BACHELOR OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING

LLM-Based Auto-Labeling of Developer Discussions: A Comparative Study of Zero-Shot, Sampling Methods, Ensembles and Judge-Guided Strategies

Chowdhury Ashfaq Shakhawat
200042123
Md Soyeb
200042157
Iftekharul Haque
200042159

Department of Computer Science and Engineering
Islamic University of Technology
September, 2025

LLM-Based Auto-Labeling of Developer Discussions:
A Comparative Study of Zero-Shot, Sampling Methods, Ensembles and Judge-Guided Strategies

Chowdhury Ashfaq Shakhawat
200042123
Md Soyeb
200042157
Iftekharul Haque
200042159

Department of Computer Science and Engineering
Islamic University of Technology
September, 2025

Declaration of Candidate

This is to certify that the work presented in this thesis is the outcome of the analysis and experiments carried out by **Chowdhury Ashfaq Shakhawat**, **Md Soyeb**, and **Iftekharul Haque** under the supervision of **Md. Tariquzzaman**, Junior Lecturer, Department of Computer Science and Engineering, IUT and co-supervision of **Dr. Kamrul Hasan**, Professor, Department of Computer Science and Engineering, IUT and **Dr. Hasan Mahmud**, Professor, Department of Computer Science and Engineering, IUT, Islamic University of Technology, Dhaka, Bangladesh. It is also declared that neither this thesis nor any part of it has been submitted anywhere else for any degree or diploma. Information derived from the published and unpublished work of others have been acknowledged in the text and a list of references is given.

Md. Tariquzzaman
Junior Lecturer
Department of Computer Science and Engineering, IUT
Islamic University of Technology (IUT)
Date: September 30, 2025

Dr. Kamrul Hasan
Professor
Department of Computer Science and Engineering, IUT
Islamic University of Technology (IUT)
Date: September 30, 2025

Dr. Hasan Mahmud
Professor
Department of Computer Science and Engineering, IUT
Islamic University of Technology (IUT)
Date: September 30, 2025

Chowdhury Ashfaq Shakhawat
Student ID: 200042123
Date: September 30, 2025

Md Soyeb
Student ID: 200042157
Date: September 30, 2025

Iftekharul Haque
Student ID: 200042159
Date: September 30, 2025

Dedicated to Palestinian people, whose daily struggles remind us to hope.

Contents

1 Introduction **1**
1.1 Research Questions 3
1.2 Contributions 4

2 Literature Review **5**
2.1 Large Language Models in Bug Classification 5
2.2 Automatic Bug Classification Techniques 14
2.3 Bug Taxonomy in Deep Learning Software 18
2.4 Comparative Analysis of Bug Classification Approaches 20
2.4.1 Unique Features of Author's Approach 22

3 Methodology **25**
3.1 Github Issues Dataset 25
3.2 Pipeline 25
3.3 GitHub Issue Labeling with Multiple LLMs 31
3.4 Results & Observations 33
3.5 Prepared Dataset 34
3.6 Workflow Summary 37
3.7 Mathematical Formula and Theories 38
3.7.1 Cohen's Kappa Coefficient 38
3.7.2 Fleiss' Kappa Coefficient 40
3.8 Rationals behind the Coefficients 42
3.9 Evaluation Metrics 43
3.9.1 Precision 43
3.9.2 Recall 43
3.9.3 F1 44
3.9.4 Confusion Matrix 44
3.9.5 Relevance to This Research 44

4 Results and Discussion **45**
4.1 GitHub Issue labeling Using individual LLM 45
4.2 GitHub Issue labeling Using 2 LLMs and 1 Judge 45
4.3 GitHub Issue labeling Using 2 LLMs and 1 Judge (tuning K-sampling) 46
4.4 GitHub Issue labeling Using 3 LLMs and 1 Judge 47
4.5 GitHub Issue labeling Using 3 LLMs and 1 Judge (tuning K-sampling) 48
4.6 LLM Issues labeling using 2 peers 1 judge with K-sampling on Prepared Dataset 49
4.7 Comparative Discussion 50
4.7.1 Overall Trends: Single Models vs. Ensembles 50
4.7.2 Effect of the Judge: Convergence and Stability 51
4.7.3 Impact of k-Sampling: Finding the Sweet Spot 51
4.7.4 Two vs. Three Peers: Marginal Accuracy, Higher Reliability . . 51
4.7.5 Reliability Trajectories with Temperature Tuning 52
4.7.6 Error Analysis and Class Imbalance 52
4.7.7 Prepared Dataset vs. GitHub Issues: Why Accuracy Differs . . 52
4.7.8 On RAG and Few-Shot: Limited Gains in Heterogeneous Issues 53
4.7.9 Practical Guidance 53
4.7.10 Threats to Validity and Limitations 53

