated the LLMs using the APIs, building a fully autonomous system, run on so-called AI agents.

The application has a decoupled architecture on the backend during the live bug ingestion. It works simply by combining zero-shot and fine-tuned models, each serving its own purpose. The initial implementation required more computational power and extensive training in order to train the model to provide additional summary

with reasoning. It resulted in a model that does classification at the average level. Consequently, it was decided to fine-tune the LLMs solely on the classification of the bugs by priority. This way, when the live ingestion happens, it firstly uses the fine-tuned model to output the severity label. Then, the second stage begins. The zero-shot LLM receives the severity output and combines it with a bug description. It is strictly said to maintain this severity label, in order to not overwrite it. Subsequently, the zero-shot LLM finishes the sequence by summarizing the severity of the bug with a textual reasoning and summary. As a result of that, the proposed system saves a dozen tokens and time during the live classification.

As regards the frontend part, the React language was used to completely rewrite the whole functionality. Firstly, the interface was designed via Figma, using the wireframes. Then, it took some time to transfer the desired style into the application. Moreover, the interface does not only have a pleasant design, but also adaptable scenarios. For example, the user has an opportunity to upload the bugs in a textual or audio format. The application also allows uploading any textual files, like DOC or PDF. Additionally, the output page is really flexible, changing based on the selected model. In the case of the traditional models, the result is strictly a severity label with the probability percentage. In contrast, LLMs output the severity combined with the main task fields, like name and description, ready to be submitted into Jira. Such a dynamic realization adds more value and positive experience to the user that can intuitively understand the difference between the models.

Regarding the database, it was decided to combine the SQLite on the backend and LocalStorage from the frontend in order to implement a working state management. The vision is to update this configuration to a more robust approach, using Supabase. However, for the testing purposes the current version is more than enough, as it satisfies the MVP. To be exact, each request of the user is saved in a designated history tab, whereas the batch evaluation results are also recorded in the database. It allows the users to return to their results for future analysis, as well as exporting them for personal use. The current approach perfectly aligns with the vision of the European Union that initiated the GENIUS project for an integration of GenAI into

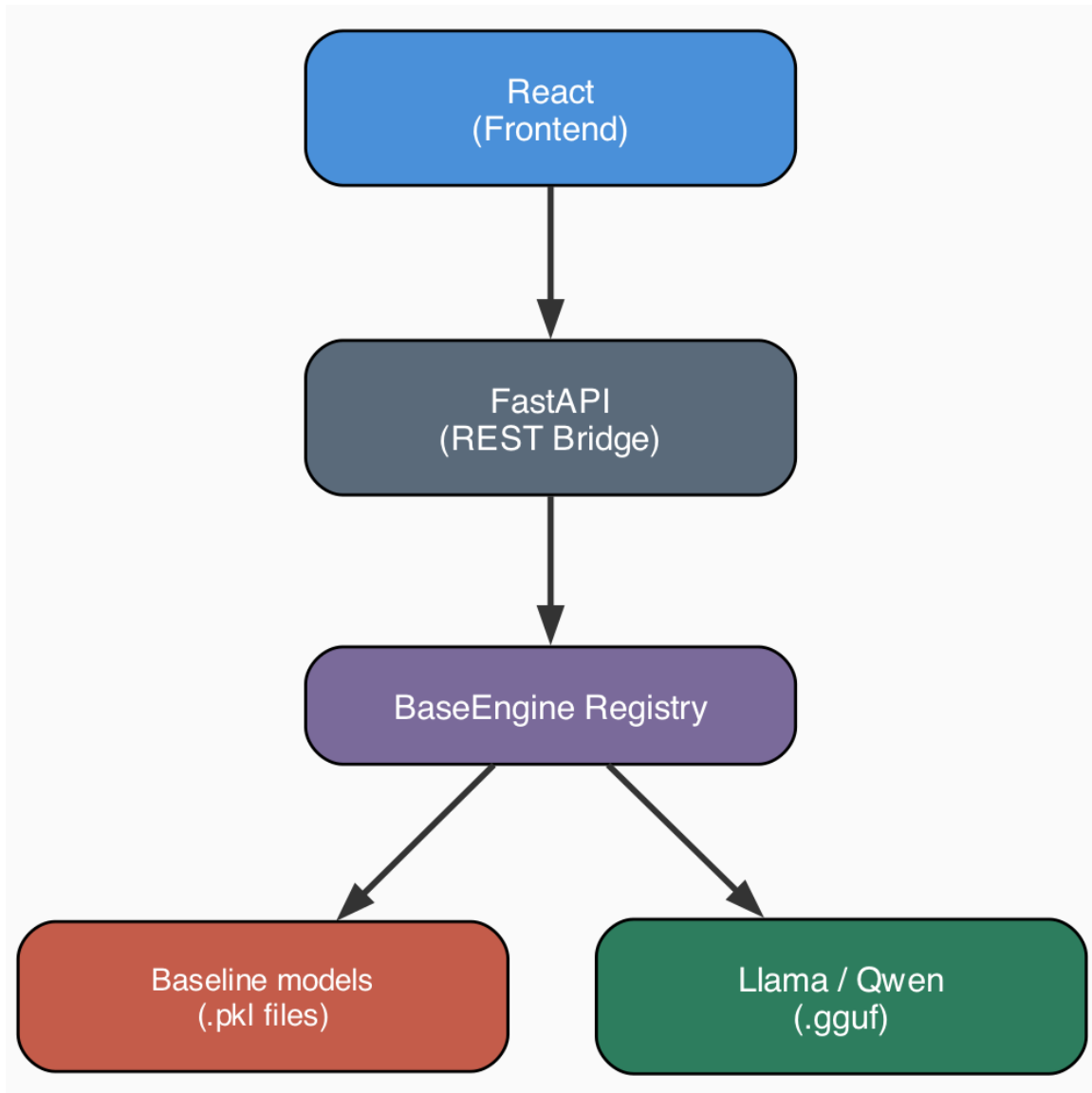


Figure 3-7: Decoupled System Design.

the usual software development workflows [18]. The objective of their project is to focus on the practical application of LLMs, rather than achieving record results in terms of the metrics. Subsequently, this thesis, without any doubt, connects the theory of NLP together with software engineering, delivering real value to the users via web application with a modern architecture. Additionally, it is worth mentioning the fact that web application and Jira integration serve as the secondary contribution to this thesis. They demonstrate practical feasibility in a good fashion, but the LLM pipeline and machine learning contribution is the main outcome of the academic work.

3.7 Reproducibility Tips

This study can be replicated by anyone, if following the specific guidelines. For the ease of use, a `config.json` was implemented at the core of the application that connects together all of the models. This file plays a significant role, as it is responsible for aligning together several phases, starting from training and ending with evaluation. Moreover, it was already mentioned that the random seed parameter of `random_state = 42` was introduced. Using this method, future researchers can easily copy these evaluation environments, where they can split and subsample the data in a similar manner.

As regards the computational architecture, in this case the research specifically divided the data into training and inference parts. The thesis started from the baseline training and evaluation, which was performed solely on the laptop, MacBook M1 Pro. Apart from that, the web application is run locally, on the laptop as well. When the fine-tuning phase was reached, there were some worries about the computational units, as the machine was not able to produce that much power. For that reason, Llama 3 & Qwen 2.5 were fine-tuned in a cloud environment, using the GPU of NVIDIA A100. Such a setup of hardware proves the accessible level for any consumer.

Tackling the problem of demanding hardware for LLM training, there is a new movement in the GenAI. This movement promotes Green AI sustainability, trying to minimize the energy consumption and carbon footprint [38]. Therefore, the choice of Unsloth library and QLoRA quantization can be justified not only by the accessible entry barrier, but also by the parameter efficient fine-tuning and accelerated training epochs. As a result of that, the study contributes to the sustainable LLM training, demonstrating that the researchers with limited resources can still achieve decent results. Additionally, it serves as an example for the corporations that waste their computational resources and drain the electricity.

Dataset Distribution for Model Evaluation			
Model Category	Training Samples	Testing Samples	Inference Hardware
Baselines	16k	4k	Local CPU (scikit-learn)
Zero-shot LLMs	N/A	2.5k	Local CPU (Ollama)
Fine-Tuned LLMs	16k	2.5k	Local CPU (llama-cpp, GGUF)

Figure 4-1: Dataset Distribution in Each Pipeline.

Chapter 4

Results

4.1 Evaluation Setup

The first step of the evaluation is incredibly crucial, as everything after that will depend on it. If there is a wrong setup, it may lead to misleading results and the whole experiment will fail. In this case, it was made sure that the evaluation environment is safe and sound for the baseline models and LLMs. The major concern was data leakage and the study prevented it with an isolated holdout set. It was already mentioned in the methodology that the research followed the 80/20 split for training and testing data. The entire dataset consists of 20,000 samples, fully balanced, containing 5,000 samples on each class. Subsequently, the training data has 16,000 samples and testing data accounts for 4,000 samples.

Turning to the evaluation of each model, traditional baselines had no problem

with the inference, as their computational latency is incredibly low. That is why the proposed system could simply use the full 4000 sample testing data during the evaluation. Meanwhile, the LLMs also could be evaluated on 4000 samples. However, the tests were executed on a local environment, having a common hardware of MacBook M1 Pro. The thesis had a purpose of proving that LLMs can be running on a consumer level hardware for such tasks. The 4000 sample test would take an enormous amount of time, for that reason it was subsampled to 2500 rows, which reaches nearly a duration of one hour for a specific LLM. Undoubtedly, the comparison remains fair, as the fixed random state was applied during the deterministic indexing. You can see the final dataset distribution in Fig. 4-1.

The bug classification models were evaluated on four metrics that are used universally in any ML domain. They are accuracy, precision, recall and F1. The reason behind such a choice is that high accuracy usually obscures the whole picture and capabilities of the model. Relying only on accuracy would not be enough, that is why this research also prioritizes the macro variants of precision, recall and F1.

Stopping on each metric, it would be great to describe their meaning and implementation. The macro-averaging approach is famous for calculating each metric independently. Then, in the end the unweighted mean is calculated. Firstly, the metric that evaluates each class on the number of correct severity predictions is macro precision. Next, there is macro recall, which focuses on true positives for a specific class. The macro F1 is the final metric, which combines together the previous two. It computes their harmonic mean as a result.

