Automating Severity Prediction of Bug Reports Using Large Language Models (LLM)

Mustafa Bilal[1], Nasir Rashid[1], Muneeb Babar[1], Sheraz Babar[2], Pervez Khan[1], Muhammad Salam[1]

[1]Department of Computer Science and IT, University of Malakand, Pakistan

[2]Department of Applied AI, School of Convergence, College of Computing and Informatics, Sungkyunkwan University, South Korea.

*__Correspondence__: nasir@uom.edu.pk

Citation | Bilal. M, Rashid. N, Babar. M, Babar. S, Khan. P, Salam. M, "Automating Severity Prediction of Bug Reports Using Large Language Model (LLM)", IJIST, Vol. 8 Issue. 3 pp 1485-1511, June 2026

DOI: https://doi.org/10.33411/IJIST/1941

Received| May 13, 2026 **Revised|** June 15, 2026 **Accepted|** June 19, 2026 **Published|** June 28, 2026.

Accurate prediction of bug report severity is essential for prioritizing software-maintenance activities, but manual severity assignment is costly, time-consuming, and potentially inconsistent. This study investigates prompt-based open-weight Large Language Models for automatically classifying bug reports as severe or non-severe without task-specific fine-tuning. The experiments used 3,535 bug reports from the CDT, JDT, Bugzilla, and Thunderbird projects. The original severity labels were mapped into two categories: blocker, critical, and major as severe, and normal, minor, and trivial as non-severe. LLaMA 3.1:8B and Qwen3:8B were evaluated under zero-shot and few-shot prompting and compared with Support Vector Machine, Multinomial Naive Bayes, Random Forest, and Long Short-Term Memory models using TF-IDF and Word2Vec representations. On the held-out test set, Qwen3:8B with few-shot prompting achieved 98.6% accuracy, 100% precision, 97.5% recall, and a 98.7% F1-score. Compared with the strongest conventional baseline, Random Forest with TF-IDF, it improved accuracy and F1-score by 26.6 and 24.2 percentage points, respectively. On an additional dataset of 200 synthetically generated bug reports, LLaMA 3.1:8B with zero-shot prompting achieved 97.99% accuracy and a 98.51% F1-score, while Qwen3:8B with few-shot prompting achieved 97.9% accuracy and a 97.5% F1-score. These results indicate that prompt-based LLMs can substantially reduce task-specific preprocessing and training requirements while providing high predictive performance. However, further evaluation on independent, human-authored, cross-project datasets is required before making broad claims regarding real-world generalization.

Keywords: Large Language Model; Machine Learning; Natural Language Processing; Deep Learning; Meta AI

Introduction:

In today's software development world, bugs are simply part of the process. A bug in a software system is any kind of error, flaw, or failure in a program that causes it to work incorrectly [1]. These issues can arise during the software system's development or later, when the software is in use. Over time, the complexity of the software system increases, and bugs may pose significant challenges to software quality and user experience.

To address these challenges, software development teams use bug reports. These reports are the primary channels through which issues are communicated to development teams and subsequently resolved. A typical bug report may include a description of the issue, its severity, the environment in which it occurs, and steps to reproduce it [2]. In large-scale systems, the number of reports can be difficult to handle. Therefore, it is important to quickly and accurately sort and prioritize them to ensure the most critical problems are addressed promptly.

A number of classical ML algorithms have been used to classify bug reports, including Naïve Bayes (NB), SVM, Random Forest (RF), Logistic Regression (LR), and AdaBoost [3]. Bug reports are classified based on severity and priority. These classical ML algorithms generally require numerical representations of text rather than directly processing raw textual data and rely heavily on word embedding techniques for feature extraction, such as TF-IDF and Word2Vec [4]. However, the limitations of these algorithms include the need for extensive pre-processing and the risk of overfitting, which can lead to poor generalization on unseen data. In addition to classical ML algorithms, researchers have explored LSTMs to tackle the same problem.

Software bug reports are an essential part of maintenance activities and play a significant role in improving software quality and user experience. Classifying these reports helps developers understand their nature, their consequences for software performance, and the reasons for failure [5]. However, classifying their severity is a tiresome and inefficient activity, particularly in large-scale systems, because it requires manual examination and significant resources. Classical machine learning approaches often encounter difficulties due to extensive training times, manual preprocessing, and limited generalizability. With prompt engineering and minimal training, Large Language Models (LLMs) such as the Meta AI open-weight model, LLaMA 3.1, and Qwen 3 provide a viable substitute for a range of tasks, including summarizing, translating languages, and answering questions [6][7][8]. Large Language Models (LLMs) and advanced ML techniques can be used to automate and improve the prediction of bug report severity. This research aims to automate the severity prediction of unstructured bug reports written in natural language using prompt engineering techniques, minimizing the need for preprocessing and reliance on labeled datasets, thereby making the process more efficient and scalable and ultimately benefiting software developers and quality assurance teams.

When using software systems, users may encounter issues or errors that go unnoticed during testing and development. Usually, these issues and errors are referred to as software bugs and defects. These bugs have a variety of negative consequences for the performance of the software system, including but not limited to security flaws, user experience issues, and system failures, which can increase overall maintenance costs [9]. In the early stages of software development, when requirements or specifications are not correctly implemented, these bugs arise.

Testing and maintenance phases in the SDLC play a vital role for many reasons, ensuring software quality and improved performance. One of the prominent activities of the maintenance phase is fixing the reported bugs [10][11]. Software failures are inevitable; no software system can be developed without bugs. These errors occur due to various factors such as evolving requirements, hardware limitations, and human errors [12]. Software

applications are a necessity of modern life, and user reviews are a vital part of their development and maintenance. Software users record their opinions as a part of interaction with the software, submit bugs, and demand new features on various platforms such as app stores, bug tracking systems (BTS), and online developer communities.

User reviews can be beneficial for enhancing software quality because they provide real-world feedback from users. User reviews underline errors that the development or testing team might miss. However, the rapid increase in such reviews and bug reports, coupled with developers' limited ability to resolve them effectively, poses a considerable challenge for software maintenance. Manually handling these reports is both laborious and time-consuming, significantly increasing the costs of software maintenance. In fact, maintenance accounts for around 60% of total expenses in the software development life cycle (SDLC), making it the most resource-intensive and costly phase [13].

Usually, when a bug arises in the software system, it is reported to the concerned software developer or team for resolution. These bugs or issues are reported through bug reports and bug tracking systems (BTS), where a bug report is a structured document that highlights issues found in the software system.

Common platforms for collecting such reports include Bugzilla and JIRA [14][15]. These bug tracking systems enable users to submit and track their bug reports. It also maintains records of reported bugs and acts as a bridge between software users and developers. A bug report typically includes features such as a unique bug ID, submission date, status (open, in progress, or resolved), severity, summary, and description. Severity is a crucial feature of a bug report that determines how soon a bug should be resolved. It assists software developers in prioritizing and fixing severe bugs in a timely [16][17].

In the Bugzilla tracking system, the severity level of a bug report can be blocker, critical, major, normal, minor, or trivial. The first three (blocker, critical, and major) are considered severe, while the last three (normal, minor, and trivial) are considered non-severe due to their impact on the software system's functionality. At reporting time, users determine and assign a severity level to bug reports, a tedious process that may be inaccurate [13]. A lack of domain knowledge and user bias may lead to an incorrect assignment of the severity feature. For software developers, manually evaluating a large number of bug reports is difficult, tiresome, and costly, especially in large-scale software systems. Similarly, automating severity levels of bug reports written in human language remains a key challenge.

Recent research has progressively shifted from manually engineered textual features toward deep neural architectures, pretrained language models, and generative LLMs. [18] proposed a topic-based feature-selection and CNN–LSTM approach for bug severity prediction, demonstrating the effectiveness of combining semantic topic information with sequential representations. [19] subsequently introduced a lightweight bug-triage framework based on pretrained language models, while [20] compared transformer-based neural text representations for bug-triaging tasks. More recently, [21] applied BERT to bug severity prediction for mobile-application maintenance. [22] proposed SevPredict using GPT-2, demonstrating the emerging application of generative language models to software-maintenance classification. [23] compared graph neural networks and LLMs for bug priority and severity prediction, showing that LLM-based severity classification is an active but not yet conclusively resolved research area. These developments establish the need for controlled evaluation of newer open-weight instruction-following models under zero-shot and few-shot settings.

Most of the existing literature focuses merely on classical machine learning algorithms based on supervised and unsupervised classifications, such as Support Vector Machine (SVM), Naive Bayes (NB), Naive Bayes Multinomial (NBM), Logistic Regression (LR), and Random Forest (RF), etc., to predict severity levels of bug reports. Recently,

automating the severity prediction of software bug reports written in natural language has received growing attention through ML techniques [2] [7] [24] [25] [26] [27] [28] [29] [30] [31] [32] [33].

These methods struggle with challenges including manual data cleaning and normalization, overfitting, reliance on labeled datasets, and poor generalizability.

Despite substantial progress in automated bug severity prediction, several research gaps remain. First, classical machine-learning and LSTM-based methods depend on labeled training data, task-specific text preprocessing, feature extraction, and hyperparameter optimization. Second, transformer-based approaches such as BERT generally require supervised fine-tuning, which can restrict rapid deployment in projects with limited labeled data. Third, although recent studies have explored GPT-2 and other LLMs for bug severity prediction, limited evidence is available concerning the prompt-only performance of newer open-weight, instruction-following models such as LLaMA 3.1 and Qwen3. In particular, the comparative effects of zero-shot and few-shot prompting have not been sufficiently examined under a common experimental protocol that also includes TF-IDF, Word2Vec, conventional classifiers, and LSTM. Furthermore, the robustness of these approaches beyond the original held-out dataset requires additional investigation. This study addresses these gaps by evaluating two open-weight instruction-following LLMs without task-specific fine-tuning and comparing them with conventional machine-learning and deep-learning baselines using common severity definitions and evaluation metrics.

Moreover, the potential of Large Language Models, particularly LLaMA 3.1 and Qwen 3, for severity prediction remains underexplored and warrants further investigation. This work proposes an automated approach that leverages LLaMA (Meta AI) and Qwen LLMs, combining prompt engineering techniques to effectively predict the severity levels of bug reports. The work aims to gain scalable predictions with minimal training requirements by leveraging zero-shot and few-shot learning techniques.

The aim of this research is to propose an automated approach using the open-weight Large Language Models LLaMA-3.1 and Qwen3 that can more accurately and efficiently predict the severity levels of bug reports and mitigate the existing limitations of classical machine learning approaches, such as manual preprocessing, overfitting, labeled dataset dependence, and poor generalizability. This research has the following objectives:

The overall objective of this study is to evaluate the effectiveness of prompt-based open-weight LLMs for binary bug report severity prediction without task-specific fine-tuning.