5 Dataset Construction **55**
5.1 Sources and Raw Ingest 55
5.2 Cleaning and Filtering 55
5.3 Final Schema 56
5.4 Data Distributions 56
5.5 Discussion 56

6 Conclusion **57**

Appendices **62**

A Prompts used for Evaluations **63**
A.1 Stackoverflow prompt for each iteration: 63
A.2 Stackoverflow initial prompt for each peer: 64
A.3 Stackoverflow Judge prompt: 65
A.4 Github Initial prompt for each peer: 65
A.5 Github Iteration prompt: 66
A.6 Github Judge Prompt: 67

List of Figures

2.1 Overview of the LLPut methodology illustrating the pipeline from data acquisition and prompt design to model evaluation and error analysis. 6
2.2 Condensed LLPut Framework Flow: From Bug Report Collection to Input Extraction and Thematic Analysis 7
2.3 Overview of the GPT-3.5-turbo based methodology for issue classification, demonstrating preprocessing, fine-tuning via OpenAI API, and evaluation across multiple repositories. 10
2.4 Workflow for GPT-based Issue Classification using OpenAI API . . . 12
2.5 Overview of the LLM-BRC framework: bug report preparation, embedding using text-embedding-ada-002, and classification via feed-forward neural network. 12
2.6 Three-level hierarchical classification of bug reports in LLM-BRC. . . 13
2.7 Ablation study comparing bug report components on different classification tasks (LLM-BRC) 14
2.8 Architecture of the ABTS system: from bug report preprocessing to classification into logical, GUI, enhancement, and analysis bugs. . . . 15
2.9 Workflow of ensemble-based classification approach using ADE for TSBR datasets (module, developer, and issue/feature classification). . 17
2.10 Workflow for analyzing deep learning bug characteristics using Stack Overflow and GitHub data. 19
2.11 Workflow of bug labeling and Cohen's Kappa agreement resolution used in [11] [16]. 19
2.12 Timeline of prominent bug classification research. 21

3.1 Distribution of Issue Labels 25
3.2 Baseline Zero-Shot Prompting 26
3.3 Zero-Shot Two Peer One Judge Ensemble 27
3.4 Two Peer One Judge with K-sampling 27
3.5 Three Peer One Judge with Temperature Tuning 29

3.6 Three Peer One Judge with K-sampling 29
3.7 Zero-shot GitHub issue labeling using 2 LLMs and arbitration. 37
3.8 Zero-shot GitHub issue labeling using 2 LLMs and arbitration. 37
3.9 Zero-shot GitHub issue labeling using 3 LLMs and arbitration. 38

4.1 Cohen's Kappa across four labeling iterations for 2 LLMs 46
4.2 Cohen's Kappa across Iterations 47
4.3 Fleiss' Kappa across three labeling iterations for 3 LLMs 48
4.4 Fleiss' Kappa across three labeling iterations for 3 LLMs 48
4.5 Confusion Matrix of 2 LLM and 1 Judge with K-sampling 50

5.1 Post-cleaning Dataset 55

List of Tables

2.1 Detailed Issue Report Classification Results by Repository and Label . 11
2.2 LLM-BRC Classification Performance across Frameworks and Tasks . 13
2.3 Performance evaluation using textual, categorical, and combined attributes (before feature selection) 15
2.4 Performance with selected features using wrapper and filtration technique 16
2.5 Comparison with state-of-the-art bug classification techniques 16
2.6 Comparative Summary of Bug Classification Methods 21
2.7 Comparative Summary of Bug Classification Methods 22

3.1 Label Taxonomy Standardization 32
3.2 Zero Shot: 2 Peers, 1 Judge (GitHub Issues) 33
3.3 Zero Shot: 2 Peers, 1 Judge with K-sampling (GitHub Issues) 33
3.4 Zero Shot: 3 Peers, 1 Judge (GitHub Issues) 33
3.5 Zero Shot: 3 Peers, 1 Judge with K-sampling (GitHub Issues) 33
3.6 Zero Shot: 2 Peers, 1 Judge with K-sampling (StackOverflow) 34
3.7 Stack Overflow Dataset Fields 35

4.1 Individual LLM accuracies and agreement for GitHub issue labeling . 45
4.2 Individual LLM accuracies for GitHub issue labeling (2-model pipeline) 45
4.3 Zero Shot 2 peers, 1 judge with k-sampling (top-3) 46
4.4 Zero Shot 2 peers, 1 judge with k-sampling (top-7) 46
4.5 Individual LLM accuracies for GitHub issue labeling (3-model pipeline) 47
4.6 Individual LLM accuracies for GitHub issue labeling (3-model pipeline) 48
4.7 Classification Report on Cleaned Dataset (1547 samples) 49

List of Abbreviations

LLM	Large Language Model
BLEU	Bilingual Evaluation Understudy
API	Application Programming Interface
NLP	Natural Language Processing
FFN	Feed-Forward Neural Network

Acknowledgement

We express heartfelt gratitude to Systems and Software Lab (SSL) for technical guidance and moral support. Special thanks to our mentors and AI models that made annotation-less labeling a reality.