In such a task of bug classification, the model can learn an easy way and become biased. Imagine, if the model just learns how to predict certain trivial tasks and completely ignores the critical ones. The consequences are massive, because the model overfits toward one of the classes. In the real world, it would mean that the critical bugs could be missed, which have much more importance for the company, compared with trivial errors. As a result of that, there is a conclusion that macro F1 is the most significant score in this experiment. In our case, the higher F1 score is, the better the model understands the boundaries between each of four severity

Macro-Averaged Classification Metrics

$$\text{Macro Precision} = \frac{1}{K} \sum_{k=1}^K \frac{TP_k}{TP_k + FP_k}$$

$$\text{Macro Recall} = \frac{1}{K} \sum_{k=1}^K \frac{TP_k}{TP_k + FN_k}$$

$$\text{Macro F1} = \frac{1}{K} \sum_{k=1}^K \frac{2 \cdot \text{Precision}_k \cdot \text{Recall}_k}{\text{Precision}_k + \text{Recall}_k}$$

Variable definitions:

TP: True Positives (correctly predicted as class *k*)

FP: False Positives (incorrectly predicted as class *k*)

FN: False Negatives (missed instances of class *k*)

Figure 4-2: Formulas Calculating the Model Performance.

classes. As you can see in Fig. 4-2, it demonstrates the formulas how the score is calculated.

4.2 Performance of Traditional Models

The first stage of this evaluation required establishing baseline results with a use of traditional ML algorithms. For that matter, the study focused on six classifiers that use TF-IDF for text vectorization. As previously mentioned, the testing of traditional ML models was done on 4000 samples. The models that participated in the evaluation were multinomial NB, SVM, LR, RF, voting ensemble, XGBoost, and CodeBERT.

Baseline Models Performance						
Model	Samples	Accuracy	Precision	Recall	F1 Score	Duration (s)
Logistic Regression	4000	0.4935	0.4894	0.4935	0.4908	0.57
SVM	4000	0.4953	0.4882	0.4952	0.4887	0.57
Random Forest	4000	0.5012	0.4951	0.5013	0.4939	1.09
Naive Bayes	4000	0.4495	0.4722	0.4495	0.4439	0.52
XGBoost	4000	0.4950	0.4902	0.4950	0.4917	0.58
Voting Ensemble	4000	0.4978	0.4922	0.4978	0.4904	1.02
CodeBERT	4000	0.5312	0.5353	0.5312	0.5322	199.68

Figure 4-3: Baseline Model Performance.

Taking as an example a random guess, it would potentially have a 25% accuracy, as there are four equally balanced classes. In our case, all of the models perform better than that, achieving nearly a score of 0.5 in macro F1. It means that the baseline models are able to learn the signals from the bug reports. However, the TF-IDF models hit their ceiling, not being able to progress over that threshold of 50% in accuracy. Meanwhile, the CodeBERT model managed to establish the peak among baselines, reaching 53% in accuracy.

Covering each model's result, there are obvious differences regarding the prediction capabilities and the duration. Starting from the first one, there is NB as the probabilistic classifier, which underperformed in the current conditions. The model achieved 45% in accuracy and 44% in F1, though winning in the execution time. NB was able to process the whole training set in just 0.52 seconds. The reason behind it is clear and it is feature independence, where the model thinks each word is independent in the bug report. Meanwhile, the models with a tree structure turned out to be the winners by successfully capturing the non-linear relationships. RF has the highest F1 score of 49.4%, whereas XGBoost is the second with 49.2%. Even though RF was the most accurate, it had a trade-off in terms of the latency, having the worst computational time of 1.09 seconds. It is two times slower than NB and it happened due to the tree-based structure, where the model has to traverse through each tree. LR and SVM are the deterministic models, which achieved the optimal balance between the inference time and the accuracy. Their F1 scores are nearly the

same, hitting 49%, while the execution duration was matched at 0.57 seconds.

After trying out the probabilistic, deterministic and tree-based approaches, it was decided to improve the results by aggregating their predictive features. It is possible by using a voting ensemble, which combines the base models. Nevertheless, the results were not satisfactory, as it still got 50% accuracy and 49% F1 score. The computational time was also 1.02 seconds. Subsequently, the conclusion was made that the issue is not in the algorithm, rather in the TF-IDF vectorization. The models are simply not able to understand the semantic meaning behind the bug reports, as they treat them as a bag of words. This task is not a spam identification, rather a much more complex analysis of the technical and logical reports. For this reason, such a traditional approach has the ceiling effect. In the Fig. 4-3 there is a full and demonstrative comparison of the baseline models.

As regards the confusion between the classes, the traditional baselines demonstrated the expected results. They learned very effectively how to distinguish between critical and trivial bugs. The models learned the vocabulary that critical bugs mostly contain the word crash, whereas the trivial bugs have the words like typo or design. However, as pretty much expected, they could not identify the clear difference between the middle classes. The confusion in the majority of cases happens between major and minor features. In the Fig. 4-4 you can see the confusion matrix of voting ensemble, justifying this statement. The hardest challenge that appeared to be was the differentiation between the middle classes (Major and Minor). The bugs share the overlapping vocabulary, especially the ones, which are located close in the severity levels. Due to that reason, the most common mistake was misclassifying Major bugs by calling them Minor.

The previously mentioned confusion between the middle classes arises due to the TF-IDF feature that serves as the bag of words, identifying the word frequency in each report. Overall, the major and minor bugs have the similar vocabulary, varying only in the specific moments related to the application part. For example, it might be an error with the navigation button and a withdraw button. In each of the scenarios, it is an issue with the button and a continued technical explanation with

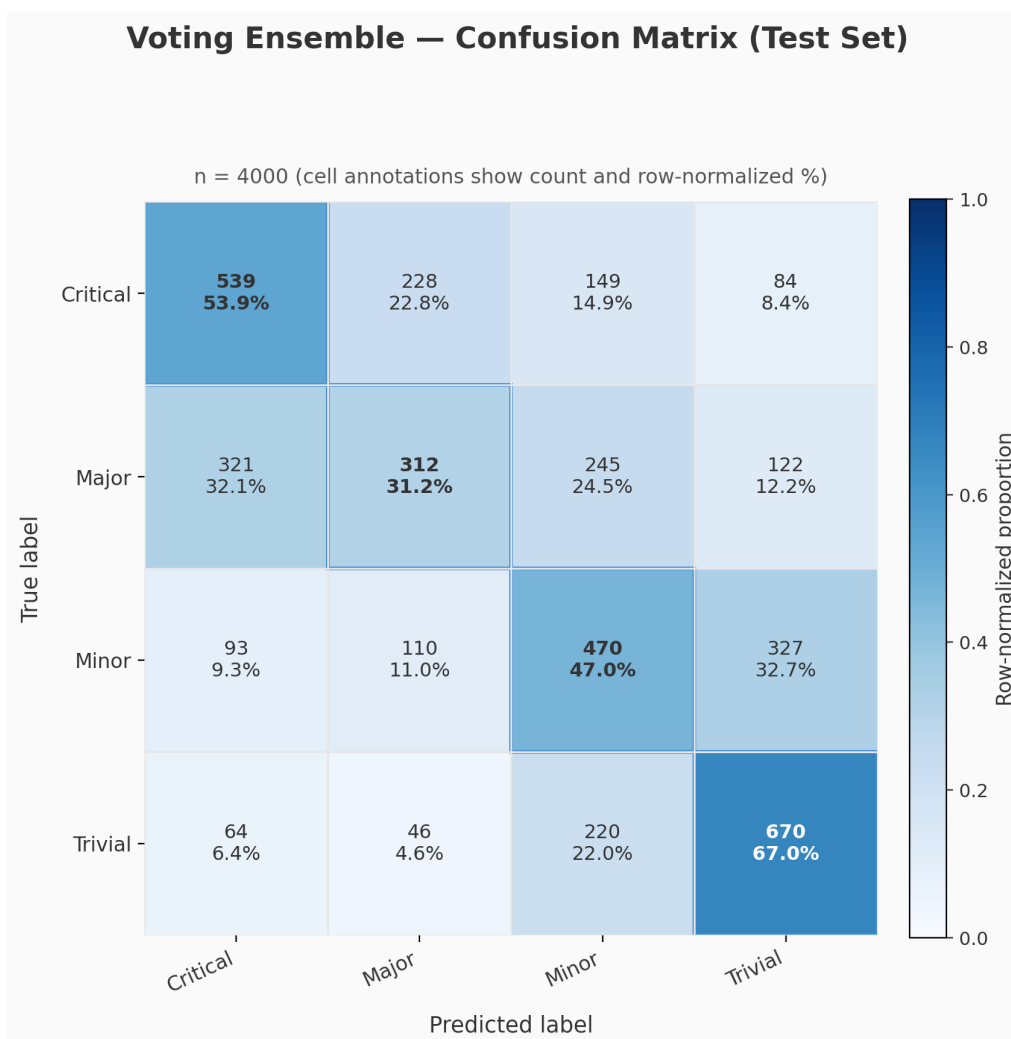


Figure 4-4: Voting Ensemble Confusion Matrix Heatmap.

a shared vocabulary. The problem is that the traditional models cannot understand the contextual intent. The withdraw button surely has much more severity for the potential users and the model should be able to understand that. Unfortunately, the TF-IDF baselines do not have that ability and their results are limited with a 50% accuracy ceiling. That leads to the next phase, where the context-aware embeddings are applied.

The next stage of the generative LLM evaluation was delayed by the important task. The neural baseline result was significant, as it could break through the traditional models' ceiling. The CodeBERT encoder managed to overcome the 50% barrier, pushing right until 53% in macro F1. Therefore, using this data the study

arrives at the point, where contextual embeddings are undoubtedly required to increase the accuracy further. Apart from that, the threshold that was introduced by CodeBERT maximizes the efforts and extends the limits for the subsequent LLMs.

4.3 Evaluation of Zero-Shot LLMs

The next phase of the baseline evaluation was the application of zero-shot LLMs. In this case, Llama 3 8B from Meta and Qwen 2.5 7B from Alibaba Cloud were connected using the Ollama engine. The task was identical to the traditional baselines, they had to identify the severity on slightly fewer samples, 2500 bugs to be exact. Considering the fact that LLMs have billions of parameters, meaning that they are computationally heavy and have much better capabilities, it was expected from them to overcome the traditional baselines with flying colors. However, the results were not that impressive, compared to the 53% F1 score introduced by the CodeBERT baseline.

Turning to the zero-shot LLM results in detail, it can be seen how the Llama model performs with 48% F1 score and 49% accuracy. At the same time, Qwen delivers negligibly better results, having 49% F1 score and 50% accuracy. Their precision is notably much higher than the traditional baselines, hitting a striking 67%. However, apart from that they perform at the same level with the TF-IDF models. Moreover, their average result cannot be justified by their inference time. Comparing the traditional models and LLMs, the difference is absolutely extreme, which you can notice in Fig. 4-5. RF had the slowest duration with roughly 1 second for 4000 samples, whereas Llama needed 57 minutes to process 2500 samples. Furthermore, Qwen even surpasses 1 hour in the execution time, taking approximately 64 minutes.