The specific objectives are:

- To measure the accuracy, precision, recall, and F1-score of LLaMA 3.1:8B and Qwen3:8B under zero-shot and few-shot prompting on a held-out portion of the 3,535-report dataset.
- To quantify the change in each LLM's performance when moving from zero-shot to few-shot prompting.
- To compare the performance of the two LLMs with SVM, Multinomial Naive Bayes, and Random Forest using TF-IDF and Word2Vec representations, as well as with an LSTM model.
- To evaluate the performance change of every model on an additional dataset containing 200 synthetically generated bug reports.
- To identify the model configuration that obtains the highest F1-score and accuracy under each evaluation condition.
- To assess whether the performance differences between the best LLM and the strongest baseline are statistically significant and practically meaningful.

This study aims to answer the following research questions.

RQ1: To what extent can the LLaMA3.1 model accurately classify software bug reports using zero-shot learning, and how does its performance compare under few-shot learning?

RQ2: How does the performance of LLaMA3.1 vary between known and unseen bug report datasets, and what does this reveal about its generalization capabilities?

RQ3: To what extent can the Qwen3 model accurately classify software bug reports using zero-shot learning, and how does its performance compare under few-shot learning?

RQ4: How does the performance of Qwen3 vary between known and unseen bug report datasets, and what does this reveal about its generalization capabilities?

RQ5: How effective are classical machine learning models in classifying software bug reports into severity categories when using TF-IDF and Word2Vec embedding techniques?

RQ6: How effectively do classical machine learning models retain performance on unseen bug reports when tested with TF-IDF and Word2Vec embeddings?

RQ7: How effective is the LSTM model in classifying software bug reports by severity based on sequential learning from textual data?

RQ8: To what extent can the LSTM model generalize to unseen bug report data, and how does its performance compare to classical models and LLMs?

RQ9: What are the relative benefits of LLaMA3.1 and Qwen3 to classical machine learning models and LSTM in the automation of bug severity prediction of reports?

Novelty and Contributions of the Study

The novelty of this study comprises the following elements:

Prompt-only use of modern open-weight LLMs:

The study evaluates LLaMA 3.1:8B and Qwen3:8B for binary bug severity prediction without task-specific parameter fine-tuning.

Comparison of zero-shot and few-shot prompting:

The study quantitatively examines whether providing a small set of labeled examples improves the performance of each LLM.

Unified comparison with multiple baseline families:

The LLMs are compared with SVM, Multinomial Naive Bayes, Random Forest, and LSTM under a common severity-label mapping and test protocol.

Comparison of two conventional text-representation techniques:

The experiments assess TF-IDF and Word2Vec representations with the classical classifiers.

Additional robustness evaluation:

The models are evaluated on a separately generated set of 200 synthetic bug reports to measure performance under a different textual distribution.

Reduced task-specific training requirements:

The study evaluates whether pretrained LLMs can classify raw bug-report text with substantially less preprocessing and no supervised parameter optimization.

Model-specific prompting findings:

The results indicate that LLaMA 3.1 performs better under zero-shot prompting, whereas Qwen3 benefits more substantially from few-shot demonstrations.

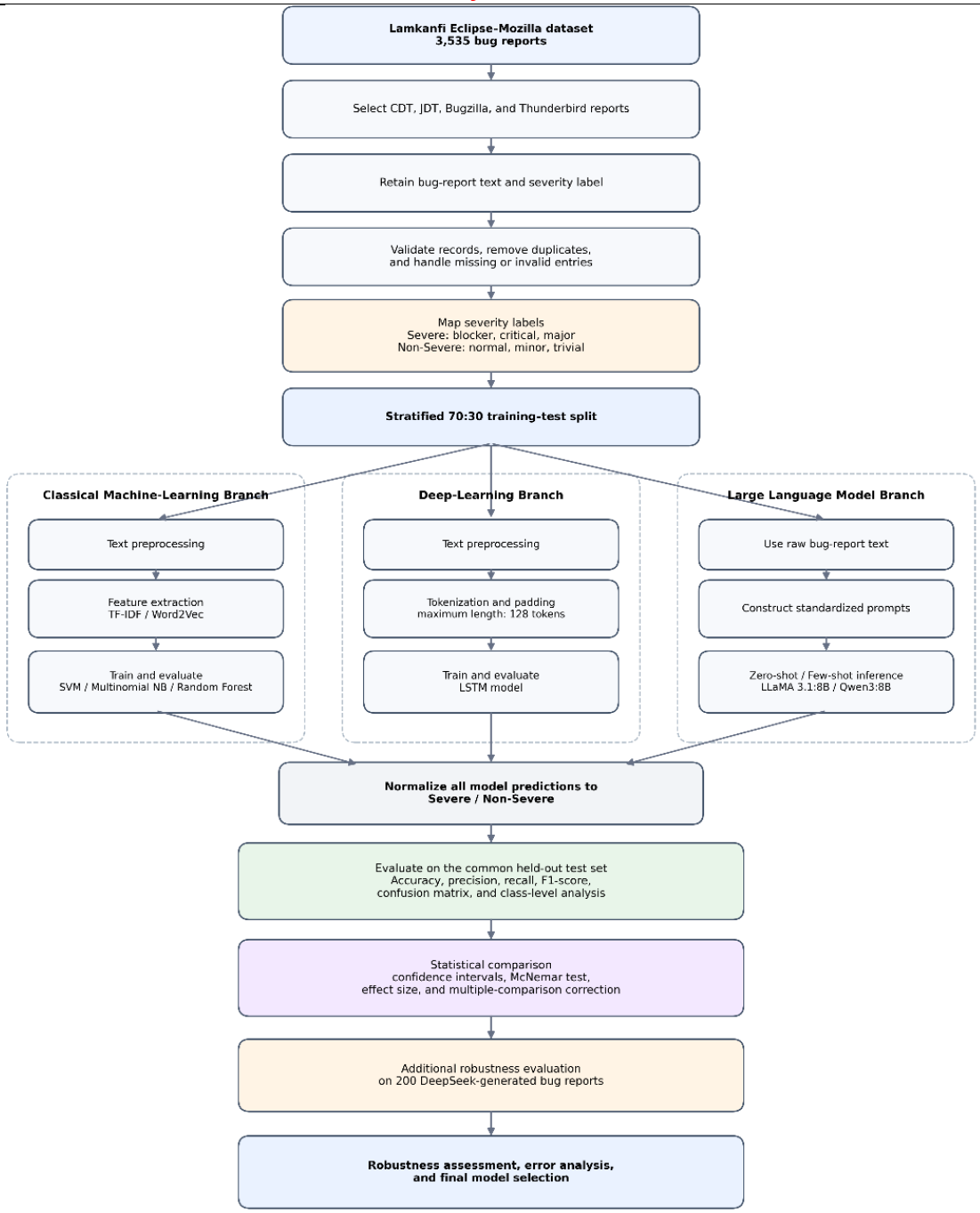

Figure 1. Workflow of the proposed bug severity prediction methodology

Material and Methods:

This section presents the research methodology used to assess and compare the effectiveness of traditional machine learning algorithms, a deep learning model, and emerging large language models in classifying bug reports as severe or non-severe. The methodology is organized into four distinct experiments; each is designed to systematically assess different approaches and identify the most accurate and efficient model for bug severity prediction. The experiments are organized as follows:

Dataset:

We reuse an open-access data set created by [33]. They explored the Bugzilla repository to collect bug reports from the Eclipse and Mozilla projects, filtering out duplicates and enhancement requests. Both the Eclipse and Mozilla projects each include

four different software products. From this dataset, we selected bug reports related to four open-source products: CDT, JDT, Bugzilla, and Thunderbird.

We focused on two key attributes of each bug report: the summary, which describes the issue, and the severity, which indicates how urgent the issue should be resolved. In total, the final dataset contains 3,535 bug reports.

As illustrated in Figure 1, the proposed workflow provides a unified experimental procedure for comparing classical machine-learning models, LSTM, LLaMA 3.1, and Qwen3 under consistent dataset and evaluation conditions.

Experiment-I (Proposed Approach):

Testing & evaluation of open-weight large language models (Llama3.1:8b and Qwen3:8B) with zero and few-shot learning techniques. An overview of the proposed approach is presented in Figure 2. This approach aims to evaluate the effectiveness of two Large Language Models (LLMs), namely LLaMA 3.1:8 B and Qwen3:8B, in automatically classifying software bug reports into two predefined categories: severe and non-severe. The classification is conducted using prompt engineering techniques under both zero-shot and few-shot learning scenarios. The objective is to identify which of the two models is more suitable and reliable for the binary classification task, particularly for automating severity prediction with minimal reliance on manual feature engineering or large labeled datasets.

LLaMA3.1:8B.

LLaMA3.1:8B is an open-weight large language model, developed by Meta. It is part of the LLaMA 3 family, containing approximately 8 billion parameters, released in July 2024, and designed to perform a wide range of NLP tasks with limited labeled demonstrations and generalization capabilities. LLaMA supports zero-shot and few-shot learning, making it suitable for a variety of tasks, such as text classification, without requiring extensive fine-tuning.

Qwen3:8B.

Like LLaMA 3.1, Qwen3 is also part of the advanced large language models released in mid-2024 as part of the Tongyi Qianwen series by Tongyi Lab, a research unit under Alibaba Group. Qwen3 supports both zero-shot and few-shot learning, enabling it to generalize effectively even with limited labeled data.

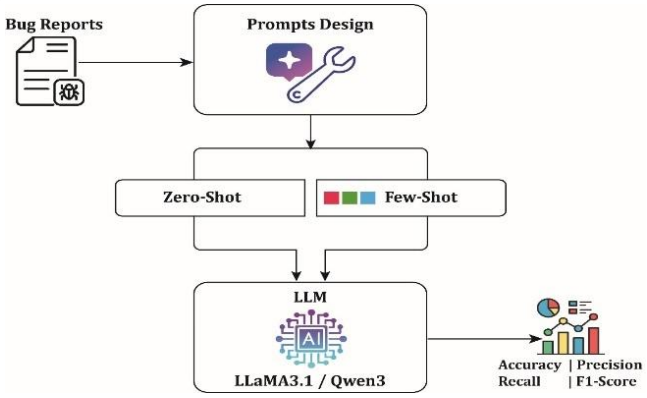

Figure 2. Overview of the Proposed Approach

The dataset has been divided into a training set (70%) and a test set (30%). The training set is used to train all the traditional machine learning algorithms, while the testing set is used for their assessment. In this study, we collected and stored bug reports for Mozilla and Eclipse from the Bugzilla repository, where each report is assigned a severity level ranging from trivial and minor to normal, major, critical, and blocker. In our proposed approach, we apply a binary classification method to predict the severity of bug reports. We classify severity levels into two categories: trivial, minor, and normal are considered non-severe, while major, critical, and blocker are considered severe.