// Abstract

Software bugs have long posed challenges to the delivery of reliable digital services, prompting extensive research into automated bug labeling. While significant advancements have been made, existing approaches often struggle with high false positive rates and face difficulties in practical deployment due to reliance on structured bug reports. Most contemporary studies utilize structured datasets containing developer-generated bug reports, typically written in natural language. These reports require manual or semi-automated extraction of relevant inputs, a process that is both time-consuming and error-prone.

With the emergence of Large Language Models (LLMs), a new research opportunity arises: can LLMs effectively extract failure-inducing inputs from unstructured, community-driven sources such as GitHub, Stack Overflow, and other developer forums? In this study, we propose a novel end-to-end pipeline that leverages LLMs for bug labeling directly from raw, unstructured text. Our methodology focuses on automated labeling, utilizing prompt-based approaches to optimize the performance of generative models.

We curated and annotated a dataset comprising 1885 Stack Overflow questions posted between 2023 and 2025, and further validated our approach using a dataset of GitHub issue reports. Through extensive experimentation, we assess the accuracy and robustness of our pipeline across diverse input formats. Unlike existing solutions, our proposed framework emphasizes simplicity, scalability, and cost-effectiveness, making it well-suited for integration into real-world software development workflows.

1 Introduction

Recent years have seen enterprise language models embedded in a wide range of domains from customer-support, chatbots and healthcare question-answering to legal assistants and enterprise search [6]. These applications rely on a lot of unstructured user requests and issue reports, and any system that supports them must understand problems expressed in natural language which is mostly unstructured. Companies spends a huge amount of money and effort to clean and process this unstructured data before feeding them into AI systems. Retrieval-augmented generation (RAG) pipelines have become common because they pair a retriever that fetches relevant documents with a generator that synthesizes answers; this modular architecture allows enterprises to surface current or proprietary knowledge without retraining models.

Beyond retrieval, the way language models decode text strongly influences their output. Top-k sampling restricts the model to the k most probable tokens at each step, increasing diversity but risking incoherence when k is large. Nucleus sampling (top-p) dynamically selects tokens whose cumulative probability reaches a threshold, balancing diversity and coherence. Temperature sampling scales the probability distribution: lower temperatures produce deterministic, repetitive text while higher temperatures increase randomness and creativity; extremely high temperatures can lead to hallucinations.

Another emerging technique in enterprise AI is LLM-as-a-Judge. Instead of manually grading model outputs, organizations increasingly use one language model to evaluate the responses of others.

Against this backdrop, we investigate whether large language models can serve as universal zero-shot labelers for unstructured issues across domains. Manual issue classification is notoriously labor-intensive and often requires domain experts [14]. While recent research shows that models like GPT-4o outperform older models for software issue classification [1], these efforts still treat bugs as simple categories and depend on curated datasets. There is no publicly available dataset of LLM-specific issues, and existing studies rarely explore how retrieval augmentation, ensemble strategies and sampling hyperparameters affect labeling quality.

To fill this gap, we build a 1500-entry dataset of GitHub issues and Stack Overflow posts describing problems encountered when using large language models. Each record is annotated with a fine-grained taxonomy of failure modes—such as Model

Loading Error, Token Limit Exceeded, and API Misuse. Our study treats this corpus as a general testbed for zero-shot labeling rather than a software-specific benchmark. We evaluate individual generative models (GPT-4.1, Claude-3.5-Sonnet, Gemini-2.5 and Deepseek-v3.1) and combine them into ensembles. Independent models label each issue; a separate LLM-judge resolves disagreements. Through this multi-model setup, we further explored how temperature tuning, top-k/top-p sampling and RAG-based prompting influence label agreement and quality.