It was important to perform a qualitative analysis and debug the output made by zero-shot LLMs in order to understand why they fail against the traditional baselines. The issue lies deeper, not in the contextual representation. In fact, zero-shot LLMs are much more capable than traditional ML models, because they provide the reasoning and the summary of the bug, justifying their choice like a real human. However, in the

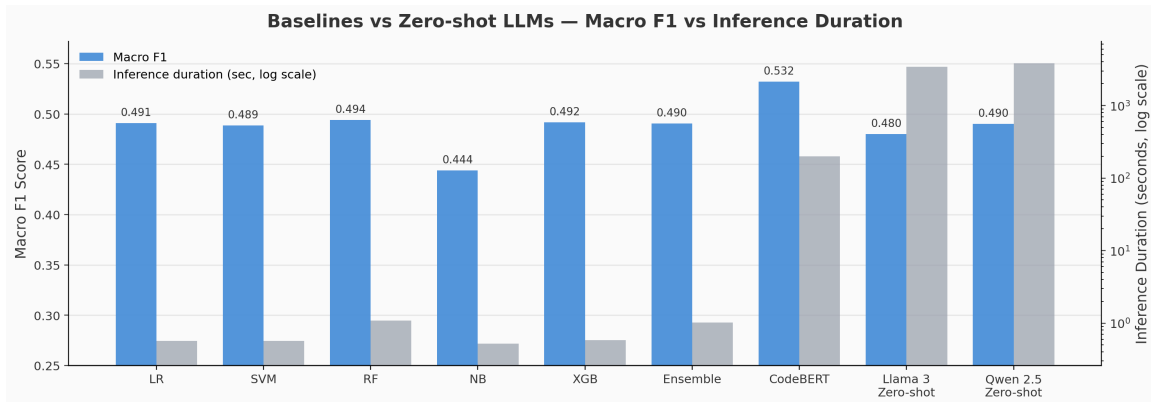


Figure 4-5: Comparison of Zero-Shot LLMs with Baselines.

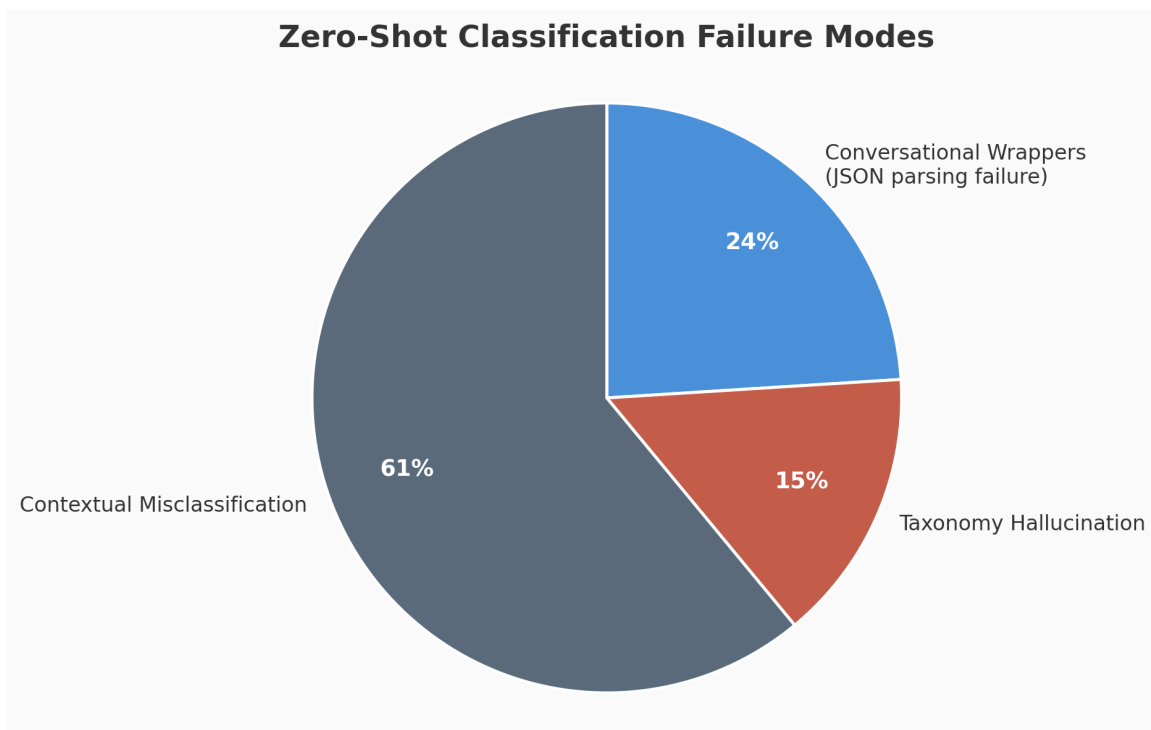


Figure 4-6: Reasons of Zero-Shot LLMs Failure.

classifying task they tend to disobey the specific command. To be exact, when asked to output only severity in a JSON format, the LLM becomes too chatty, outputting the unnecessary information, like providing additional assistance. Sometimes it breaks the JSON schema, which is required in the CI/CD pipeline for an efficient data parsing. Due to this case, the system has to record a misclassification, which ruins a certain number of predictions.

Additionally, the zero-shot LLMs struggle with hallucinations. The instruction prompt clearly mentions four classes of severity and asks the model to output one of them as a result. Due to the fact that the model was not aware of such hierarchy before, it sometimes comes up with its own tier list, adding new classes or renaming current ones. As an example, the zero-shot LLMs output blocker or medium classes in several cases. Apart from that, the LLMs without fine-tuning are impacted by the emotional tone of the bug report. For instance, if you use a dramatic description for a trivial bug, including your personal frustration and other irrelevant factors, the LLM marks this bug as major, being guided by the human's feelings.

To conclude, the downsides of the default LLMs, depicted in Fig. 4-6, are critical and they state that they cannot be applied for severity classification in the real scenario. In order to use such models in a software engineering company, there has to be an adaptation for their domain, culture and specific guidelines. The model has to learn the patterns from the previous data for being smarter and contextually aware. That leads to the conclusion that zero-shot LLMs, even with a huge computational power, have problems in the comprehension of a strict instruction and differentiation between the severity levels.

4.4 Evaluation of Fine-Tuned LLMs

Moving forward, the next stage of this research included the fine-tuning of LLMs. Considering that zero-shot LLMs failed and delivered only 49% of F1 score, equalling the traditional baselines and hanging back the CodeBERT model, that ceiling had to be overcome. In this case, LoRA technique was applied with a supervised instruction tuning on Llama 3 8B and Qwen 2.5 7B. The training set consists of 16,000 samples, the same as for the traditional baselines. The goal of fine-tuning is not to teach the models the technical language, but rather to familiarize it with severity tiers of four classes and reports structure. As an outcome, the research got the impressive metrics, outperforming the base ML models, as well as the zero-shot LLMs. It proves the hypothesis that any LLM can be adapted for a specific domain using the fine-

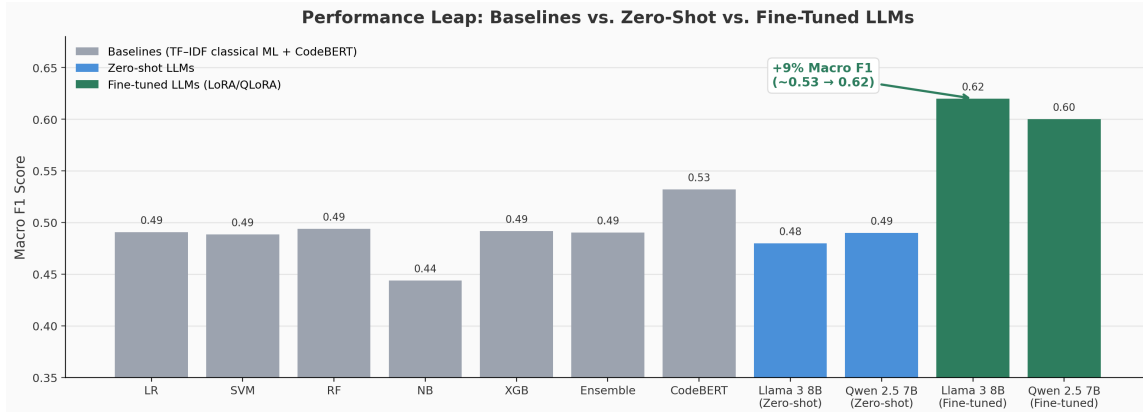


Figure 4-7: The Performance Comparison of Baselines, Zero-Shot and Fine-Tuned LLMs.

tuning technique.

Starting from the fine-tuned Llama, it demonstrated the accuracy of 62% and F1 score of 62% as well. It became a winner, improving for nearly 13% over the zero-shot LLMs in terms of the F1 score. Simultaneously, the fine-tuned Qwen had a close battle with Llama and delivered 61% accuracy coupled with 60% F1 score. It is still an impressive improvement of 11% in F1. Moreover, both of the models had a better inference time compared to zero-shot LLMs. Fine-tuned LLMs had an average execution of 55 minutes for 2500 samples, which is 2 minutes faster than zero-shot Llama and 9 minutes faster than zero-shot Qwen. Overall, the result of 62% in F1 and 13% improvement is really strong, considering how complex the bug reports are. The average software engineer would also struggle with an accurate severity classification, as it contains a lot of unfamiliar code and logs. For a detailed comparison, there is a Fig. 4-7 with an illustrative difference.

The implementation of the fine-tuning LoRA methodology for both models has proven its efficiency. The task was successfully completed for LLMs to learn the JSON structure and to output a severity label strictly. The number of the errors due to the parsing reached 0%, as the models remembered the format. Moreover, both of the models perfectly fit the classification task. In the case of Qwen 2.5, it works ideally in the programming domain, dealing with the coding and having technical expertise. As regards Llama 3, it has a decent mathematical base. The results showed that

fine-tuned Llama 3 8B outperformed Qwen 2.5 7B by 2% in F1 score. In theory, it can be due to the larger parameter size of Llama 3, which plays a huge role in the model's capabilities. Also, Llama 3 can be better in instruction following, making it more reliable for fine-tuning. Even though there is a slight difference in the accuracy, both of the models delivered satisfactory results and proved that fine-tuning works effectively for any LLM architecture.

Considering that the overall metrics of F1 are strong, the thesis still needs to provide the analysis on each class. It allows the user to understand with which classes does the model struggle. The results highly varied across severity types. As regards fine-tuned Llama 3, the highest precision was for the Trivial class, reaching 0.79. Meanwhile, precision for Critical = 0.75, Major = 0.49 and Minor = 0.62. The recall for Llama 3 is as follows: Critical = 0.49, Major = 0.77, Minor = 0.60 and Trivial = 0.64. Turning to Qwen 2.5, the Minor class has best precision, that equals 0.68. Other than that, Critical = 0.60, Major = 0.62 and Trivial = 0.60. The recall for Qwen 2.5 is as follows: Critical = 0.75, Major = 0.47, Minor = 0.37 and Trivial = 0.86. It leads us to the conclusion that Llama 3 is able to predict Critical and Trivial bugs with a high precision. However, it fails with the recall of Critical tasks. In contrast, Qwen 2.5 is great at detecting Critical issues, but fails with Minor ones. Subsequently, this trend shows that each model develops a unique ability of classification during the fine tuning process.