Prompt Engineering:

Prompt engineering is the process of designing and refining the input given to a Large Language Model (LLM) to guide it toward producing accurate, relevant, or desired outputs for a specific task. It involves crafting prompts (instructions or questions) to maximize the model's performance without additional training.

Zero-shot Learning:

In Zero-shot learning, in zero-shot learning, the model performs a task without receiving task-specific examples in the prompt. It depends exclusively on its pre-trained knowledge to make predictions.

Few-shot Learning:

In Few-shot learning, the model is given a few labeled examples (e.g., 3 to 5) in the prompt before asking it to perform the task. These examples sometimes help guide the model's response more accurately.

Procedure:

The test dataset was first prepared by ensuring that all entries were properly formatted and accurately labeled for the classification task.

To enable zero-shot and few-shot learning, customized prompts were designed.

These customized prompts were then passed through locally installed open-weight LLaMA3.1 and Qwen3 models.

The performance of both models was evaluated using standard metrics.

To assess the effectiveness of both models, a comparative study was conducted across zero-shot and few-shot settings.

At the end, on the basis of comprehensive results, the most efficient model and learning setup were identified to accurately classify bug reports into two predefined binary classes of severe and non-severe.

Experiment-II:

Training, testing, and assessment of Classical ML models with different word embedding methods, as shown in Figure 3. The objective of this experiment is to assess the efficiency of three different traditional machine learning models. The experiment will further examine two types of textual embedding techniques and compare the results of all models using standard performance evaluation metrics.

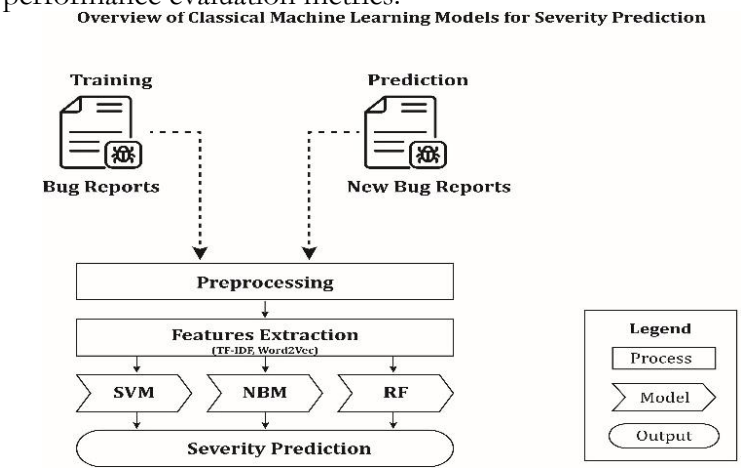

Figure 3. Overview of Classical Machine Learning Models

Support Vector Machine (SVM):

SVM is a well-known machine learning algorithm that uses supervised learning for performing classification tasks [34]. It identifies a hyperplane in an n-dimensional space that perfectly separates data points into different labels. It performs equally well on both simple and complex data and is popular for classifying text and images.

Random Forest (RF):

A machine learning algorithm that functions like a group of decision trees [35]. Each tree provides a result, and the forest chooses the top answer based on majority voting. It is popular due to various factors like handling missing data well, accuracy, and to some extent, avoiding overfitting.

Naıve Bayes Multinomial (NBM):

A simple probabilistic classification algorithm commonly used for textual data classification based on Bayes' theorem [36]. It works by calculating probabilities of words in different categories and then guessing which category a new text belongs to.

Dataset:

The same dataset has been used throughout the experiments. The training set (70%) is used to train the models, while the testing set (30%) is used to evaluate their effectiveness.

TF-IDF (Term Frequency-Inverse Document Frequency):

TF-IDF is a statistical measure used to evaluate the importance of a word in a document relative to a collection of documents. It helps in highlighting words that are significant in a particular document while downscaling common words across the corpus.

Word2Vec:

Word2vec is a Natural Language Processing (NLP) method for producing vector representations of words. These vectors provide information about a word's meaning based on the surrounding words. By modeling text from a large corpus, the Word2Vec method approximates these representations.

Procedure:

Initially, the dataset was passed through the preprocessing stage to convert textual descriptions into a well-structured format, including text cleaning, tokenization, and lemmatization.

The dataset was then split into 70% for training and 30% for testing and evaluating the models.

Using TF-IDF and Word2Vec word embedding techniques, textual features were extracted from the training data.

Using both of the word embedding techniques, three distinct classical ML models comprising NBM, SVM, and RF were trained on 70% of the data.

Individual models were assessed using a 30% test dataset and measured their performance in terms of accuracy, precision, recall, and F1-score.

Finally, results were evaluated across models and embedding schemes to identify the best-performing combination.

Experiment-III:

Testing and evaluation of a deep learning model, Long-Short Term Memory (LSTM). This experiment focuses on the training and evaluation of Long Short-Term Memory (LSTM) for classifying bug reports into two binary classes.

Long Short-Term Memory (LSTM):

LSTM is a kind of recurrent neural network that can store information over long time intervals, which is then used for future calculations, e.g., classification tasks [37][38]. LSTM model can be used for a variety of tasks, including but not limited to language translation, video analysis, and text-to-speech.

Procedure:

Initially, the dataset is preprocessed to remove noise and transform the text into a structured format.

The data set is subsequently divided into a training set (70%) and a test set (30%) to facilitate model training and performance evaluation.

The model is trained at this stage.

The model's performance is assessed on the test set using standard evaluation metrics, including accuracy, precision, recall, and F1-score.

Experiment-IV:

Comparative study of the results from experiments I, II, and III to identify the best-performing model for automating severity prediction of bug reports. This experiment aims to assess and compare the performance of three techniques in classifying software bug reports into two binary classes: severe and non-severe. Some of these approaches include Large Language Models (reviewed in Experiment I), traditional machine learning algorithms (examined in Experiment II), and a Long Short-Term Memory (LSTM) model (studied in Experiment III). Analyzing the results' accuracy, precision, recall, and F1-score allows us to see which approach is most effective for automatically predicting the severity of bug reports.

Procedure:

Gather the results from Experiments I, II, and III, listing accuracy, precision, recall, and F-measure for all the models.

Conduct performance comparison, considering how LLMs (LlaMA 3.1 and Qwen3) work when compared with classical machine learning models (SVM, NBM, RF) and the LSTM.

Evaluate and find out the model that performs best in this task and handles the bug reports.

Analyze the outcomes and their relevance to the field of software engineering, especially in predicting the severity of bug reports.

Implementation:

This section describes the entire implementation workflow, which is essential for executing the experiments for the bug report classification task. The key objective of this research is to predict and distinguish between severe and non-severe bug reports by using a variety of methodologies. This section also covers tools and technologies used in the study.

Since the testing is carried out in Python, it is important to note that version 3.12.7 was used for its extensive ecosystem of tools and libraries, including deep learning, machine learning, and natural language processing (NLP), which are essential to this work. Numerous libraries are used in the implementation process, each with a specific function in the workflow. Scikit-learn is used to implement and evaluate classical ML models (SVM, Naive Bayes, Random Forest) on bug report text features**.** PyTorch facilitates the integration of LLM-based severity classification approaches (LLaMA, Qwen3) and future deep learning extensions. NLTK supports basic natural language processing tasks, such as tokenization and text cleaning, to prepare bug report descriptions. Pandas is used to manage bug report datasets, enabling efficient data handling, preprocessing, and storage of severity predictions, while OLLaMA provides a seamless interface for utilizing LLaMA and Qwen3 LLMs through prompt engineering to classify bug report severity. NumPy offers foundational numerical operations for handling text embedding vectors and calculating evaluation metrics.

To ensure efficient execution of classical ML models, deep learning experiments, and Large Language Model (LLM)-based severity classification, a robust hardware environment was employed with the following specifications: an ASUS 12th Generation laptop with a 28 GB Shared GPU and 40 GB RAM, providing dependable computational resources for implementation and experimentation. Python environments are managed using Conda in Jupyter Notebook to ensure dependency compatibility. Ollama is integrated to interact smoothly with and utilize the locally installed LLaMA 3.1:8 B and Qwen3:8B models for prompt engineering and severity prediction tasks.

Experiment-I, Evaluating LLMs (LLaMA3.1 and Qwen3):

To assess the effectiveness of LLMs in bug report classification tasks, we have chosen publicly available open-weight Meta LLaMA3.1:8B and Qwen3:8B, both with 8 billion parameters, downloaded and installed on a local system through Ollama, an open-

source framework that lets users download and run large language models like LlaMA, Qwen, and DeepSeek, etc., locally. Both models were evaluated using the same zero-shot and few-shot prompt engineering techniques.

Zero-Shot Prompt:

A prompt that asks the model to perform a task without providing any prior examples is shown in Figure 4.

```
# System Prompt for Bug Severity Classification
system_prompt = """
You are an expert in bug analysis and classification, skilled in determining the severity of software bugs.
Follow the instructions below to analyze and classify the given bugs.

1. **Definitions**:
   - **Severe Bug (1)**: A bug that significantly impacts system functionality, user experience, or security.
   - **Not Severe Bug (0)**: A minor issue with little to no impact on functionality.

2. **Task**:
   - Classify each bug as either **1 (Severe)** or **0 (Not Severe)**.
   - Provide a short explanation for your decision.
```

Figure 4. Example of zero-shot prompt

Few-Shot Prompt:

A prompt that includes a limited number of prior examples of the task to guide the model in generating the desired output is shown in Figure 5.

```
1. **Definitions**:
   - **Severe Bug (1)**: A bug that significantly impacts system functionality, user experience, or security.
   - **Not Severe Bug (0)**: A minor issue with little to no impact on functionality.

   **Examples**:
   - Bug Description: "App crashes when user clicks 'Submit'"
     Classification: 1
   - Bug Description: "Color mismatch in the settings page"
     Classification: 0
   - Bug Description: "Login form not working on mobile devices"
     Classification: 1

2. **Task**:
   - Classify each bug as either **1 (Severe)** or **0 (Not Severe)**.
   - Provide a short explanation for your decision.
```

Figure 5. Example of a few-shot prompt

Experiment-II, Evaluating NBM, SVM, and RF using TF-IDF and Word2Vec:

This experiment evaluates the performance of three classical ML models, including SVM, NBM, and RF, by using TF-IDF and Word2Vec textual embedding techniques.