Reliability is central to our framework. Borrowing from psychometrics, we use Cohen's kappa to measure agreement between two models [16] and Fleiss' kappa to assess consensus across three or more raters [17]. These metrics account for chance agreement and allow us to determine when the ensemble has converged on trustworthy labels. We monitor kappa scores while iteratively adjusting sampling parameters and prompts, stopping when additional adjustments no longer improve agreement. To contextualize our results, we also train traditional machine-learning classifiers (e.g., XGBoost, Random Forest) on the same dataset and compare their performance to our generative approach. This comprehensive evaluation illuminates the strengths and limitations of LLM-based auto-labeling relative to conventional methods.

1.1 Research Questions

To achieve our objectives, the following research questions are addressed in this study:

1. **How good can large language models (LLMs) serve as reliable zero-shot labelers for unstructured issue reports without domain-specific fine-tuning?**
   Addressed through experiments with individual LLMs (GPT-4.1, Claude-3.5, Gemini, Deepseek) on GitHub and Stack Overflow issues.

2. **How does prompt design impact the performance of generative models in bug labeling tasks?**
   The study explores how variations in prompt engineering influence the effectiveness and consistency of LLM outputs when classifying unstructured bug reports.

3. **How effective are ensemble frameworks (2-peer-1-judge and 3-peer-1-judge) in increasing inter-model agreement and reducing bias compared to single-model predictions?**
   Explored through experiments with adjudication pipelines using LLM-as-Judge.

4. **What is the impact of decoding strategies—temperature tuning, top-k, and top-p sampling—on the stability and reliability of labels produced by LLM ensembles?**
   Tested by systematically varying hyperparameters and measuring changes in accuracy, F1, and kappa statistics.

5. **How does the inclusion of multiple annotators (LLMs) affect the consistency and accuracy of bug labeling?**
   This question examines whether ensemble strategies with multiple LLMs improve the robustness and reliability of automated bug labeling.

1.2 Contributions

1. **A study that shows zero-shot auto-labeling capability of LLM and its ensembles.**
   A study of auto-labeling capability of enterprise LLMs, its ensembles and settings. The authors weave together retrieval augmentation, decoding strategies, ensemble labeling, and LLM-judging to understand the performance difference of each.

2. **A labeled dataset with recent LLM issues dataset**
   The authors found a gap of well-labeled LLM Issues dataset in recent times. So as a contribution we have prepared a labeled dataset with 1547 LLM issues from 2023 till 2025. The dataset has been prepared with stackoverflow developer discussions and under domain expert's supervision.

2 Literature Review

The application of Large Language Models (LLMs) in software engineering has gained momentum in recent years, particularly for bug classification tasks [1], [7]. This chapter reviews the most relevant studies in this domain, presenting thematic insights and comparative performance across approaches.

2.1 Large Language Models in Bug Classification

LLPut: Large Language Models for Bug Report Input Generation

Conducted an empirical investigation into the capabilities of generative Large Language Models (LLMs), particularly LLaMA, Qwen, and Qwen-Coder, for automating the extraction of failure-inducing inputs from software bug reports [9] [2] [10]. Their work introduced LLPut, a technique that focuses on parsing natural language descriptions in bug reports to identify structured, executable commands necessary for bug reproduction [9].

The motivation behind LLPut stems from the inherent ambiguity in bug reports, which are often written in unstructured and informal language. This makes manual extraction of reproduction steps time-consuming and error-prone. Proposed using zero-shot and one-shot prompting techniques to evaluate the efficiency and adaptability of LLMs in minimal-context scenarios [9]. Their results demonstrated that these models—especially Qwen—could reliably convert ambiguous natural language into structured commands [2]. Notably, Qwen achieved a BLEU-2 score [15] of 0.5 or higher in 62.62% of cases, indicating a strong similarity between its generated outputs and expert-labeled references [2] [15].