As you can notice, there is a striking coincidence in the accuracy, recall and F1 scores of the fine-tuned Llama 3. Each of the metrics has a 62% result. The similar pattern is seen for fine-tuned Qwen 2.5, which has 60-61% in that area. It can be explained by the fact that the models learned how to differentiate between the classes. Furthermore, the dataset undersampling worked perfectly, as the study reached an ideal balance between the classes. It affected the models' knowledge and it did not overfit towards any of the severity types.

In the majority of the cases, the datasets in software engineering are imbalanced, containing mostly minor bugs. The model learns a simple and straightforward way to predict one of the overfitting classes, achieving a top accuracy, but struggling with

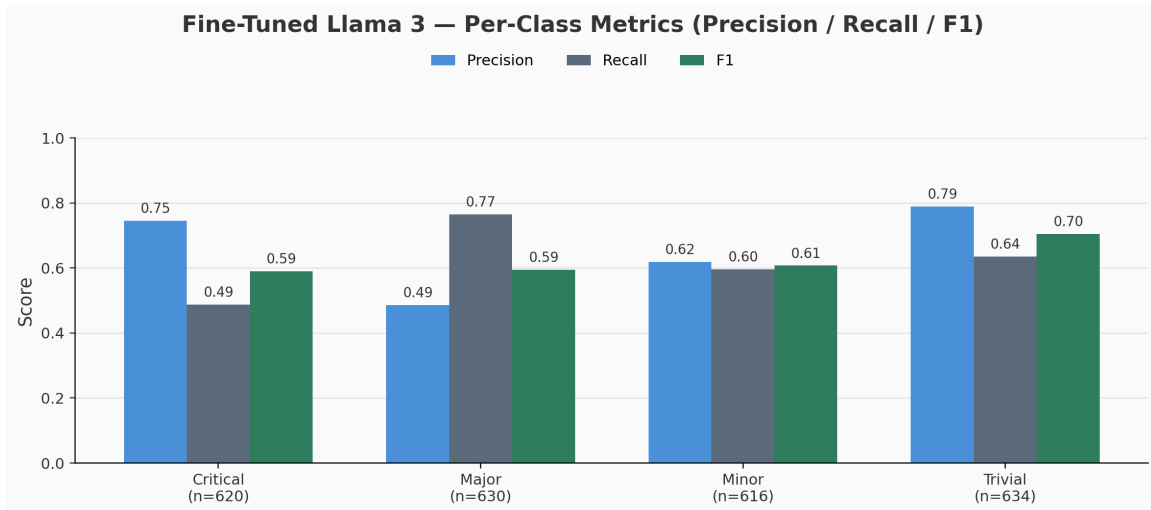


Figure 4-8: Class Distribution For Fine-Tuned Llama 3.

recall. In this case, the recall and precision percentages are stable and close to each other. For Llama 3 the recall is 62% and precision is 66%, whereas Qwen 2.5 has a recall of 61% and precision of 63%. The Fig. 4-8 shows that distribution clearer. It means that there is no bias towards any of the classes. The fine-tuning affected the LLMs positively and they managed to learn how to distinguish between severe and trivial issues. Thanks to the balancing strategy, the potential problem of a lazy shortcut was eliminated at the initial stage.

4.5 Computational Latency

As one of the goals of this project is to prove the application of ML models in a real software engineering environment, the inference duration was saved for each model. It was important to compare not only the classifying capabilities, but also the speed and computational requirements. In the case of traditional baselines, a rapid execution could be seen, in the range of 0.5 and 1.1 seconds, for 4000 samples overall. The result is undoubtedly efficient in terms of computational power, as the standard CPU quickly processes the vectorized bug reports with a use of TF-IDF. In this scenario, the classification happens almost instantly, as there is just an optimized matrix multiplication.

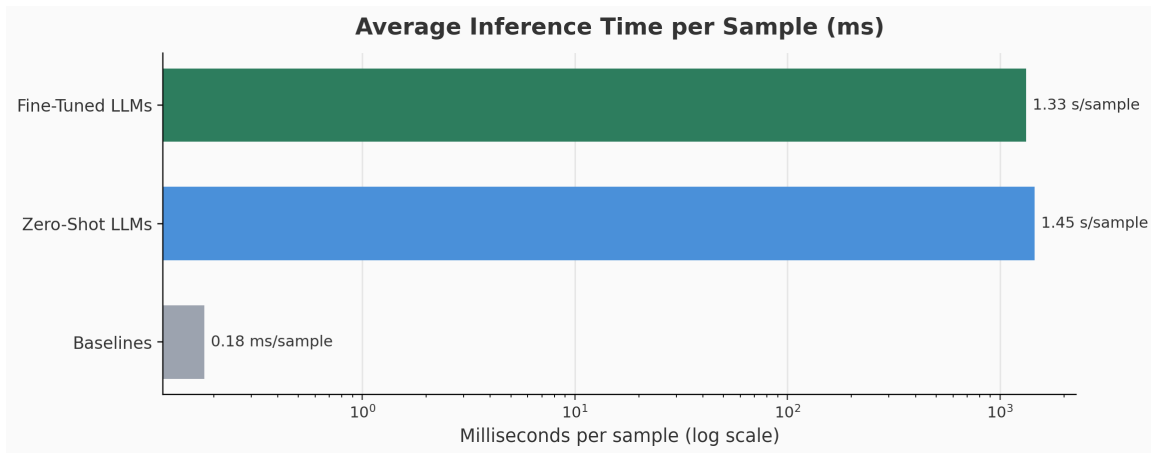


Figure 4-9: Comparison of Average Inference Time across Baselines and LLMs.

On the other hand, you can notice an incredible difference in the computational latency between the traditional models and LLMs. It is fully shown in the Fig. 4-9. Starting from zero-shot models, they were quite slower even than the fine-tuned versions. Zero-shot Llama 3 processed a 2500-sample dataset in 3428 seconds, whereas Qwen 2.5 needed 3845 seconds for the same task. It can be explained by the fact that they apply autoregressive token generation. To be precise, there is a wild time contrast, because LLMs have to read through the entire bug report, almost like a real human. Also, the instructions and severity explanations are added to that, topping up the token limit. Even if the model processes the prompt quickly, it has to generate its own response, which also requires several seconds, as the zero-shot LLMs tend to disobey the JSON structure and output a longer answer.

Turning to the fine-tuned LLMs, they delivered impressive results in terms of the inference latency. Even though the models became more powerful at severity classification, they even managed to become optimized, saving 5-13% of time during the inference. Such a success happens thanks to the domain adaptation. To be exact in the numbers, fine-tuned Llama 3 reaches the 3274 seconds result after processing the same dataset of 2500 samples, while Qwen 2.5 is a little slower, stopping at 3366 seconds. Overall, that accounts for approximately 3-10 minutes saved during the batch evaluation. The analysis of the gap shows that token mechanics plays a huge role in this task. When the LLM was fine-tuned, it learned the instructions

and severity explanations, which is why it processes such commands more quickly than their zero-shot version. Moreover, the fine-tuned LLM performs perfectly at outputting the severity in a JSON format. The model does not hallucinate and adheres to the instructions. As a result, lots of tokens were saved, winning in terms of the latency.

To compare, traditional ML models offer a rapid speed, delivering a latency close to seconds. They are an ideal choice for instant bug reporting. The models can be called via API to deliver a live result. Whereas, submitting a batch of errors to LLMs will require a much longer period, depending on the exact number of the bugs. This scenario would be useful, when the user wants to prioritize quality and deep contextual understanding over speed. Hypothetically, the processing should not take more than an hour, which is a decent result, possibly analyzing the batch of bugs over a workday.

In the initial stage, the application contained the monolithic type of architecture for live bug ingestion and batch evaluation features. It means that during the evaluation of 2500 samples, it was required from the LLMs to produce a JSON string, containing the summary, description and reasoning apart from the severity. After running a couple of tests, there was a decision to create a decoupled architecture, where during the batch evaluation, the model was only required to output a severity. This way, the LLM performs much better in accuracy and speed. Meanwhile, the live bug ingestion contains a feature of additional reasoning and summary, which delivers more value for a potential user, even though taking a little bit more time for a single bug processing.

As previously mentioned, the live bug ingestion was divided into two stages, creating a decoupled architecture. While the zero-shot LLMs run as at the beginning, producing all of the fields at one iteration, the fine-tuned models were altered considerably. In order to reach the maximum accuracy and F1 score, they were trained to output severity only. Subsequently, during the live bug ingestion they require additional support for outputting the reasoning and summary. At the first stage of a decoupled architecture a prompt is sent to a fine-tuned LLM, which is only required

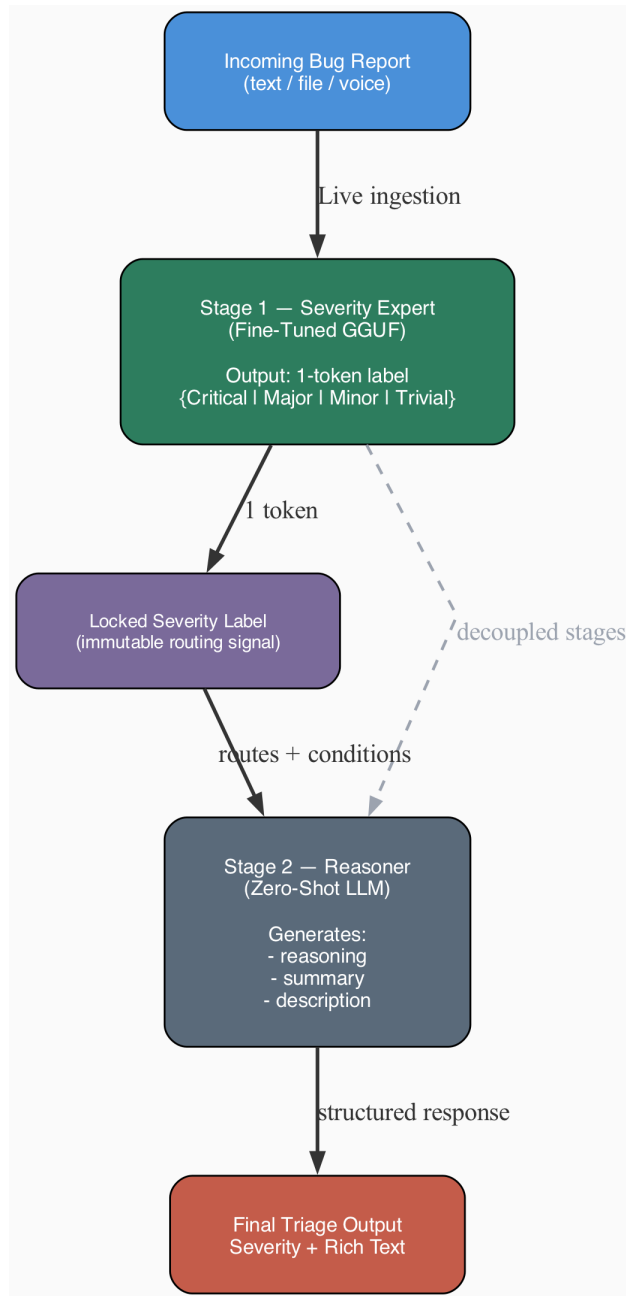


Figure 4-10: Diagram of a Decoupled Architecture in the LLM Pipeline.

to output a severity. With such an approach the highest accuracy is achieved and this model acts as an expert classifier. Then, after the severity is received, the second stage begins, where the zero-shot LLM is asked to accept the bug report together with the severity level. The prompt strictly requires the model to analyze the issue and provide the reasoning, summary and description without touching the severity.