Data Preprocessing: Before feeding the data into ML models, we clean and structure the bug report descriptions to achieve better, more accurate results. We have used the Natural Language Toolkit (NLTK) [39] and TextBlob [40] to implement natural language processing techniques.

Text Cleaning: Bug reports often have inconsistencies and unnecessary characters and irrelevant information.

The following preprocessing steps were applied:

Lowercasing: Everything becomes lowercase for consistency.

Removing Special Characters: Clean out punctuation, symbols, and anything not a word or number.

Stop Word Removal: Remove common words (like "the", "is", "at") that don't add value for classification.

Tokenization and Lemmatization:

Tokenization is essential in NLP tasks as it breaks down text into organized word sequences that can be easily analyzed. For example," The developers were fixing the bugs

more quickly than ever.” becomes [” The”,” developers”,” were”,” fixing”,” the”,” bugs”,” more”,” quickly”,” than”,” ever”,”.”] after tokenization. Lemmatization simplifies words by converting their comparative and superlative forms to their base or root form, so words like "fixing" become "fix", and "developers" become "developer”. Making it easier for models to classify the text, as shown in Figure 6.

```python
# Text preprocessing function
def preprocess_text(text):
    # Lowercase the text
    text = text.lower()
    # Remove special characters and numbers
    text = re.sub(r'[^a-z\s]', '', text)
    # Remove stop words
    stop_words = set(stopwords.words('english'))
    words = text.split()
    filtered_words = [word for word in words if word not in stop_words]
    # Lemmatize words
    lemmatizer = WordNetLemmatizer()
    lemmatized_words = [lemmatizer.lemmatize(word) for word in filtered_words]
    return ' '.join(lemmatized_words)
```

Figure 6. Data Preprocessing steps

Text Embeddings:

After the tokenization and lemmatization process, it’s time to convert bug report text into a format that ML models can understand; we have used different embedding techniques. To feed the textual descriptions of bug reports into an ML classifier, the words are converted into fixed-length numerical vectors. This transformation is achieved using the Word2Vec skip-gram model, originally introduced by [18]. TF-IDF assigns weights to words based on how often they appear in a report compared to other reports, helping us highlight the most relevant terms. Word2Vec represents words as numerical vectors, capturing their meaning and relationships with other words in the text, as shown in Figure 7.

```python
# Load the datasets
train_df = pd.read_csv('Bug Datasets/Main-Data-Set_train.csv')
test_df = pd.read_csv('Bug Datasets/Main-Data-Set_test.csv')

# Display the first few rows of the training data
print("Training Data Sample:")
print(train_df.head())

# Ensure the required columns are present
if 'Short Description' not in train_df.columns or 'Is Severe?' not in train_df.columns:
    raise ValueError("Both 'Short Description' and 'Is Severe?' columns must be present in the dataset.")

# Extract features and labels for training and testing
X_train = train_df['Short Description'].apply(preprocess_text)
y_train = train_df['Is Severe?']
X_test = test_df['Short Description'].apply(preprocess_text)
y_test = test_df['Is Severe?']

# Feature extraction using TF-IDF
vectorizer = TfidfVectorizer(max_features=1000)
X_train_tfidf = vectorizer.fit_transform(X_train)
X_test_tfidf = vectorizer.transform(X_test)

# Models to evaluate
models = {
    "NaiveBayes": MultinomialNB(),
    "SVM": SVC(probability=True, random_state=RANDOM_SEED),
    "RandomForest": RandomForestClassifier(n_estimators=100, random_state=RANDOM_SEED)
}
```

Figure 7. Implementation of ML Models

Experiment-III, Implementation of LSTM:

This experiment assesses the effectiveness of a deep learning approach, specifically the LSTM model. Like traditional ML models, the text data was cleaned, lowercased, special characters removed, stop-words eliminated, and lemmatized to ensure consistency. The preprocessed text was tokenized and padded to a uniform sequence length (128 tokens) to meet the input requirements of the LSTM model.

Experiment IV, Integrated evaluation of overall results:

All models, including LLMs (LLaMA 3.1 and Qwen 3), traditional ML algorithms (SVM, NBM, RF), and the LSTM model, were evaluated on the same test dataset. This ensured a consistent and fair basis for comparison of their classification performance, as shown in Figure 8.

```
    return {
        "Accuracy": accuracy,
        "Precision": precision,
        "Recall": recall,
        "F1-Score": f1,
        "Detailed Report": class_report_df
    }

# Main execution
if __name__ == "__main__":
    df_bugs = pd.read_csv('FinalTest_dataset.csv')

    df_results = classify_bugs(df_bugs)

    if df_results.empty:
        print("No valid classifications found. Please check logs.")
    else:
        metrics = evaluate_metrics(df_results['Is Severe?'], df_results['Predicted_Is_Severe'])
        print(metrics)

        metrics_df = pd.DataFrame([metrics]).drop(columns="Detailed Report")
        detailed_metrics_df = metrics["Detailed Report"]

        with pd.ExcelWriter('BugSeverityPrediction_Llama3_Local.xlsx') as writer:
            df_results.to_excel(writer, sheet_name='Results', index=False)
            metrics_df.to_excel(writer, sheet_name='Metrics', index=False)
            detailed_metrics_df.to_excel(writer, sheet_name='Detailed Metrics', index=True)

        print(f"\n✅ Saved results to BugSeverityPrediction_Llama3_Local.xlsx")
```

Figure 8. Metrics computation

Results and Discussion:

This section discusses the experimental findings from evaluating different approaches to predicting the severity of software bug reports. The main aim is to assess how well a model performs in terms of accuracy, efficiency, and ability to generalize when classifying bug reports as either severe or non-severe.

Experiment-I: Evaluating LLMs (LLaMA3.1 and Qwen3) with zero-shot and few-shot learning:

This experiment investigates the use of LLaMA3.1 and Qwen3 to automatically classify bug reports into severe and non-severe categories. Unlike traditional models, these LLMs require no prior training and can handle diverse, unstructured data using zero-shot and few-shot techniques. The goal is to assess their performance and generalizability in real-world bug report classification tasks.

RQ1: To what extent can the LLaMA3.1 model accurately classify software bug reports using zero-shot learning, and how does its performance compare under few-shot learning?

LLaMA3.1 Performance:

To explore the capabilities of the LLaMA 3.1 model for classifying software bug reports, we evaluated its performance under both zero-shot and few-shot learning conditions. The classification task involved predicting whether bug reports were Severe or Non-Severe based solely on their textual descriptions. The evaluation was conducted using four standard performance metrics: accuracy, precision, recall, and F1-score.

Table 1. Performance metrics of LLaMA 3.1:8B using Zero-Shot and Few-Shot learning on the test dataset

Model / Technique	Accuracy (%)	Precision (%)	Recall (%)	F1-score (%)
LLaMA 3.1:8B (Zero-Shot)	94.4	98.5	91.1	94.6
LLaMA 3.1:8B (Few-Shot)	93.7	95.1	92.9	94.0

Zero-Shot Learning Performance:

The LLaMA3.1:8B model demonstrated remarkable performance in the zero-shot setting, achieving 94.4% accuracy, indicating a strong ability to classify bug reports without prior training. It exhibited an exceptionally high precision of 98.5%, suggesting that when it identified a bug report as" severe," it was correct nearly all the time. The recall was slightly lower at 91.1%, meaning the model missed a few actual severe cases. Nevertheless, the F1-score, which balances precision and recall, stood at a strong 94.6%, confirming the model's robust zero-shot capabilities.

Few-Shot Learning Performance:

When a few labeled examples were introduced (few-shot learning), the model's recall improved from 91.1% to 92.9%, indicating it became better at detecting more actual severe bug reports. However, this gain in recall came with a slight drop in accuracy to 93.7% and precision to 95.1%. The F1-score in the few-shot setting was 94.0%, slightly lower than in the zero-shot setting, as shown in Figure 9.

Figure 9. Performance of LLaMA3.1 with Zero Shot and Few-shot learning on test dataset.

These results suggest that zero-shot learning with LLaMA3.1 is not only effective but slightly more balanced overall. The model performs well out of the box without the need for fine-tuning or labeled training data. Although few-shot learning helped in increasing recall, it slightly compromised precision and accuracy. This trade-off highlights the model's sensitivity to prompt variation and the limited context provided by few-shot examples.

LLaMA3.1 shows strong generalization and high effectiveness in both zero-shot and few-shot learning settings, with zero-shot slightly outperforming in terms of the balance between precision and recall. This demonstrates the model's viability as a scalable, low-maintenance solution for severity classification of software bug reports, especially in scenarios where labeled data is limited or unavailable.

RQ2: How does the performance of LLaMA3.1 vary between known and unseen bug report datasets, and what does this reveal about its generalization capabilities?

Performance on Deep Seek-generated synthetic bug report dataset:

After evaluating LLaMA3.1 on the original 30% test set, where it showed strong performance in both zero-shot and few-shot settings, an additional evaluation was conducted and test how well it generalizes to completely unseen data. For this, a new dataset of 200 distinct bug reports was created using DeepSeek, with the same structure but different entries. This experiment aimed to assess whether LLaMA3.1 could maintain its effectiveness when exposed to new and diverse bug descriptions.

Table 2. Performance metrics of LLaMA 3.1:8B using Zero-Shot and Few-Shot learning on the DeepSeek-generated dataset

Model / Technique	Accuracy (%)	Precision (%)	Recall (%)	F1-score (%)
LLaMA 3.1:8B (Zero Shot)	97.99	97.06	100.00	98.51
LLaMA 3.1:8B (Few- shot)	95.56	93.65	100.00	96.72

Zero-Shot Performance:

On the DeepSeek dataset, LLaMA3.1 in the zero-shot setting delivered even better results than before. It achieved 97.99% accuracy, up from 94.4% on the original test set. The precision remained high at 97.06%, while recall improved to a perfect 100%, indicating that the model correctly identified all severe bug reports. Its F1-score also improved to 98.51%, indicating a more balanced and confident performance. This improvement shows that LLaMA3.1 not only handles previously seen data well but also performs consistently, even better on unseen, independently generated bug reports. This strongly supports its ability to generalize across varied bug report content without requiring retraining or extensive data preparation.

Few-Shot Performance:

In the few-shot setting, the model also performed well, achieving 95.56% accuracy, slightly lower than in the zero-shot case. Precision dropped slightly to 93.65%, while recall remained at 100%, as in the earlier test. The F1-score was 96.72%, still excellent but slightly lower than the zero-shot performance on the same dataset.

Interestingly, the trend observed earlier on the original test data remains consistent: few-shot learning slightly boosts recall but can lead to a small drop in precision and accuracy. On both datasets, zero-shot learning showed more balanced results, making it the stronger general-purpose choice for this task.