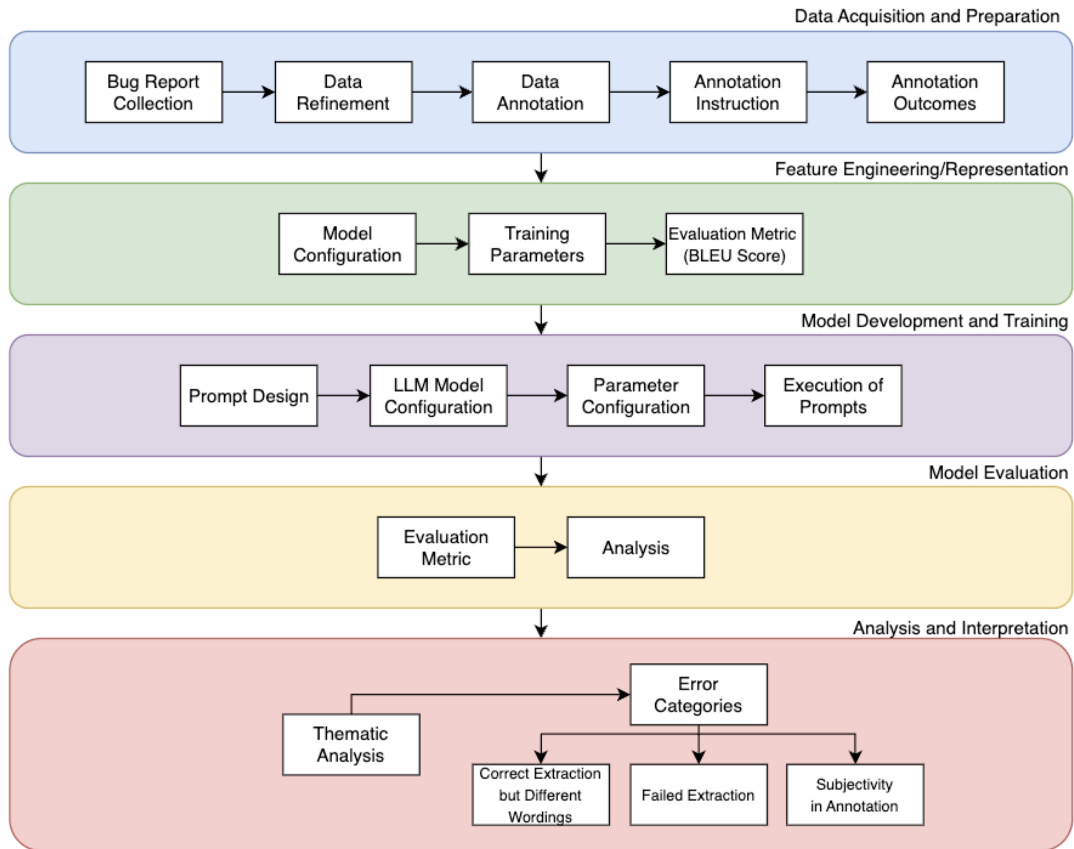

Figure 2.1: Overview of the LLPut methodology illustrating the pipeline from data acquisition and prompt design to model evaluation and error analysis.

Despite the promising results, the study identified key challenges. One major issue was the sensitivity of LLMs to slight changes in prompt wording and structure. Error analysis revealed three primary failure modes: technically correct but semantically altered outputs, outright extraction failures, and inconsistencies caused by annotation ambiguity. These challenges highlight the limitations of relying solely on prompting without domain-specific adaptation.

To overcome these issues, the authors proposed several future directions, including dynamic and iterative prompt construction strategies, fine-tuning on specialized bug report datasets, and leveraging expert input during both model training and evaluation. These strategies aim to make the extraction process more robust, scalable, and applicable to real-world debugging workflows.