This stage is called reasoning, as it just provides the additional explanation. The full algorithm of actions can be seen in the Fig. 4-10. Therefore, the zero-shot LLM is saved from hallucinating and providing the inaccurate or even non-existent severity type. Finally, the output contains a result with great precision and rich according analysis.

4.6 Web Application and Importing into Jira

One of the goals of this project is to prove its applicability in the real world scenario. For this matter, the architecture of this app is at the professional level, implementing frontend with React and integrating it together with backend via FastAPI. This approach perfectly aligns with the modern enterprise solutions. Moreover, for the convenience of use, the bug reporting is allowed in multiple formats. The user has the options with a manual input, where they type the description of the bug. Apart from that, there is an audio-to-text library that allows the users to dictate the report with voice. Obviously, the most common file formats are also embedded in the application, letting the user upload PDF, TXT, DOCX and JSON types of documents. Such a diverse functionality makes sure that the user has freedom in their reporting style.

The user flow of the live triage looks pretty straightforward from the frontend side. The user submits a bug report in a desired format and gets transferred to the result page, where it contains the severity label, description, summary and reasoning of the bug. For visual purposes, the Figs. 4-11 and 4-12 contain the described user flow. However, on the backend it is a huge load of operations, including two stages in a decoupled architecture. As previously mentioned, the first stage calls the fine-tuned LLM, which solely classifies the severity, whereas the second stage sends that label to the zero-shot LLM, which produces additional text explanation.

To explain the additional text generation, it is needed for guiding the project managers and logging enough context for a future review. On the backend, the parser successfully connects the output from both LLMs and sends them to the frontend, where the user sees a structured design with inserted fields of summary, description

CyberTriage New Ingestion Batch Evaluation History analyst@cybertrriage.com

Ingest New Bug Report

Paste logs or a bug description and choose a classification model.

Classification Model

Baseline ML LLMs

MODEL VERSION

Llama 3 8B (Fine-tuned)

Bug Description / Logs

Paste stack trace, error logs, or user description here...

Click to upload or drag and drop
TXT, LOG, JSON, PDF, DOCX up to 10MB

Analyze Bug

Figure 4-11: Bug Ingestion Interface.

CyberTriage New Ingestion Batch Evaluation History analyst@cybertrriage.com

Analysis Results

Critical Completed

AI Reasoning

The bug report describes a critical issue where both workspaces (build and dev) are locked, indicating a system crash or data loss. This severity classification is appropriate because it matches the definition of Critical: System crash, data loss, security vulnerability.

Export to Jira

Summary

Workspaces Locked

PRIORITY

Critical

Description

Both build and development workspaces become unresponsive, suggesting a critical issue that affects the overall system functionality.

Save Draft Create Jira Ticket

Figure 4-12: Analysis of the Bug with Four Fields.

and reasoning. These fields are standard for ticket systems like Jira. Each developer needs to understand the potential cause of the bug, for that reason this information is provided. In the Fig. 4-14 you can see how the mapping of the fields is done inside

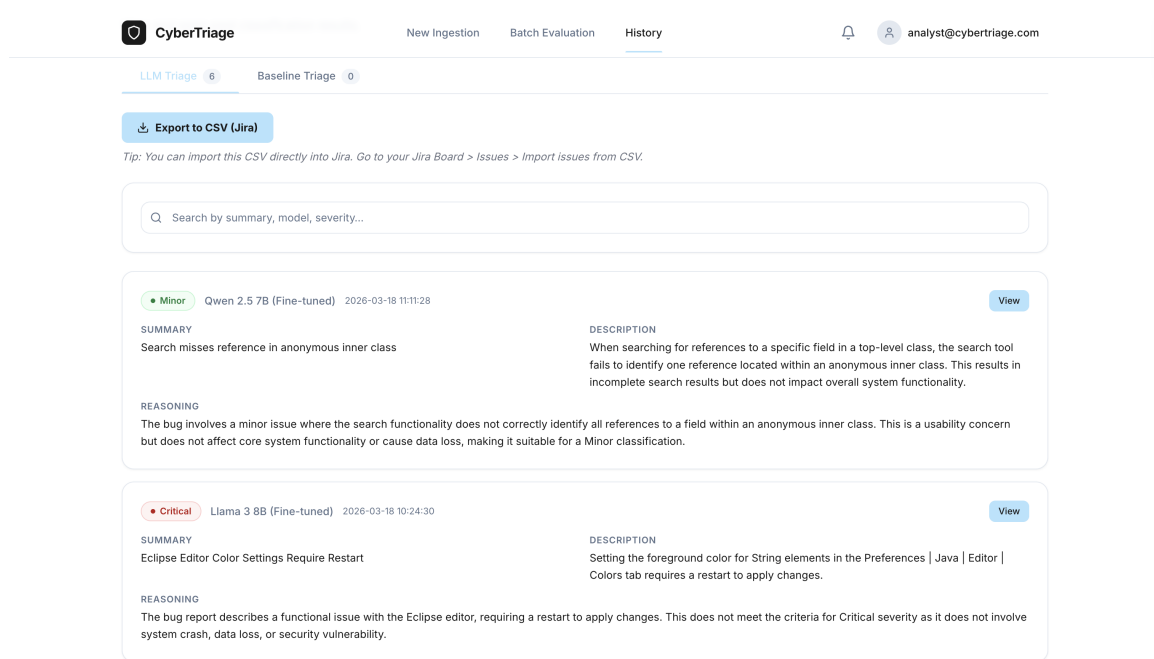


Figure 4-13: Bug History with an Export Function.

of the Jira system.

One of the most useful features inside of the application is the adaptation of the bug report for Jira. It bridges the gap between the academic purpose and a real bug triage in a software engineering domain. To be precise, each bug ingestion is saved in the history and left to be exportable into Jira. The history page is demonstrated in Fig. 4-13. To make it happen, the study adapted the JSON output for the import module in Jira, which requires a CSV file with strict column names and formatting. In fact, the data fields have to be wrapped in quotes, so the research implemented this converter. Additionally, there was a mismatch in the priority levels, as Jira is designed for all task types, not only bugs. Subsequently, the mismatched severities were mapped together, converting critical into highest priority and trivial into lowest, whereas major is called high and minor is low priority. Such a robust converter guarantees that the bug report will be successfully imported into Jira without crashing. In the Fig. 4-15, there is an example of a fortunate batch import.

Considering that the import function was seamlessly integrated into the web application, it delivers an immediate value for the potential user. The project manager

Setup

Settings

Map fields

Map values

Map fields

Select the CSV fields to import, then set how you would like these converted to fields in JIRA. You can optionally map field values on the next screen.

CSV Field	JIRA field	Map field value
Description (e.g. When searching for references to a specific ...)	→ Description	☐
Issue Type (e.g. Bug)	→ Issue Type	☒
Priority (e.g. Low)	→ Priority	☐
Summary (e.g. Search misses reference in anonymous inner class ...)	→ Summary	☐

Next

Back

Figure 4-14: Import Process in Jira.

Jira

Search

+ Create

For you

Recent

Starred

Apps

Plans

Spaces

Teams

Filters

Search work items

Default filters

My open work items

Reported by me

All work items

Open work items

Done work items

Viewed recently

Created recently

Resolved recently

Updated recently

View all filters

More

All work

Search all apps

Apps

Share

Basic

JQL

project=THS AND issuekey>=THS-37 AND issuekey<=THS-42 AND creator=currentUser()

Clear filters

Save filter

Work	Assignee	Reporter	Priority	Status	Res
☐ THS-42 HTTP 400 Bad Request Error	Unassigned	Asset	Highest	TO DO	Unresolver
☐ THS-41 Duplicate Entries in 'Sites to Visit' List	Unassigned	Asset	High	TO DO	Unresolver
☐ THS-40 Code-Assist Proposal Inconsistency	Unassigned	Asset	Highest	TO DO	Unresolver
☐ THS-39 Internal Package Export Issue	Unassigned	Asset	Lowest	TO DO	Unresolver
☐ THS-38 Eclipse Editor Color Settings Require Restart	Unassigned	Asset	Highest	TO DO	Unresolver
☐ THS-37 Search misses reference in anonymous inner class	Unassigned	Asset	Low	TO DO	Unresolver

6 of 6

Figure 4-15: Successfully Imported Tasks in Jira.

can export the whole batch of bugs analyzed by AI, delivering a high quality report and saving a lot of time for their software development team. Therefore, this tool turns the academic experiment into the actual software, which can transform the daily routine of bug triaging.

Chapter 5

Discussion

5.1 Addressing Research Questions

5.1.1 Research Question 1

In general, the gap in the performance between the traditional ML models and LLMs is significant. That phenomenon was justified by the experiment run in this research, where the F1 score was evaluated on a software bug dataset. It is clearly noticeable how traditional models hit the ceiling of 50% in F1, no matter which type of algorithm it is. There were probabilistic, linear and tree-based models participating in the overall comparison, none of them could break through the established threshold. At the same time, the CodeBERT encoder pushed the limit to 53% in F1, which was a peak performance among the baselines.

The main reason for such a stagnation is the TF-IDF vectorization approach. This method is already outdated, not being able to process complex technical reports, which are highly interconnected by logic. If there is a reliance on feature independence, treating all words and code chunks separately, the possibilities are deliberately limited by that. It was already proven by Lamkanfi et al., who researched the traditional classifiers in the software bug classification [32]. They identified that the classic models are too simple for understanding technical logic or polysemy, which fully matches with our findings. Apart from that, due to the vocabulary overlap in

the middle classes, like major and minor, the models struggled to learn the difference. According to Tian et al., his opinion was discussed, where he stated that the severity should rather be an ordinal and contextual metric, not isolated as in TF-IDF vectorization [46].