Comparative Insights:

When comparing results across both datasets, it's clear that LLaMA3.1 generalizes exceptionally well. Its performance on the DeepSeek dataset not only holds up but actually improves in several metrics, especially recall and F1-score. This confirms that the model can maintain high performance in real-world scenarios, even when the bug report data varies significantly in content and phrasing**.**

These findings support the model's practical use in dynamic environments where labeled data may be limited and bug reports are constantly evolving. Zero-shot learning with LLaMA3.1, in particular, emerges as a highly reliable approach for scalable bug report classification.

RQ3: To what extent can the Qwen3 model accurately classify software bug reports using zero-shot learning, and how does its performance compare under few-shot learning?

Qwen3 Performance:

To answer RQ3, we tested Qwen3:8B in both zero-shot and few-shot learning settings using the original 30% test dataset. The goal was to see how well the model could classify bug reports as either Severe or Non-Severe based on their descriptions.

Table 3. Performance metrics of Qwen3:8B using Zero-Shot and Few-Shot learning on the test dataset

Model / Technique	Accuracy (%)	Precision (%)	Recall (%)	F1-score (%)
Qwen3:8B (Zero Shot)	92.9	100	87.0	93.0
Qwen3:8B (Few-shot)	98.6	100	97.5	98.7

Zero-Shot Learning:

In the zero-shot setting, where the model was not shown any training examples beforehand, Qwen3 performed well. It achieved an accuracy of 92.9%, meaning most predictions were correct. The model's precision was 100%, which is excellent; it means that every time it predicted a report as" severe," it was always right. However, its recall was

87.0%, showing that it missed some actual severe reports. The overall F1-score was 93.0%, indicating good performance, though slightly imbalanced due to the lower recall.

Few-Shot Learning:

In the few-shot setting, where we gave the model a small number of labeled examples before testing, Qwen3's performance improved significantly. It reached a high preductive accuracy of 98.6%, and precision remained perfect at 100%. More importantly, recall improved to 97.5%, meaning it correctly identified nearly all severe bug reports. The F1-score also increased to 98.7%, showing strong balance and consistency in its predictions.

From this comparison, we can see that Qwen3 works well without training (zero-shot), as shown in Figure 10, but it becomes even more effective with just a few labeled examples (few-shot). While it was already highly precise in the zero-shot setting, the few-shot approach helped it catch more of the severe cases it missed earlier. This pattern is different from LLaMA3.1, which performed slightly better in the zero-shot case. Qwen3, by contrast, clearly benefits from few-shot learning.

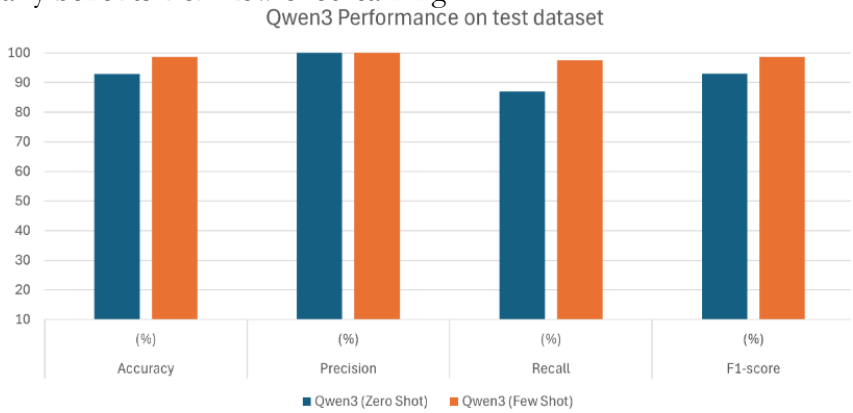

Figure 10. Performance of Qwen with Zero Shot and Few-shot learning on the test dataset

RQ4: How does the performance of Qwen3 vary between known and unseen bug report datasets, and what does this reveal about its generalization capabilities?

Qwen3 Performance on DeepSeek-Generated Dataset:

To address RQ4, the Qwen3:8B model was tested on the DeepSeek-generated dataset, using both zero-shot and few-shot learning settings. This allowed us to assess the model's ability to generalize beyond the original test data.

Table 4. Performance metrics of Qwen3:8B using Zero-Shot and Few-Shot learning on the DeepSeek-generated dataset

Model	Accuracy (%)	Precision (%)	Recall (%)	F1-score (%)
Qwen3:8B (Zero-Shot)	96.8	95.5	98.2	96.8
Qwen3:8B (Few-shot)	97.9	96.3	98.8	97.5

Zero-Shot Learning:

In the zero-shot setting, Qwen3:8B achieved an accuracy of 96.8%, noticeably higher than the 92.9% it scored on the original test set. Precision was slightly reduced to 95.5%, but recall improved significantly to 98.2%, indicating the model was more effective at identifying all severe bug reports. The F1-score also increased to 96.8%, showing a stronger balance between precision and recall than previously observed. Such enhancement indicates that Qwen3:8B generalizes well to unseen data, particularly in the absence of further training. The increase in recall score is notably significant in the severity classification task, where overlooking a significant issue can negatively affect software reliability.

Few-Shot Learning:

Qwen3's performance was also high in the few-shot setting. This achieved an accuracy of 97.9%, a precision of 96.3%, and a recall of 98.8%, yielding a remarkable F1-

score of 97.5. Such scores are well-balanced and yield reliable classification outcomes. Relative to its previous performance, this demonstrates that Qwen3:8B continues to benefit from exposure to labeled examples, even at the smallest proportions.

Comparative Insights:

When comparing both datasets:

Qwen3:8B recall score improved greatly on unseen data both in zero-shot and in few-shot settings.

The model showed considerable precision and accuracy on a consistent basis and when the model was introduced into unseen data.

These findings attest to the strong generalizing capability of the model, particularly with the use of few-shot learning.

Performance Comparison of LLaMA3.1 and Qwen3:

Based on the analysis of the results from both models, it was found that LLaMA3.1 was well suited to the zero-shot learning setting and thus most appropriate for classification tasks where labeled data was not required. At the same time, with few labeled examples, Qwen3 models achieved improved performance through few-shot learning. Experimental results reveal that both models perform well at automating severity prediction for bug reports. However, their performance varies depending on the selection of prompt-engineering techniques. Without any training, LLaMA3.1 generalizes better, making it more efficient in zero-shot settings, while Qwen3 shows improved adaptability when some contextual direction is provided.

Experiment-II: Evaluating NBM, SVM, and RF using TF-IDF and Word2Vec:

The experiment examines how well three classical ML models, including NBM, SVM, and RF, with two word embedding techniques, can classify bug reports into two binary classes. To carry out this evaluation, all models were trained on 70% of the labeled dataset and tested on the remaining 30% as the test set. Additionally, to assess the generalization ability, each model was evaluated on a separate, unseen dataset generated through DeepSeek.

RQ5: How effective are classical machine learning models in classifying software bug reports into severity categories when using TF-IDF and Word2Vec embedding techniques?

Analysis of Classical Models using TF-IDF:

As shown in Table 5, the performance of classical machine learning models using TF-IDF was moderate across all evaluation metrics. Naïve Bayes Multinomial (NBM) achieved an accuracy of 71.2%, with precision at 73.0%, recall at 74.5%, and an F1-score of 73.7%. This reflects a fairly balanced performance, with slightly better recall, indicating the model was reasonably effective at identifying severe bug reports.

Table 5. Performance metrics of Classical ML Models using TF-IDF Embedding on the test dataset

Model / Technique	Accuracy (%)	Precision (%)	Recall (%)	F1-score (%)
NBM (TF-IDF)	71.2	73.0	74.5	73.7
SVM (TF-IDF)	71.0	73.8	72.1	72.9
RF (TF-IDF)	72.0	73.8	75.2	74.5

The Support Vector Machine (SVM) produced similar results: 71.0% accuracy, 73.8% precision, 72.1% recall, and an F1-score of 72.9%. The slightly lower recall suggests that the SVM may have missed a few severe cases while maintaining a consistent balance. Random Forest (RF) With a precision of 73.8%, an F1-score of 74.5%, the best recall of 75.2%, and the greatest accuracy of 72.0%, Random Forest (RF) demonstrated a little improvement over NBM and SVM. This reveals that RF outperformed other classical ML algorithms in detecting severe bug reports while keeping false positives under control.

Analysis of Classical Models using Word2Vec:

Table 6 and Figure 11 presents the performance of classical machine learning models, including Naïve Bayes Multinomial (NBM), Support Vector Machine (SVM), and Random Forest (RF), when paired with Word2Vec embeddings for the binary classification of bug report severity.

Table 6. Performance metrics of Classical ML Models using Word2Vec Embedding on the test dataset

Model / Technique	Accuracy (%)	Precision (%)	Recall (%)	F1-score (%)
NBM (Word2Vec)	54.2	54.2	100	70.3
SVM (Word2Vec)	63.1	64.9	69.5	67.1
RF (Word2Vec)	60.4	62.3	68.1	65.1

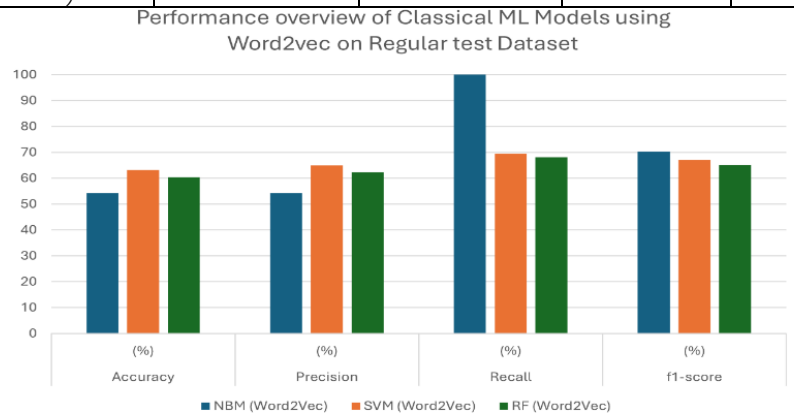

Figure 11. Performance of the Classical ML Models with Word2Vec on the Regular Test Dataset

Naïve Bayes Multinomial (NBM). As shown in Figure 5.3, NBM achieved the lowest accuracy, 54.2%, indicating that it struggled to distinguish between severe and non-severe bug reports using Word2Vec features. While it achieved a perfect recall of 100%, meaning it successfully identified all severe reports, its precision was very low at 54.2%, resulting in an F1-score of 70.3%. This suggests that while NBM was aggressive in labeling reports as severe, it also misclassified many non-severe ones, leading to a high number of false positives.