LLPut: Breakdown of Methodology

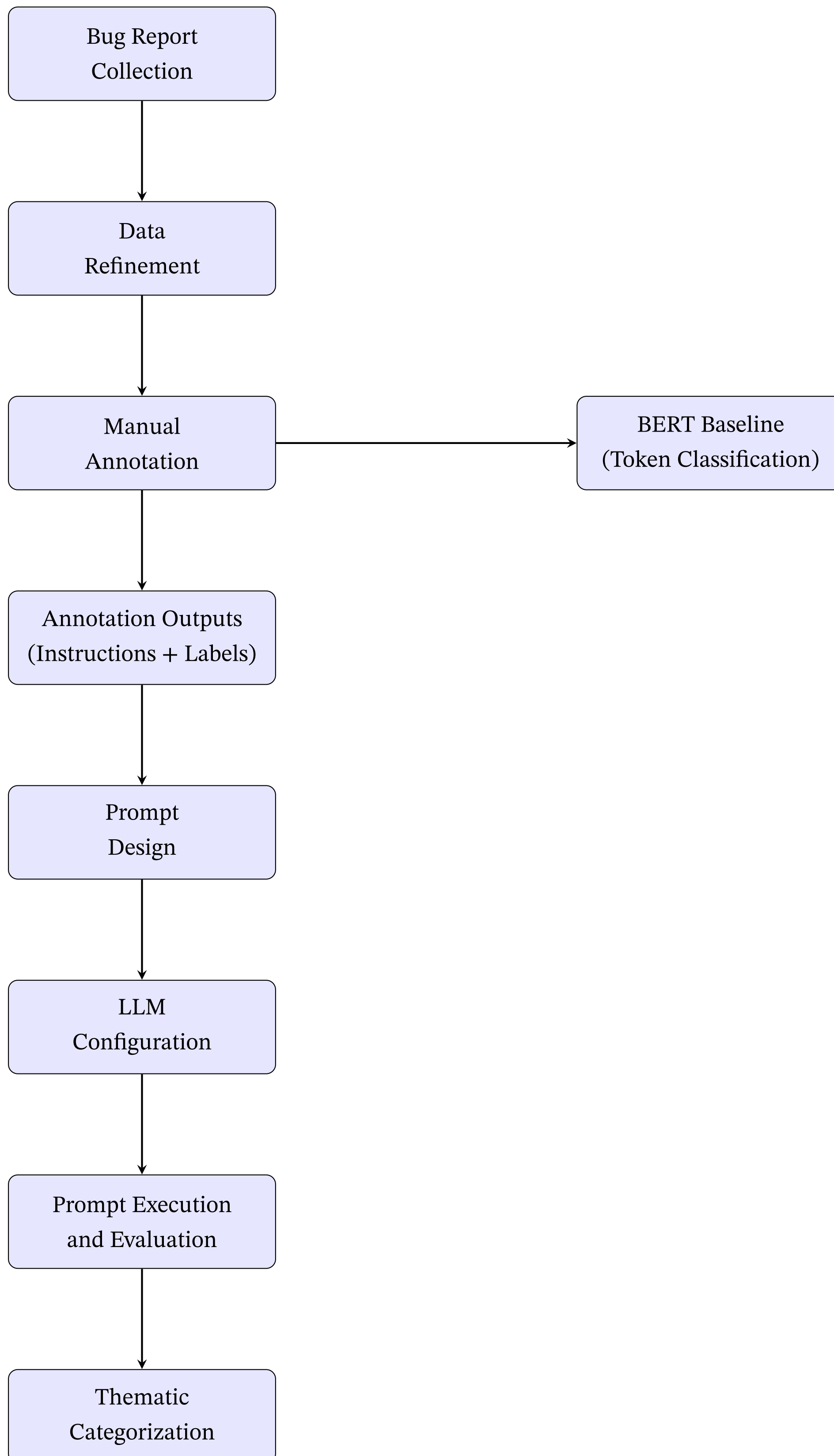

Figure 2.2: Condensed LLPut Framework Flow: From Bug Report Collection to Input Extraction and Thematic Analysis

Dataset Preparation

- *Collection*: Bug reports were collected from the Linux coreutils project via the Red Hat Bugzilla – Bug List. Filtering for Fedora and Red Hat Linux products and coreutils as the component yielded 779 bug reports [9] [3].
- *Refinement*: Reports announcing new coreutils releases were removed, leaving 753 valid reports [9].
- *Annotation*: Two authors manually annotated a random sample of 250 reports, discussing disagreements to reach consensus. Reports with vague or missing descriptions were excluded, resulting in 206 usable bug reports [9].
- *Annotation Instructions*: Annotators were instructed to extract explicit and implicit cues to reconstruct reproduction steps, including expected and observed outcomes. Even if the report was unstructured, they attempted to extract the command(s) [9].
- *Annotation Outcomes*: Among the 206 annotated reports, 149 contained identifiable input commands. Another 54 were marked as "None" (no reproducible command), and 3 ambiguous cases were also marked as "None" [9].

Application of LLMs for Input Extraction

- *Baseline*: A BERT-based NLP model was fine-tuned for token classification [6]. It used the 149 command-containing reports (80/20 train-test split) to evaluate baseline performance in identifying command tokens [9].
- *Generative LLMs*: Three open-source LLMs—LLaMA, Qwen, and Qwen-Coder—were employed to compare generative capability [9] [2] [10].
- *Prompting Strategies*: Zero-shot and one-shot prompting approaches were tested. One-shot prompts with a single example yielded the most structured results [4].
- *Prompt Design*: Each model was instructed to extract exact command(s) or output "None" if none existed. The prompt included a sample bug description and expected command.

Model Configuration

- **LLaMA**: Model used — llama-2.5-70B
- **Qwen**: Version — qwen2.5:32b-instruct
- **Qwen-Coder**: Version — qwen2.5-coder:32b

- **Parameters**: Temperature was fixed at 0 to reduce randomness. The Ollama platform was used to host and execute the models.

Evaluation Metric

The BLEU (Bilingual Evaluation Understudy) score was used to assess how closely the extracted commands matched human-annotated references [15]. BLEU-2 was chosen due to the brevity of command phrases. A BLEU-2 score above 0.30 indicated intelligible output, while scores above 0.50 represented highly fluent and accurate extraction.