On the other hand, LLMs after a proper fine-tuning can overcome the ceiling and reach up to 62% in F1 score. Important to mention, the transformer architecture, which contains the attention mechanisms, plays one of the main roles here. The LLM after learning the contextual dependencies in the bug reports is able to act as a real human expert, mapping the problem with some software component and understanding the risks. It acts much more advanced compared to traditional baselines that just count the occurrences of the critical words. Subsequently, the problem of a shared vocabulary is no longer an obstacle, as the LLM is able to navigate any text, comprehending the intent behind it.

5.1.2 Research Question 2

Moving forward, the second research question tackled the Low-Rank Adaptation (LoRA) technique. There was a doubt whether a significant boost in the classification accuracy would be achieved. It turned out that the fine-tuning worked successfully, as zero-shot prompting was simply ineffective. Even though the models like Llama 3 and Qwen 2.5 contain billions of parameters and have a general knowledge on any domain, it was still not enough to overcome the classic ML models. The accuracy hit the plateau of 50% and F1 was limited with a 49% score.

The outcome of the zero-shot models was not consistent over the period, leading to occasional hallucinations. This is the main reason why default LLMs fail at this task. Their technical ground is already advanced enough to understand the code and error logs, what they actually lack is the adherence to the specific output format. As stated before by Heo, the zero-shot LLMs struggle with understanding the exact rules in a domain, as well as the cultural nuances inside of the software company [20]. That combined leads to the deviation from the instructions and unexpected behavior.

The situation changed considerably after the application of LoRA. It influenced

the LLM’s internal weights and adjusted the pre-trained neural networks. As a result of that, the model could learn the dialect of bug reporting inside of the software companies. Additionally, it could understand that there exist only four severity levels, not coming up with its own ones. Following the good practice of Agarwal and Madhavan, the research was able to fine-tune the LLM without removing its base engineering knowledge [2]. Using the best practice of LoRA, the study developed a top quality classification engine, which sticks to the instructions and outputs the desired result without any surprises. To conclude, the fine-tuned LLM stopped being chatty and delivered a full consistency in the successful parsing of a severity, skyrocketing the F1 score to 62%.

5.1.3 Research Question 3

The third research question is directly related to the messy nature of bug reports that contain a lot of stack traces and code snippets. The LLMs are able to navigate the noise without any issues, differentiating between the critical markers and irrelevant data. After running the experiments on the traditional ML models, it became clear that they get confused inside of the technical noise. It was previously justified by Goyal and Sardana, where they came up with a conclusion that classic models get lost and miss the real severity triggers [17]. The predictive signal vanishes due to the feature frequency reliance. To be exact, when the classifier sees a stack trace of 100 lines, it treats them as equal variables, each independent of each other.

In the past, for traditional models, one of the most efficient solutions for removing noise was data pruning. It was shown by Shivaji et al., the datasets had to be reduced to the optimal sizes, leaving behind lots of tokens and relying on extreme feature selection [44]. Even though it boosted the performance of traditional models, it could leave them without a valuable context, often misleading in some of the edge cases.

In case of the fine-tuned LLMs, they truly act as an instructed senior QA engineer, effectively ignoring the repetitive logs. The transformer architecture helps immensely in the question with noisy data, it dynamically allocates the attention to the most

necessary areas. When the model analyzes the report, it skims through the irrelevant data, stopping deeply on the real semantic triggers. This way, the LLM is able to deliver the best accuracy without data pruning, which might get rid of the important information.

5.1.4 Research Question 4

Turning to the last question, splitting classification and summarization tasks impacts the LLM’s accuracy, as well as the latency of the output, which is shown in Fig. 5-1. Such an implementation is called a decoupled architecture. The initial concern was the autoregressive token generation. For example, when the LLM is required to identify severity and output the reasoning, its attention is divided across several tasks, plummeting the classification accuracy. Apart from that, there is a risk of unacceptable output duration, when the LLM is asked to generate a long response at a single run. It can even lead to VRAM crashes that totally decrease the user satisfaction.

The most important goal of the research was to maximize the accuracy of severity prediction. Meanwhile, the summarization was a secondary objective, more like a nice addition. Subsequently, dividing the system into two pipelines is a wise decision in terms of the computational resources. In the first stage, the fine-tuned LLM fully focuses on the classification task, delivering the peak accuracy of 62%. The output at the first stage is much faster, as the model does not generate a long answer by spending thousands of tokens.

As regards the second stage, the zero-shot LLM produces a summarization based on the bug report and received severity. This process is safe and sound, thanks to the hard constraint on severity, meaning that it cannot be changed, the model is secure from hallucination. The zero-shot LLM already has decent reasoning skills, understanding the technical language. Splitting the live bug ingestion into two pipelines positively affects the user’s final output. As a result, the user is able to get a top quality analysis without a huge overload on the hardware.

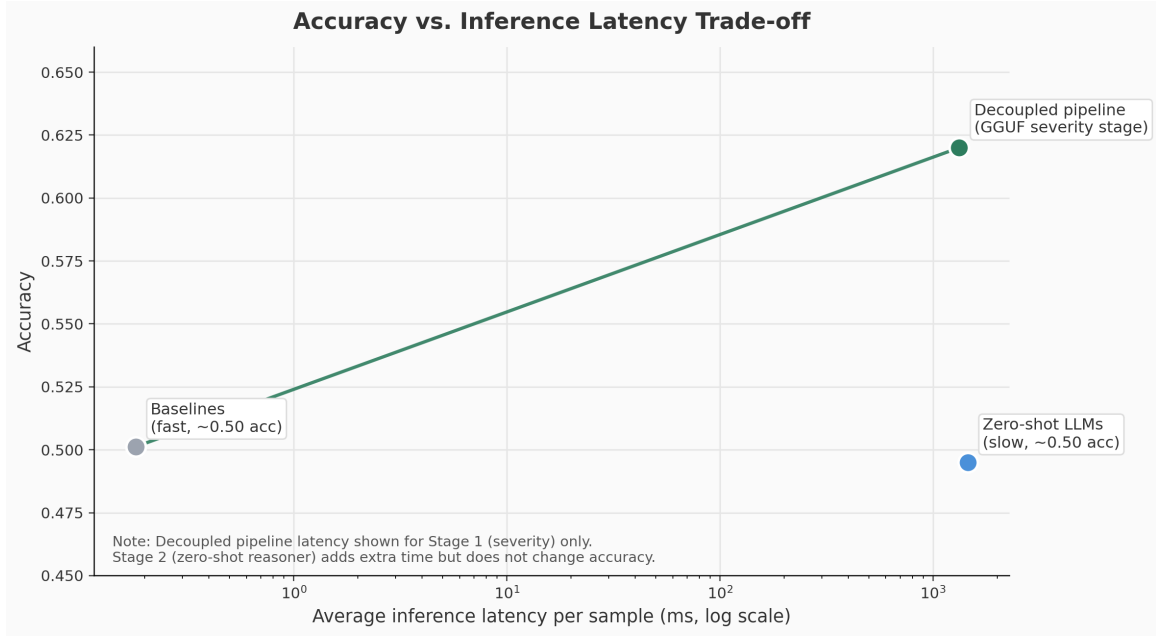


Figure 5-1: Trade-Off in Accuracy and Inference Latency.

5.2 Comparison of Llama and Qwen

One of the main objectives of this research was to demonstrate the dominance of LLMs in the specific task of bug classification. Besides, the thesis used two LLMs for a practical demonstration that fine-tuning works efficiently on any architecture. While having Llama 3 8B and Qwen 2.5 7B among the options, it would be unwise to skip the comparison between these two state-of-the-art models. Considering the fact that they were in almost equal conditions, being evaluated on 2500 samples, the slightly different results were received. The fine-tuning technique was absolutely the same, applying the QLoRA hyperparameters in a Python script. The final metrics suggest that Llama 3 performed marginally better, surpassing Qwen by 2% in F1 score.

Strikingly, Qwen 2.5 was initially the favorite to win the battle against Llama 3. Though the number of the parameters is lower, the model made by Alibaba Cloud was trained specifically for programming and calculus [42]. The model is the ideal choice for following a JSON structure according to their technical report. Nevertheless, the model loses a head-to-head comparison to a less technical Llama 3.

In this research, the bug triaging process is a strict severity classification. It does not require the model to generate the code or debug the issue. The model’s results can only be visible from the classification task, where the accuracy and F1 score can be calculated. Subsequently, for the model to reach high accuracy, it also needs to strictly adhere to the instructions. This is exactly the case for Llama 3, which excels at instruction following thanks to the great post-training alignment, where Meta made a huge focus [36]. Llama 3 perfectly follows the prompt instructions and produces the correct severity label without any deviation. Meanwhile, Qwen 2.5 has a slightly more advanced technical knowledge. However, it is not realized as an advantage. In fact, the immunity to edge cases and severity confusion turned out to be a more superior quality in this task, as Llama 3 demonstrated how resilient it is throughout the batch evaluation.

Moreover, apart from the training differences, Meta’s model is a bit larger in terms of the parameters size. Llama 3 contains 8 billion parameters, while Qwen has 7 billion of them. While it may not be a grand disadvantage for Qwen, it still justifies its slight underperformance compared to Llama. As there are highly dense vector embeddings, Llama is able to navigate through the bug report with more computational power, catching the complex triggers. To conclude, both LLMs became extremely efficient after fine-tuning, proving our hypothesis and surpassing traditional baselines. Even if the difference is not significant, Llama 3 still becomes a winner thanks to the strict instructional alignment.

5.3 Practical Application

Nowadays, most of the academic research is focused only on theoretical evaluation. In this case, the study combines the benchmarking together with the real application for software engineering. This is the type of software that development teams can adopt without any deep introduction. There is not only the strong and decoupled LLM pipeline, but also the React and FastAPI stack that follows the modern application standards. If any of the companies tried to use this software for their professional

use, their return on investment would be incredible. There is no doubt that project managers spend a vast amount of time trying to process the endless bug reports, which contain a lot of messy and unstructured data. Moreover, they may lack the technical degree to give a conclusion on them. Even if they try to consult with system architects, there is still a latency in the communication. The automation of the bug triage could be a handy tool for any software team, saving valuable resources for the company. Considering that the current accuracy is equal to 62%, it is a great tool for at least a preliminary assessment. Additionally, the model could be retrained on a specific dataset of tasks exported from the company's ticket system. Thus, the software engineers will be busy with bug fixing instead of a time-costly evaluation and analysis.

The application developed for this thesis guarantees the automation of the bug triage process. It impacts one of the most important metrics inside of the engineering teams, mean time to repair (MTTR). It is a frequent event when the task receives an incorrect severity and gets endlessly reassigned, creating a chain. Such a phenomenon is famous as bug tossing. In our case, the automated solution makes sure that the severity is correctly assigned at the initial stage. Moreover, the system provides a structured and logical explanation with a potential cause of the bug. Considering that it is also transferable directly to Jira, it becomes a handy tool for any software engineer. By providing a clear summary and assessing the severity correctly, this bug triaging system removes the miscommunications and positively affects the MTTR. The Fig. 5-2 displays how an automation in bug triaging can influence the SDLC, making the life easier for developers.