Support Vector Machine (SVM) performed moderately better, with an accuracy of 63.1%, precision of 64.9%, and recall of 69.5%, yielding an F1-score of 67.1%. These metrics reflect a more balanced but still limited performance, likely due to challenges in capturing contextual semantics through averaged Word2Vec vectors. Random Forest (RF) followed a similar trend, achieving 60.4% accuracy, 62.3% precision, and 68.1% recall, resulting in an F1-score of 65.1%. Like SVM, it demonstrated a somewhat balanced output, but its performance remained below expectations for practical deployment.

RQ6: How effectively do classical machine learning models retain performance on unseen bug reports when tested with TF-IDF and Word2Vec embeddings?

Performance of Classical Models on DeepSeek-Generated Dataset:

To evaluate whether classical machine learning models can maintain performance on unseen bug report data, we tested them on a newly generated dataset using DeepSeek. These models were previously trained and tested using TF-IDF and Word2Vec embeddings on a known dataset. Here, we aim to assess how well they generalize and whether there is evidence of a performance drop or overfitting.

Performance with TF-IDF:

When evaluated on the DeepSeek-generated dataset, the accuracy of all classical models declined slightly compared to their original test set performance: Naïve Bayes Multinomial (NBM) dropped from 71.2% to 66.33%, though it still retained good

performance, with an F1-score of 76.33% and strong recall of 81.82%. SVM went from 71.0% to 64.82% accuracy, and RF from 72.0% to 64.32% accuracy, both showing a noticeable performance reduction. While the decline isn't severe, it reflects the fact that these models are somewhat sensitive to new data. Still, their performance with TF-IDF remains fairly consistent, showing moderate generalizability without major overfitting.

Table 7. Performance metrics of Classical ML Models using TF-IDF Embedding on DeepSeek-generated dataset

Model / Technique	Accuracy (%)	Precision (%)	Recall (%)	F1-score (%)
NBM (TF-IDF)	66.33	71.52	81.82	76.33
SVM (TF-IDF)	64.82	72.14	76.52	74.26
RF (TF-IDF)	64.32	71.03	78.03	74.37

Using Word2Vec Embeddings: The drop in accuracy became more noticeable when Word2Vec embeddings were used. NBM (Word2Vec) retained its accuracy at 66.33%, but at the cost of balance: recall reached 100%, while precision fell to 66.33%, meaning the model predicted nearly everything as severe. While the F1-score of 79.76% looks strong, it hides this skewed behavior, as shown in Figure 12.

Table 8. Performance metrics of Classical ML Models using Word2Vec Embedding on DeepSeek dataset.

Model / Technique	Accuracy (%)	Precision (%)	Recall (%)	F1-score (%)
NBM (Word2Vec)	66.33	66.33	100.00	79.76
SVM (Word2Vec)	46.73	62.04	50.76	55.83
RF (Word2Vec)	50.25	64.86	54.55	59.26

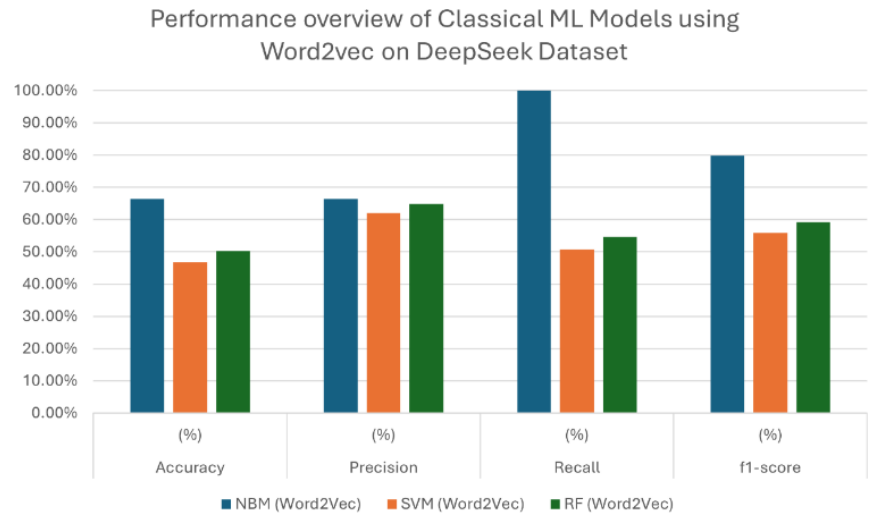

Figure 12. Performance of the Classical ML Models with Word2Vec on the DeepSeek Dataset

The substantial performance reduction suggests that SVM and RF learned too much from the training data and struggled to handle new bug reports. Even though Word2Vec is good at capturing word meanings, these models had trouble making sense of that kind of data, probably because they aren't built to work well with the way Word2Vec represents text.

Experiment-III: Evaluation of an LSTM:

The Long Short-Term Memory (LSTM) model was trained on 70% of the dataset and tested on the remaining 30%, similar to the setup used for classical machine learning models. Since LSTM can natively handle sequential input, it was trained without external embeddings such as TF-IDF or Word2Vec. The model was also evaluated on the DeepSeek-generated dataset to assess its generalization ability and to check for signs of overfitting.

RQ7: How effective is the LSTM model in classifying software bug reports by severity based on sequential learning from textual data?

Table 9. Performance metrics of LSTM Model on test dataset

Model / Technique	Accuracy (%)	Precision (%)	Recall (%)	F1-score (%)
LSTM	71.6	76.6	68.6	72.3

Performance on the Original Test Dataset:

On the original test dataset, LSTM delivered moderate results: accuracy of 71.6%, precision of 76.6%, recall of 68.6%, and F1-score of 72.3%. This shows that LSTM was reasonably effective at learning from the training data and capturing sequential patterns in the bug report text. Its relatively high precision suggests that when the model predicted a bug as severe, it was usually correct. However, the recall was slightly lower, indicating it missed some severe cases. LSTM performs competitively against classical TF-IDF models, achieving slightly higher precision and a balanced F1-score.

RQ8: To what extent can the LSTM model generalize to unseen bug report data, and how does its performance compare to classical models and LLMs?

Table 10. Performance metrics of LSTM Model on DeepSeek-generated dataset

Model / Technique	Accuracy (%)	Precision (%)	Recall (%)	F1-score (%)
LSTM	52.76	70.65	49.24	58.04

Performance on the DeepSeek Dataset:

When tested on the DeepSeek-generated dataset, LSTM's performance dropped significantly: Accuracy fell to 52.76%, Recall dropped to 49.24%, and F1-score decreased to 58.04%. While precision stayed relatively decent at 70.65%, the sharp decline in recall shows that the model struggled to identify most of the severe cases in the new data. This performance drop indicates that the model did not generalize well and had overfit to patterns in the original dataset, as shown in Figure 13.

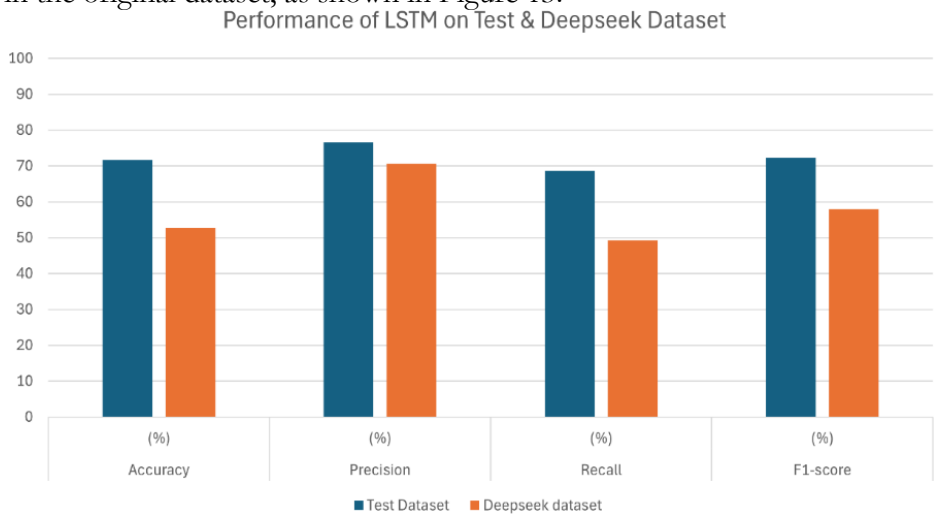

Figure 13. Performance of the LSTM model on the original test dataset versus the DeepSeek dataset

LSTM, while more advanced than traditional ML models, still exhibited overfitting. It lacked robustness when exposed to bug reports that were phrased or structured differently from those seen during training.

Comparative Perspective:

Compared to classical TF-IDF models, LSTM achieved slightly better performance on the original test set but showed a larger drop on unseen data. LLMs (LLaMA3.1 and Qwen3) not only retained performance but also improved on the DeepSeek dataset, clearly outperforming LSTM and demonstrating much stronger generalization.

Experiment-IV: Comparative Analysis of LLMs, Classical Models, and LSTM:

This experiment focuses on evaluating all previously tested models, LLMs, classical machine learning algorithms, and LSTMs, based on their performance across two datasets: the original test dataset and the DeepSeek-generated dataset. The goal is to identify the most accurate, efficient, and generalizable model for automating bug report severity classification.