Error Analysis

Thematic error analysis was conducted on 38 samples where all models scored below 0.30 [9]. Errors were categorized as:

- Correct extractions with different phrasing,
- Total extraction failures,
- Cases of subjective or ambiguous annotations.

Summary

LLPut presents a robust and multi-phase methodology:

- A carefully annotated dataset was created using Linux coreutils bug reports.
- BERT was used as a baseline extraction model [6].
- Three generative LLMs (LLaMA, Qwen, Qwen-Coder) were tested using structured prompts.
- Performance was benchmarked using BLEU-2 [15].
- Detailed error analysis identified bottlenecks and challenges for prompt-based LLMs in real-world debugging scenarios.

An overview of the LLPut pipeline can be seen in Figure 2.1.

Overall, LLPut demonstrates the feasibility of using generative LLMs for structured input extraction in software engineering contexts and lays the groundwork for more advanced, domain-aware automated debugging systems.

Applying Large Language Models API to Issue Classification

Aracena et al. introduced a robust method for classifying software issue reports by applying LLM, particularly using OpenAI's GPT-3.5-turbo model through the OpenAI

fine-tuning API [1]. This innovative approach addresses the challenge of effectively classifying issue reports using relatively small datasets, thus significantly reducing the need for extensive manual data annotation, a common requirement in traditional training methods [1].

The study involved comprehensive data preprocessing, emphasizing rigorous noise removal, text standardization, and meticulous format conversion into structured JSON data. Two specific methods of noise removal and data cleaning were implemented: the first focused on eliminating special characters, punctuation, emojis, and URLs, while the second enhanced this process by incorporating standardized placeholders for URLs, HTML tags, and usernames. Such rigorous preprocessing steps optimized the dataset for effective fine-tuning, enabling the GPT model to leverage its inherent capability to generalize effectively from limited training examples.

Performance evaluations were meticulously conducted across five distinct open-source repositories, with each repository fine-tuned individually to create specialized models that better capture repository-specific nuances. This repository-specific fine-tuning led to impressive results, with a macro-average precision of 83.24%, recall of 82.87%, and an F1-score of 82.8% [1]. Notably, precision reached as high as 90.72% in specific scenarios, indicating strong classification accuracy.

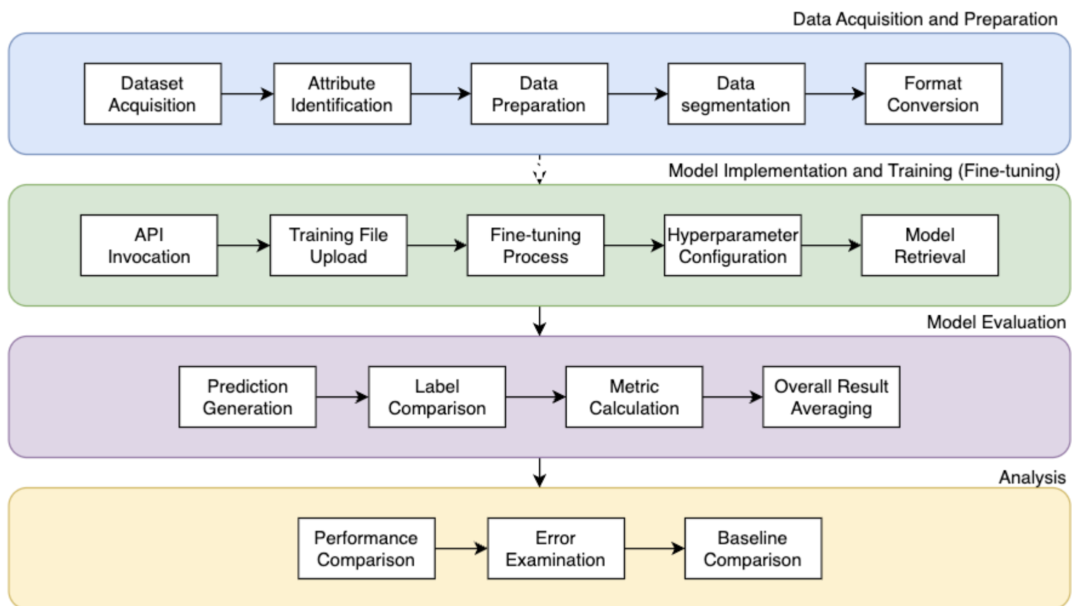

Figure 2.3: Overview of the GPT-3.5-turbo based methodology for issue classification, demonstrating preprocessing, fine-tuning via OpenAI API, and evaluation across multiple repositories.