Assessing the system's applicability into modern workflows, there is a huge potential of dominating this area. Firstly, the application already has a feature of exporting via CSV format. It is fully adapted for Jira standards, so that the import completes successfully. The majority of the software companies, 60% to be exact, operate in Jira. Such an integration signifies a compliance with global trends and a useful feature for developers, connecting with CI/CD pipelines. When the system outputs a JSON, during an export, it maps the related fields to Jira's format and creates a CSV

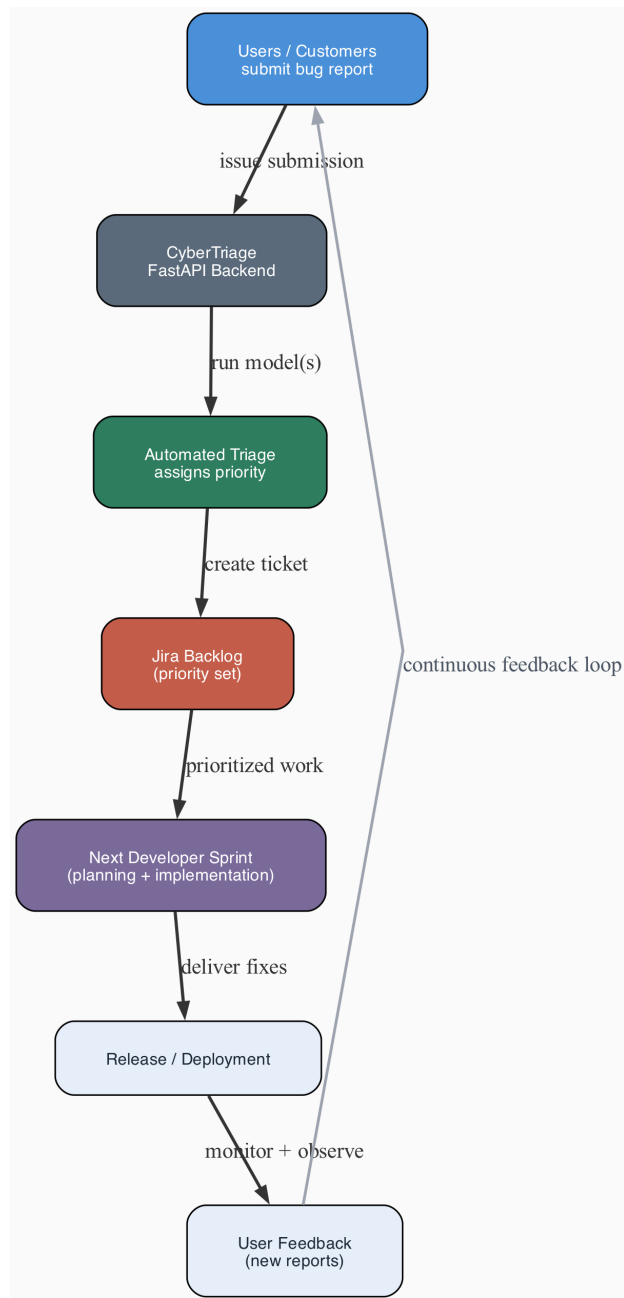


Figure 5-2: Flowchart of Automated Bug Triaging in a SDLC.

file. The import function inside of Atlassian allows uploading a dozen of such files, demonstrating how each bug is saved and which fields got transferred correctly. This user flow allows the project managers to easily assign the issues, without manually creating them. Additionally, this functionality can further be upgraded by the Jira Cloud REST API, that allows creating tickets immediately, without uploading any

files.

To conclude, the modern directions in software engineering are successfully followed in this research. As it was mentioned by Gropler et al., the European Union calls everyone to focus on the practical application of generative AI, rather than non-stop following the theoretical benchmarks [18]. This research demonstrates not only the potential application of a system, with artificial metrics, but the real product, ready for production use. The product guarantees that locally hosted LLMs can act as an engine, whereas the fine-tuning is accessible to everyone, delivering much better results. Thus, this thesis promotes the daily use of AI for beneficial purposes in software development.

5.4 System Limitations

Turning to the limitations of the system and threats to validity, it was made sure to cover them all, in order to act in accordance with empirical guidelines introduced by Wohlin et al. [48]. One of the first limitations is the computational latency during the bug ingestion analysis made by LLMs. The system is simply not able to handle thousands of bugs in a short period of time. Big software developing corporations may have a high number of reports, requiring powerful hardware and hours of a processing period. The traditional ML models can still be useful in such scenarios, solely because of their rapid response. Moreover, in order to run the models with 7-8B parameters, the 4 bit quantization had to be utilized. The research made by Dettmers et al. already highlighted the QLoRA technique trade-off, when the model is compressed and it decreases the VRAM requirements from 16 GB to 6 GB [14]. The disadvantage on the other side is the dropping accuracy compared to the analogous models with 16 bits.

Moving forward, there was another obstacle to domain familiarity. Most of the company cultures and software domains vary in terms of the task reporting, as well as their severity range. In this case, the data was gathered from Hugging Face, which collected the tasks from open-source repositories like Mozilla and Eclipse. Those

bugs are directly related to web applications and IDE environments. Subsequently, the programming languages used there are mostly related to Java and C++. On the other hand, there may be mobile, gaming and other operating systems, where the current solution is not adapted for. For that purpose, the fine-tuned model would have to be retrained again on a target dataset. As regards the F1 score, it may fluctuate depending on the complexity and cohesion of data, due to the different severity standards in various fields.

Ultimately, the system undoubtedly has limitations in the provided information. The LLM is advanced enough to produce an accurate severity level, adding up the reasoning, summary and description. However, for some teams it may not be enough. One of the lacking information is the potential assignee to the issue, where AI analyses the previous ticket history and proposes the developer for a fix. Additionally, the LLM could identify the specific module in the system and assign an epic to the bug. These upgrades would move the system to the next level, making it irreplaceable in the long term. Even if an F1 score of 62% is a great improvement compared to the traditional models, it is still not good enough to be fully used in the production environment. At the moment, the system should be used under supervision, serving for a decision support. In other words, it can also be called as human-in-the-loop triage assistant. Currently, the system achieves the MVP level, providing the necessary functionality. Meanwhile, there is a clear vision of boundaries and how the system can move forward, satisfying more users. In order to deliver a better experience, the project should concentrate on company-specific fine-tuning. It can be achieved by using a corporate Jira data and improving calibration methods.

Chapter 6

Conclusion

6.1 Research Journey

Nowadays, companies lose millions of dollars on bugs and their inefficient fixes. With AI advancements, the software development lifecycle needs a revolution in terms of handling issues. There is no limit to the project sizes, they expand and produce more bugs than ever. If we try to tackle each bug report manually, it becomes inconsistent and time-consuming, leading us to a subjective assessment. Due to those problems, the bugs are incorrectly given a severity label, resulting in a time loss. The MTTR metric decreases and tasks get sent from one developer to another, creating chaos. Undoubtedly, the automated tool for bug classification is needed as air in modern software engineering.

The solution as an automated bug triage system is proposed in this research. The ultimate goal is to compare the NLP techniques for severity classification, assessing the capabilities of the most advanced methods, like LLMs. As an initial step, it was important to prepare a structured and balanced dataset, suitable for the software engineering domain. Then, after subsampling the data to 20,000 bugs, equally for each class, there was a comparison stage. The research created a dual pipeline for training and testing the traditional ML models, relying on TF-IDF, with LLMs. The most fascinating evaluation was hidden in the second pipeline, containing zero-shot Llama 3 8B and Qwen 2.5 7B together with their fine-tuned counterparts. The

fine-tuning strategy became the game-changer thanks to the Low-Rank Adaptation (LoRA).

Apart from running academic benchmarking, the ultimate goal was to deliver a useful big triaging application at the MVP level. While the LLMs were fine-tuned and installed as GGUF files, they were inserted into the system, operating on React from the frontend and FastAPI on the backend. What initially was a theoretical experiment and something abstract, turned out to be applied practically. It happened thanks to the developed features, like file parsing, audio-to-text conversion and export to Jira via CSV. This demonstration led to the conclusion that advanced NLP methods can be seamlessly integrated in today's software development lifecycle, delivering immediate value for Agile teams.

6.2 Key Takeaways

Firstly, covering the results in the first pipeline of traditional models, it led to the conclusion that they are insufficient for a proper classification. No matter which architecture, the model simply hits the ceiling of 50% in F1 score, due to the naive text vectorization using TF-IDF. Such an approach totally misses the semantically valuable dependencies, because of the feature independence at the baseground. In addition to that, after facing the vocabulary overlap, the model gets confused with the close severity labels. The TF-IDF approach was easily surpassed by the neural model CodeBERT, delivering 53% in F1 score.

The next phase, which required the zero-shot LLM comparison, brought interesting insights to the light. Even though the standard models contain billions of parameters and are trained for technical tasks, they struggled with the classification task. They were not able to overcome classic models, stagnating at 49% F1 score. The reason behind the failure was not the lack of technical knowledge, rather an unfamiliarity with the specific task format, as well as the deviation from the expected output. The zero-shot LLMs occasionally hallucinated and broke the JSON parser by outputting the extra text or invented severity.

The fine-tuning phase turned out to be transformative in this research. While previous attempts could not surpass the F1 score of 53%, the LoRA tuning method allowed to use generative AI capabilities to its fullest. Afterwards, Llama 3 8B delivered 62% F1 score, while Qwen 2.5 7B fell a little behind at 60%. The biggest achievement was that fine-tuned models learned the format of the bug reports and how to output an appropriate severity, in a correct schema for JSON parsing. As previously mentioned, Llama 3 came out as a winner, even though Qwen 2.5 positioned itself as a coding model. The final outcome was that Llama 3 is better at following instructions and it managed to stay resilient throughout the severity mapping.

Last but not least, the live bug ingestion divides the LLM pipelines into two stages, where the fine-tuned model acts a classifying expert and zero-shot model performs the additional summarization. This way, it improves the computational latency and accuracy of the result by focusing on the most important job, severity classification. During the first stage, the fine-tuned LLM does not generate an excessive number of tokens, overloading the hardware. Instead, it produces the uneditable severity label, which is passed to the second LLM. Subsequently, using such a modern AI-agentic architecture, it provides not only the theoretical measurements, but the software with a high potential in the practical use.

6.3 Future Roadmap

Currently, the system achieved the MVP state and already became available to be used in practice. The functionality introduced in the system, on top of that connected with Jira, is enough to deliver the immense value. However, the generative AI domain is accelerating every day, providing new features and opportunities with each model release. The bug triaging software has great potential and to bring it into reality, there has to be established the future roadmap for the development.