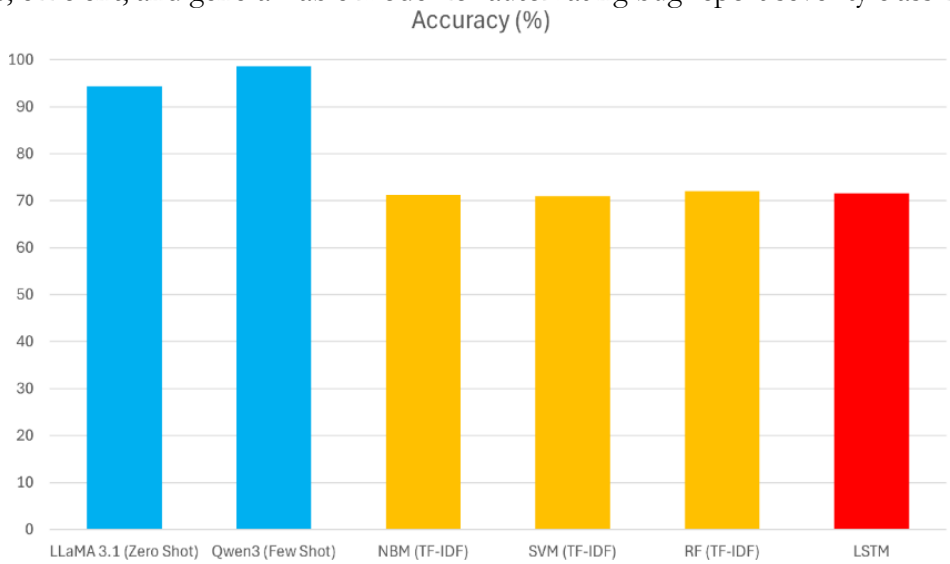

Figure 14. Accuracy of proposed approach against baseline models

LLaMA3.1 delivered consistent performance across both datasets. It performed slightly better in zero-shot than in few-shot mode on the original test set and maintained or improved its performance on DeepSeek data. It achieves Accuracy (DeepSeek, Zero-Shot): 97.99% and F1-Score (DeepSeek, Zero-Shot): 98.51%, while Qwen3 shows greater improvement with few-shot learning, especially in generalization. It gives an Accuracy (DeepSeek, Few-Shot): 97.9% and an F1-Score (DeepSeek, Few-Shot): 97.5%. LLMs outperformed all other models in both accuracy and generalization, even without training. Models showed moderate performance, as shown in Figure 14 on the original dataset and retained acceptable performance on DeepSeek data. Best F1-Score: NBM (TF-IDF): 76.33%. Models, especially SVM and RF, showed sharp performance drops on DeepSeek, suggesting overfitting. NBM achieved high recall (100%) but low precision, indicating bias toward predicting severity. SVM (Word2Vec): 55.83%. Classical models performed reasonably well with TF-IDF but struggled to generalize with Word2Vec. LSTM showed moderate results on the test dataset, with an accuracy of 71.6% and an F1-score of 72.3%, while on the DeepSeek dataset, performance declined sharply to an accuracy of 52.76% and an F1-score of 58.04%. LSTM demonstrated signs of overfitting and lacked robustness when exposed to new, unseen bug report data.

LLMs (especially LLaMA 3.1 and Qwen 3) are the most reliable and scalable choice for this task. They offer high accuracy, minimal preprocessing, and strong generalization without retraining. Classical ML models are acceptable for stable datasets but require careful tuning and choice of embeddings. LSTM, while promising, showed limited ability to adapt to diverse bug report styles and may not scale well without further optimization.

RQ9: What are the relative benefits of LLaMA3.1 and Qwen3 to classical machine learning models and LSTM in the automation of bug severity prediction of reports?

LLaMA3.1 and Qwen3 clearly stand out compared to traditional machine learning models and LSTM. They don't require training, complex preprocessing, or labeled data to perform well. Despite that, they achieved higher accuracy and showed stronger generalization on new, unseen bug reports. In contrast, classical models and LSTM struggled

to maintain performance outside the test dataset. This shows that LLMs are not only more accurate but also far more adaptable and efficient for real-world bug report severity classification tasks.

Experimental results indicated that our proposed LLM-based approach consistently outperformed classical ML models and LSTMs in classifying bug reports. In both zero-shot and few-shot learning scenarios, the approach achieved high accuracy while minimizing preprocessing requirements and eliminating the need for prior training. These performances directly support the research aim, minimizing dependence on labeled datasets and manual preprocessing. The process through which these results were achieved highlights the inherent strengths of LLMs. Unlike classical models, which rely heavily on structured features generated through techniques such as TF-IDF and Word2Vec, LLMs leverage their pre-trained knowledge base and contextual understanding of language. This allows them to interpret bug report descriptions directly in their natural language form without extensive manual feature engineering. By using prompt engineering, the models were guided to classify reports as severe or non-severe in a way that closely mirrors human reasoning. In the few-shot setting, providing a small number of labeled examples further enhanced performance by enabling the model to adapt quickly to the domain-specific context of bug reports. On the other hand, using TF-IDF word embeddings, classical models like Naïve Bayes Multinomial, SVM, and Random Forest performed reasonably well but struggled with Word2Vec. Furthermore, these models exhibited a significant drop in performance on unseen data, indicating limited generalization. This outcome is expected since TF-IDF captures the frequency-based importance of words, which performs effectively for known datasets but struggles to generalize beyond the training distributions. Word2Vec, while capturing semantic relationships, requires large training data to be effective; on smaller or domain-specific datasets, such as bug reports, its representations may fail to capture the nuances required for accurate classification. LSTM demonstrated better performance on sequential data than classical models but failed to match the robustness and consistency of LLMs, particularly when tested on new datasets. Although LSTM is designed to capture temporal and contextual dependencies in text, its training is data-intensive and prone to overfitting. Without extensive labeled data, its ability to generalize to unseen bug reports remains limited. Overall, the results confirm that LLMs not only achieve high accuracy but also offer practical advantages in scalability, adaptability, and generalization. These models reduced the need for manual pre-processing and complex feature engineering while still achieving superior results. Moreover, their resilience on unseen data demonstrates their potential for real-world deployment, where software maintenance tasks often involve unpredictable and evolving bug reports. These outcomes strongly support the proposed approach and its potential for real-world applications in software maintenance, making it a promising direction for future automation of bug report severity prediction.

Strengths of the Proposed Approach:

High Accuracy Without Training: A major strength of this approach is that models such as LLaMA 3.1 and Qwen 3 can accurately classify bug reports without prior training. Even in zero-shot settings, where the model hasn't seen any labeled examples, the results were highly accurate. This is especially useful in real-world scenarios where labeled data is limited or unavailable.

Minimal Preprocessing Requirements: Unlike classical models that require extensive data cleaning, formatting, and feature extraction, LLMs work directly with raw text. This means less manual effort is needed, making the process faster and more efficient.

Strong Generalization to Unseen Data: The LLMs didn't just perform well on the data they were evaluated on; they also handled entirely new data impressively. This shows that the

models are not overfit to a single dataset and can generalize well to different bug reports, which is important in real software projects.

Works Without Labeled Datasets: One of the biggest challenges in machine learning is finding enough labeled data. With LLaMA 3.1 and Qwen 3, even a few examples (or none at all) are enough to achieve strong performance. This makes the approach ideal for fast-paced environments where labeled bug reports may not always be available.

Limitations of the Proposed Approach:

The proposed LLM-based approach showed strong performance; it is important to recognize certain limitations that may affect its practical adoption.

High Computational Requirements: Although LLMs like LLaMA 3.1 and Qwen 3 eliminate the need for training, they still require considerable computational power to run effectively. Inference tasks demand access to a high-performance GPU and sufficient RAM, which may not be readily available in all development environments. This can limit access to these models for smaller teams or organizations without advanced hardware.

Sensitivity to Prompt Design: The performance of these models heavily depends on how prompts are designed. A carefully designed prompt can produce accurate, reliable predictions, while unclear or poorly designed prompts may yield unreliable outputs. This highlights prompt engineering as a critical skill that can pose challenges for users with little or no experience with how language models respond to various types of instructions.

Implications for Software Engineering:

This study demonstrates the potential of large language models such as LLaMA 3.1 and Qwen 3. These models can improve how we processe bug reports in software development. They can be used to facilitate severity prediction with high accuracy. Setup is straightforward deployment. This reduces manual work and boosts efficiency. These models also perform well on new and continuously changing data. That makes them useful in real-world situations. Plus, they need very little preprocessing or labeled data. So, they can easily fit into current workflows. Overall, these open-weight large language models can provide a intelligent and adaptive approach to accelerate bug triaging process and software maintenance.

Conclusion:

This research explored how Large Language Models (LLMs) such as LLaMA 3.1 and Qwen 3 can be used to automatically classify software bug reports by severity. The goal of this study was to assess how well large language models (LLMs) perform compared with traditional machine learning models such as SVM, NBM, RF, and the deep learning model LSTM. The results show that open-weight LLMs, using zero-shot and few-shot learning, achieve exceptional accuracy and other performance metrics, far surpassing those of classical ML models. One of the key findings was the potential of LLMs to not only perform well on known data but also maintain consistent accuracy on unknown and unseen data. This indicates LLM models can produce generalized results and are suggesting their potential applicability in software development projects. where the textual details of bug reports may vary widely.

Traditional ML models with TF-IDF textual embeddings showed adequate performance but did not perform well with Word2Vec. Furthermore, these models showed a significant decline in performance when evaluated on unseen data. LSTM performed well but showed signs of overfitting when tested on a new dataset, compared with the LLaMA 3.1 and Qwen 3 models. In general, LLMs provide a scalable and efficient approach to automating severity prediction in bug reports, therefore saving developers time and effort and raising software quality.

This study provides significant contributions to the field of software engineering, especially in the domain of software maintenance and bug handling:

Leveraging Open-weighted LLMs for Bug Severity Prediction: By leveraging emergent open-weighted LLaMA3.1 and Qwen3 models, this work developed an improved and optimized approach to classify bug reports into two binary classes, facilitating software developers to resolve critical issues in a timely manner, a crucial factor in ensuring software quality.

Minimizing Manual Preprocessing with Prompt Engineering: Zero-shot and few-shot learning minimize the requirement for labeled data and training, therefore saving human effort and enabling faster deployment. This helps to streamline the process for software maintenance tasks.

Comparing Large Language Models with Classical ML Models: After a comprehensive assessment, the study emphasizes how LLaMA3.1 and Qwen3 beats classical ML models and LSTM in terms of accuracy and other evaluation metrics, enabling developers and researchers to better understand which tools perform more effectively in severity classification tasks.

Better Generalization for Real-World Maintenance: By testing the models on unseen data, the study shows that LLMs adapt well to real-world bug reports, making them a more reliable option for ongoing and evolving software projects.

Future Work: While this study demonstrated the effectiveness of LLMs for binary classification of bug reports, there are still areas worth exploring:

Expanding to Multi-Class Classification: This research focused on classifying bug reports into two categories: severe and non-severe. Future work could explore multi-class classification to predict more specific severity levels (e.g., critical, major, minor), offering deeper insights and more precise prioritization for software teams.

Exploring More Lightweight or Open-Source Models: Although LLaMA3.1 and Qwen3 delivered strong results, they require substantial computational resources. Future studies may consider using smaller or more lightweight open-source models to strike a balance between performance and practicality, especially for environments with limited hardware.

References:

[1] "(PDF) Software Engineering: A Practitioner's Approach 9 th Edition." Accessed: Jul. 06, 2026. [Online]. Available: https://www.researchgate.net/publication/365946272_Software_Engineering_A_Practitioner's_Approach_9_th_Edition

[2] Muhammad Ali Arshad, Adnan Riaz, "SevPredict: Exploring the Potential of Large Language Models in Software Maintenance," *AI*, vol. 5, no. 4, pp. 2739–2760, 2024, doi: https://doi.org/10.3390/ai5040132.