While there is a feature of bug exporting into Jira, for now the responsible person has to download the CSV file and manually export it into Jira. At the moment, it is already a valuable feature, saving from a lot of manual work. Nevertheless, this

activity could even be improved by the integration of direct REST API. This way, the user could see the generated bug reports directly in Jira, without any additional transfer. Moreover, there is an opportunity to expand the number of integrations, adding famous issue trackers like Linear, Trello and Asana. In this scenario, a much bigger audience could be targeted, providing flexibility for the teams.

Apart from that, the bug triaging system could further be upgraded to become a preventive bug detector. Presently, the LLM acts blindly at the receival of the bug report, it does not see the full picture. It would be wise to integrate a connection between the Git commits and the whole codebase. Then, the model could identify the real root causes of the bugs and suggest the fixes at the early stages. If implemented, the system would act not as an assistant, but as an independently operating expert within the software team. Also, the LLM versions update with the speed of light. Nowadays, there is an ongoing battle between corporations, like OpenAI, Anthropic and Google, in the pursuit of the benchmarks. Considering those changes, it would be necessary to update the model in the engine of the system, in order to stay relevant and make the system more optimized.

Finally, the bug triaging process requires a more diverse input optionality. To be exact, the bug reports are usually supported by the video recordings and screenshots. The system needs an implementation of computer vision for being able to process the visual artifacts. Considering this addition, the system would receive the eyes and guide the user through the recording, pointing to a specific second in the video, where the bug happens. To sum up, if this application continues to be developing with more advanced NLP solutions, as well as the regular functionality, it would cover the majority of the pain points the software team members experience. As a result, the system would turn into an intelligent agent, automating the manual workflows and serving as a decision support tool.

Bibliography

- [1] N. Adhikari, R. Bista, and J. C. Ferreira. Leveraging machine learning for enhanced bug triaging in open-source software projects. *IEEE Access*, 13:136237–136254, 2025.
- [2] A. Agarwal and M. Madhavan. Fine-tuning code models for bug severity prediction: A comparative study with GraphCodeBERT and LoRA, jan 2026.
- [3] S. V. A. Ammu, S. S. Sehra, S. K. Sehra, and J. Singh. Amalgamation of classical and large language models for duplicate bug detection: A comparative study. *Computers, Materials & Continua*, 83(1):435–453, 2025.
- [4] Analytics Vidhya. LSTM for text classification | beginners guide to text classification. Analytics Vidhya, jun 2021.
- [5] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, May 2006.
- [6] A. Arcuri and L. Briand. A hitchhiker’s guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 24(3):219–250, 2012.
- [7] P. Bhattacharya, I. Neamtii, and C. R. Shelton. Automated, highly-accurate, bug assignment using machine learning and tossing graphs. *Journal of Systems and Software*, 85(10):2275–2292, Oct 2012.
- [8] T. U. Boran, D. M. Taha, Mahmut Furkan Gön, and E. Tüzün. Past, present, and future of bug tracking in the generative ai era. arXiv preprint arXiv:2510.08005, 2025.
- [9] L. Chen et al. A survey on evaluating large language models in code generation tasks. *arXiv preprint arXiv:2408.16498*, aug 2024.
- [10] D. Chicco and G. Jurman. The advantages of the matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation. *BMC Genomics*, 21(1), jan 2020.
- [11] C. A. Choquette-Choo, D. Sheldon, J. Proppe, J. Alphonso-Gibbs, and H. Gupta. A multi-label, dual-output deep neural network for automated bug triaging. In

2019 18th IEEE International Conference On Machine Learning And Applications (ICMLA), pages 937–944, Dec 2019.

- [12] Davor Cubranic. Automatic bug triage using text categorization. In *Proceedings of the 16th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, 2004.
- [13] A. P. Desai, R. Prajapati, T. Ravi, M. Luqman, and P. Yadav. Emerging trends in LLM benchmarking. In *2024 IEEE International Conference on Big Data (BigData)*, pages 8805–8807, dec 2024.
- [14] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer. QLoRA: Efficient finetuning of quantized LLMs. *arXiv preprint arXiv:2305.14314*, 2023.
- [15] S. Feng and C. Chen. Prompting is all you need: Automated android bug replay with large language models, feb 2024.
- [16] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer. A survey of quantization methods for efficient neural network inference. *arXiv preprint arXiv:2103.13630*, jun 2021.
- [17] A. Goyal and N. Sardana. Machine learning or information retrieval techniques for bug triaging: Which is better? *Prace Naukowe Uniwersytetu Ekonomicznego we Wrocławiu*, Jan 2017.
- [18] R. Gröpler et al. The future of generative AI in software engineering: A vision from industry and academia in the european GENIUS project. *arXiv preprint arXiv:2511.01348*, 2025.
- [19] Joshua Harris et al. Evaluating large language models for public health classification and extraction tasks. *arXiv preprint arXiv:2405.14766*, may 2024.
- [20] S. Heo. Prompt driven multimodal large language models for concrete defect identification. *IEEE Access*, 13:160278–160287, 2025.
- [21] H. Hong et al. Implementing and evaluating automated bug triage in industrial projects. *IEEE Access*, pages 1–1, Jan 2024.
- [22] X. Hou et al. Large language models for software engineering: A systematic literature review. *arXiv preprint arXiv:2308.10620*, Aug 2023.
- [23] E. J. Hu et al. LoRA: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.
- [24] Hadi Jahanshahi, Mucahit Cevik, K. Mousavi, and Ayşe Başar. Adptriage: Approximate dynamic programming for bug triage. *IEEE Transactions on Software Engineering*, 49(10):4594–4609, Oct 2023.

- [25] G. Jeong, S. Kim, and T. Zimmermann. Improving bug triage with bug tossing graphs. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Jan 2009.
- [26] H. Jin, L. Huang, H. Cai, J. Yan, B. Li, and H. Chen. From LLMs to LLM-based agents for software engineering: A survey of current, challenges and future. *arXiv preprint arXiv:2408.02479*, 2024.
- [27] L. Jonsson, M. Borg, D. Broman, K. Sandahl, S. Eldh, and P. Runeson. Automated bug assignment: Ensemble-based machine learning in large scale industrial contexts. *Empirical Software Engineering*, 21(4):1533–1578, Sep 2015.
- [28] M. R. Karim, A. Ihara, E. Choi, and H. Iida. Identifying and predicting key features to support bug reporting. *Journal of Software: Evolution and Process*, 31(12), aug 2019.
- [29] K. Kiashemshaki, A. Khosravani, A. Hosseinpour, and A. Akhavan. Automated bug triaging using instruction-tuned large language models. *arXiv preprint arXiv:2508.21156*, 2025.
- [30] H. G. Kim and Y. Park. Calibrating F1 scores for fair performance comparison of binary classification models with application to student dropout prediction. *IEEE Access*, 13:136554–136567, jan 2025.
- [31] A. Kumar, S. Haiduc, P. P. Das, and P. P. Chakrabarti. LLMs as evaluators: A novel approach to evaluate bug report summarization. *arXiv preprint arXiv:2409.00630*, 2024.
- [32] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals. Predicting the severity of a reported bug. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR)*, May 2010.
- [33] G. Lemaître, F. Nogueira, and C. K. Aridas. Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in machine learning. *Journal of Machine Learning Research*, 18(17):1–5, 2017.
- [34] Khabibulla Madatov and Sattarova Sapura Beknazarovna. Vectorization of Uzbek texts using the TF-IDF vectorizer method, nov 2023.
- [35] S. Mani, A. Sankaran, and R. Aralikkatte. Deeptriage: Exploring the effectiveness of deep learning for bug triaging. *arXiv preprint arXiv:1801.01275*, 2018.
- [36] Meta AI. Introducing meta llama 3: The most capable openly available llm to date. <https://ai.meta.com/blog/meta-llama-3/>, Apr 2024.
- [37] S. Mukherjee. LoRA: Low-rank adaptation of large language models explained. DigitalOcean, jul 2025.

- [38] H. Nisar, A. Tapp, and Marius George Linguraru. Green AI: Systematic review and guidelines for sustainability in artificial intelligence. *Preprints.org*, 2026.
- [39] F. Pedregosa et al. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12(85):2825–2830, 2011.
- [40] Prerna, Ranjana Beshra, and P. Singh. Bug triaging: A comprehensive review of current practices and the path ahead. In *2025 International Conference on Systems, Signals and Image Processing (IWSSIP)*, pages 1685–1689, Jun 2025.
- [41] H. Qin and X. Sun. Classifying bug reports into bugs and non-bugs using lstm. In *Proceedings of the Tenth Asia-Pacific Symposium on Internetware*, pages 1–4, Sep 2018.
- [42] Qwen Team. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*, 2024.
- [43] H. Sartaj, S. Ali, Paolo Arcaini, and A. Arcuri. Search-based software engineering and AI foundation models: Current landscape and future roadmap, oct 2025. Available at ResearchGate.
- [44] S. Shivaji, E. J. Whitehead, R. Akella, and Sunghun Kim. Reducing features to improve code change-based bug prediction. *IEEE Transactions on Software Engineering*, 39(4):552–569, Apr 2013.
- [45] H. Soudani, E. Kanoulas, and F. Hasibi. Fine tuning vs. retrieval augmented generation for less popular knowledge. *arXiv preprint arXiv:2403.01432*, 2024.
- [46] Y. Tian, D. Lo, and C. Sun. Drone: Predicting priority of reported bugs by multi-factor analysis. In *2013 29th IEEE International Conference on Software Maintenance (ICSM)*, Sep 2013.
- [47] Q. Umer, H. Liu, and I. Illahi. Cnn-based automatic prioritization of bug reports. *IEEE Transactions on Reliability*, 69(4):1341–1354, Dec 2020.
- [48] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.
- [49] T. Wolf et al. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, 2020.
- [50] Wwww.tha.de. Types of prompting, 2016.
- [51] J. Xuan et al. Towards effective bug triage with software data reduction techniques. *IEEE Transactions on Knowledge and Data Engineering*, 27(1):264–280, Jan 2015.

- [52] J. Xuan, H. Jiang, Z. Ren, and Z. Luo. Automatic bug triage using semi-supervised text classification. In *2010 IEEE 26th International Conference on Software Maintenance (ICSM)*, pages 1–10, 2010.
- [53] G. Yang, T. Zhang, and B. Lee. Towards semi-automatic bug triage and severity prediction based on topic model and multi-feature of bug reports. In *2014 IEEE 38th Annual Computer Software and Applications Conference*, Jul 2014.
- [54] Q. Zhang et al. A survey on large language models for software engineering. *arXiv preprint arXiv:2312.15223*, 2023.
- [55] Y. Zhao, L. Xiao, and S. Wong. A platform-agnostic framework for automatically identifying performance issue reports with heuristic linguistic patterns. *IEEE Transactions on Software Engineering*, pages 1–22, jan 2024.