[3] P. Mahajan, K. Choudhary, N. Rana, R. Kumar, and S. Deshmukh, "Software Bugs Classification Using SVM, RF, DT Algorithms," *Lect. Notes Networks Syst.*, vol. 1237, pp. 51–62, 2025, doi: 10.1007/978-981-96-1185-0_5/SAVE-RESEARCH.

[4] G. Sharma and P. Singh, "Comparative Study: Word2Vec Versus TF-IDF in Software Defect Predictions," *Lect. Notes Networks Syst.*, vol. 1165, pp. 95–107, 2025, doi: 10.1007/978-981-97-8336-6_8/SAVE-RESEARCH.

[5] Ömer Köksal, Bedir Tekinerdogan, "Automated Classification of Unstructured Bilingual Software Bug Reports: An Industrial Case Study Research," *Appl. Sci.*, vol. 12, no. 1, p. 338, 2022, doi: https://doi.org/10.3390/app12010338.

[6] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, "LLaMA: Open and Efficient Foundation Language Models," *arXiv:2302.13971*, 2023, [Online]. Available: https://arxiv.org/abs/2302.13971

[7] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, Graham Neubig, "Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing," *arXiv:2107.13586*, 2021, [Online]. Available:

https://arxiv.org/abs/2107.13586

[8] "Application of Large Language Models to Software Engineering Tasks: Opportunities, Risks, and Implications | IEEE Journals & Magazine | IEEE Xplore." Accessed: Jul. 06, 2026. [Online]. Available: https://ieeexplore.ieee.org/document/10109345

[9] Ehsan Mashhadi, Shaiful Chowdhury, "An empirical study on bug severity estimation using source code metrics and static analysis," *J. Syst. Softw.*, vol. 217, p. 112179, 2024, doi: https://doi.org/10.1016/j.jss.2024.112179.

[10] Quanjun Zhang, Chunrong Fang, Yuxiang Ma, Weisong Sun, Zhenyu Chen, "A Survey of Learning-based Automated Program Repair," *arXiv:2301.03270*, 2023, [Online]. Available: https://arxiv.org/abs/2301.03270

[11] Kevin Moran, "Enhancing Android Application Bug Reporting," *arXiv:1706.01118*, 2017, [Online]. Available: https://arxiv.org/abs/1706.01118

[12] Ashima Kukkar, Rajni Mohana, "Does bug report summarization help in enhancing the accuracy of bug severity classification?," *Procedia Comput. Sci.*, vol. 167, pp. 1345–1353, 2020, doi: https://doi.org/10.1016/j.procs.2020.03.345.

[13] "Deep Neural Network-Based Severity Prediction of Bug Reports | IEEE Journals & Magazine | IEEE Xplore." Accessed: Jul. 06, 2026. [Online]. Available: https://ieeexplore.ieee.org/document/8685106

[14] J. N. Singh, K. V. Gupta, S. Kumar, R. Priya, and K. Saini, "Bugzilla Bug Tracking System," *Proc. - IEEE 2023 5th Int. Conf. Adv. Comput. Commun. Control Networking, ICAC3N 2023*, pp. 1522–1526, 2023, doi: 10.1109/ICAC3N60023.2023.10541310.

[15] "Revolutionize IT Support with Jira Service Management | Atlassian." Accessed: Jul. 06, 2026. [Online]. Available: https://www.atlassian.com/software/jira/service-management?campaign=20339374771&adgroup=152515828964&targetid=kwd-1683663083742&matchtype=p&network=g&device=c&device_model=&creative=672220291174&keyword=jira service management ticketing tool&placement=&target=&ds_eid=700000001721198&ds_e1=GOOGLE&gad_source=1&gad_campaignid=20339374771&gbraid=0AAAADAVAKcuW8U5-PmnTAmLnQcfu_ALU&gclid=CjwKCAjwpK3SBhASEiwAtV1SPFyF4746okaEe5U82fQqux4wYXLEjq3UmoO_e5zJPiaSBOmN0QB-5hoCGYsQAvD_BwE

[16] M. A. Arshad and H. Zhiqiu, "Using CNN to Predict the Resolution Status of Bug Reports," *J. Phys. Conf. Ser.*, vol. 1828, no. 1, p. 012106, Feb. 2021, doi: 10.1088/1742-6596/1828/1/012106.

[17] Antonios Saravanos, Matthew X. Curinga, "Simulating the Software Development Lifecycle: The Waterfall Model," *Appl. Syst. Innov.*, vol. 6, no. 6, p. 108, 2023, doi: https://doi.org/10.3390/asi6060108.

[18] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, Jeffrey Dean, "Distributed Representations of Words and Phrases and their Compositionality," *arXiv:1310.4546*, 2013, [Online]. Available: https://arxiv.org/abs/1310.4546

[19] Yang, J. Kim and G., "Bug Severity Prediction Algorithm Using Topic-Based Feature Selection and CNN-LSTM Algorithm," *IEEE Access*, vol. 10, pp. 94643–94651, 2022, doi: 10.1109/ACCESS.2022.3204689.

[20] "A Light Bug Triage Framework for Applying Large Pre-trained Language Model | Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering." Accessed: Jul. 11, 2026. [Online]. Available: https://dl.acm.org/doi/10.1145/3551349.3556898

[21] A. K. Dipongkor and K. Moran, "A Comparative Study of Transformer-Based Neural Text Representation Techniques on Bug Triaging," *Proc. - 2023 38th IEEE/ACM Int. Conf. Autom. Softw. Eng. ASE 2023*, pp. 1012–1023, 2023, doi:

10.1109/ASE56229.2023.00217.

[22] Asif Ali, Yuanqing Xia, "BERT based severity prediction of bug reports for the maintenance of mobile applications," *J. Syst. Softw.*, vol. 208, p. 111898, 2024, doi: https://doi.org/10.1016/j.jss.2023.111898.

[23] "Graph Neural Network vs. Large Language Model: A Comparative Analysis for Bug Report Priority and Severity Prediction | Proceedings of the 20th International Conference on Predictive Models and Data Analytics in Software Engineering." Accessed: Jul. 11, 2026. [Online]. Available: https://dl.acm.org/doi/10.1145/3663533.3664042

[24] M. Theingi and N. L. Wah, "Automated Prediction of Bug Reports' Severity using Attention-based Convolutional Neural Network," *Proc. 22nd IEEE Int. Conf. Comput. Appl. ICCA 2025*, 2025, doi: 10.1109/ICCA65395.2025.11011201.

[25] F. Thung, D. Lo, and L. Jiang, "Automatic defect categorization," *Proc. - Work. Conf. Reverse Eng. WCRE*, pp. 205–214, 2012, doi: 10.1109/WCRE.2012.30.

[26] Piotr Bojanowski, Edouard Grave, Armand Joulin, Tomas Mikolov, "Enriching Word Vectors with Subword Information," *arXiv:1607.04606*, 2017, [Online]. Available: https://arxiv.org/abs/1607.04606

[27] S. Kumar, S. Muttoo, and V. B. Singh, "Classification of Software Defects Using Orthogonal Defect Classification," *Int. J. Open Source Softw. Process.*, vol. 13, no. 1, pp. 1–16, Jan. 2022, doi: 10.4018/IJOSSP.300749.

[28] Y. Zhou, Y. Tong, R. Gu, and H. Gall, "Combining text mining and data mining for bug report classification," *Proc. - 30th Int. Conf. Softw. Maint. Evol. ICSME 2014*, pp. 311–320, Dec. 2014, doi: 10.1109/ICSME.2014.53.

[29] S. Kumar, M. Sharma, V. B. Singh, and S. K. Muttoo, "Bug Report Classification into Orthogonal Defect Classification Defect Type using Long Short Term Memory," *Proc. - 2021 3rd Int. Conf. Adv. Comput. Commun. Control Networking, ICAC3N 2021*, pp. 285–287, 2021, doi: 10.1109/ICAC3N53548.2021.9725398.

[30] "CaPBug-A Framework for Automatic Bug Categorization and Prioritization Using NLP and Machine Learning Algorithms | IEEE Journals & Magazine | IEEE Xplore." Accessed: Jul. 06, 2026. [Online]. Available: https://ieeexplore.ieee.org/document/9388660

[31] Ahmed Fawzi Otoom, Sara Al-Jdaeh, "Automated Classification of Software Bug Reports," *ACM Int. Conf. Proceeding Ser.*, 2019, [Online]. Available: https://dl.acm.org/doi/10.1145/3357419.3357424

[32] Ashima Kukkar, Rajni Mohana, "A Supervised Bug Report Classification with Incorporate and Textual field Knowledge," *Procedia Comput. Sci.*, vol. 132, pp. 352–361, 2018, doi: https://doi.org/10.1016/j.procs.2018.05.194.

[33] A. Lamkanfi, J. Pérez, and S. Demeyer, "The eclipse and mozilla defect tracking dataset: A genuine dataset for mining bug information," *IEEE Int. Work. Conf. Min. Softw. Repos.*, pp. 203–206, 2013, doi: 10.1109/MSR.2013.6624028.

[34] S. Suthaharan, "Machine Learning Models and Algorithms for Big Data Classification," vol. 36, 2016, doi: 10.1007/978-1-4899-7641-3.

[35] H. M. Tran, S. Van Nguyen, S. V. U. Ha, and T. Q. Le, "An analysis of software bug reports using random forest," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 11251 LNCS, pp. 273–285, 2018, doi: 10.1007/978-3-030-03192-3_21/SAVE-RESEARCH.

[36] Ahmed Fawzi Otoom, Sara Al-Jdaeh, "An implementation of naive bayes classifier," *ACM Int. Conf. Proceeding Ser.*, 2019, [Online]. Available: https://dl.acm.org/doi/10.1145/3357419.3357424

[37] X. Ye, F. Fang, J. Wu, R. Bunescu, and C. Liu, "Bug Report Classification Using

LSTM Architecture for More Accurate Software Defect Locating," *Proc. - 17th IEEE Int. Conf. Mach. Learn. Appl. ICMLA 2018*, pp. 1438–1445, Jul. 2018, doi: 10.1109/ICMLA.2018.00234.

[38] "Classifying Bug Reports into Bugs and Non-bugs Using LSTM | Proceedings of the 10th Asia-Pacific Symposium on Internetware." Accessed: Jul. 06, 2026. [Online]. Available: https://dl.acm.org/doi/10.1145/3275219.3275239

[39] Edward Loper, Steven Bird, "NLTK: The Natural Language Toolkit," *arXiv:cs/0205028*, 2002, [Online]. Available: https://arxiv.org/abs/cs/0205028

[40] "TextBlob: Simplified Text Processing — TextBlob 0.19.0 documentation." Accessed: Jul. 06, 2026. [Online]. Available: https://textblob.readthedocs.io/en/dev/