Detailed analysis revealed variations in performance across different repositories and

labels, particularly highlighting challenges associated with correctly classifying issues labeled as "questions." The analysis pointed out that the ambiguity inherent in the "question" label significantly impacted the model's performance, suggesting potential improvements by clarifying labeling standards within software repositories.

Table 2.1: Detailed Issue Report Classification Results by Repository and Label

Repo	Label	Precision	Recall	F1-score	Epochs
facebook/react	bug	0.8333	0.9500	0.8878	3
facebook/react	feature	0.8557	0.8900	0.8725	3
facebook/react	question	0.9024	0.7400	0.8132	3
tensorflow	bug	0.9072	0.8800	0.8934	10
tensorflow	feature	0.9318	0.8200	0.8723	10
tensorflow	question	0.7913	0.9100	0.8465	10
bitcoin	bug	0.7339	0.8000	0.7656	3
bitcoin	feature	0.8318	0.8900	0.8599	3
bitcoin	question	0.7381	0.6200	0.6739	3
opencv	bug	0.7288	0.8600	0.7890	6
opencv	feature	0.9091	0.8000	0.8511	6
opencv	question	0.8617	0.8100	0.8351	6
Overall Avg.		**0.8324**	**0.8287**	**0.8280**	–

Furthermore, a comparison with baseline models employing sentence transformers such as `all-mpnet-base-v2` demonstrated that [1]'s GPT-based approach slightly outperformed these conventional models, underscoring the efficiency and suitability of GPT fine-tuning, particularly in scenarios with constrained data resources.

Overall, this study contributes significantly to software engineering practices by demonstrating the viability and effectiveness of LLMs like GPT-3.5-turbo for issue classification tasks [1]. It underscores the importance of tailored fine-tuning and preprocessing techniques to maximize performance, particularly within the resource-constrained and highly variable contexts characteristic of open-source software development.

The methodology described in this paper focuses on using the OpenAI fine-tuning API with GPT-3.5-turbo to automate issue classification. The goal is to achieve reliable issue prioritization even with a reduced training dataset.

Applying LLM API to solve the classification problem of the methodology

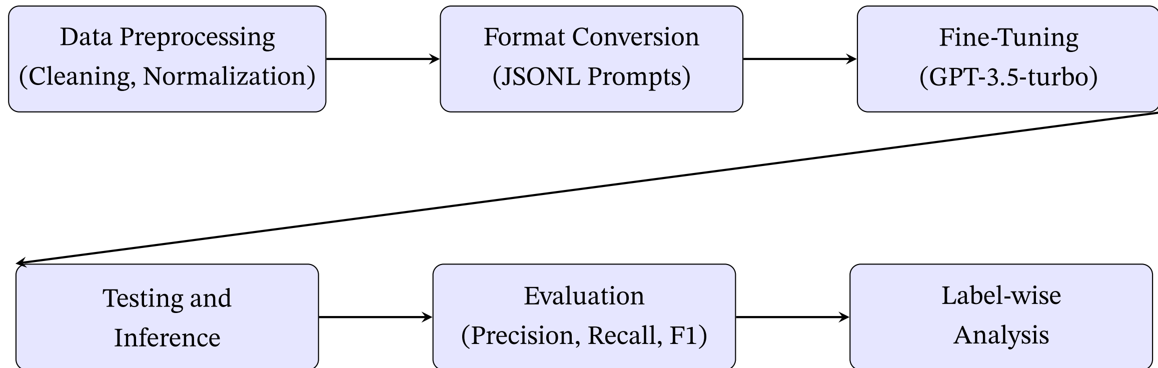

Figure 2.4: Workflow for GPT-based Issue Classification using OpenAI API

LLM-BRC: Bug Report Classification for Deep Learning Frameworks

Introduced LLM-BRC, a sophisticated framework developed to classify bug reports in deep learning environments such as TensorFlow, MXNET, and PaddlePaddle [7]. Unlike traditional approaches, LLM-BRC leverages OpenAI's advanced embedding model, `text-embedding-ada-002`, to convert bug report components into dense, high-dimensional vectors (1,536 dimensions). These vectors are then used to train a Feed-Forward Neural Network (FFN) for classification.

The authors collected 3,110 labeled bug reports across the three frameworks, extracting titles, descriptions, and developer comments to form a comprehensive dataset. The embedding model captures semantic and contextual information significantly better than previous methods like word2vec or BERT. Through extensive experimentation, the framework achieved impressive classification accuracies ranging from 92% to 98.75% across multiple classification tasks—namely bug/non-bug, BOH/MAN (Bohrbug vs. Mandelbug), and ARB/NAM (Aging-related vs. Non-aging Mandelbugs).

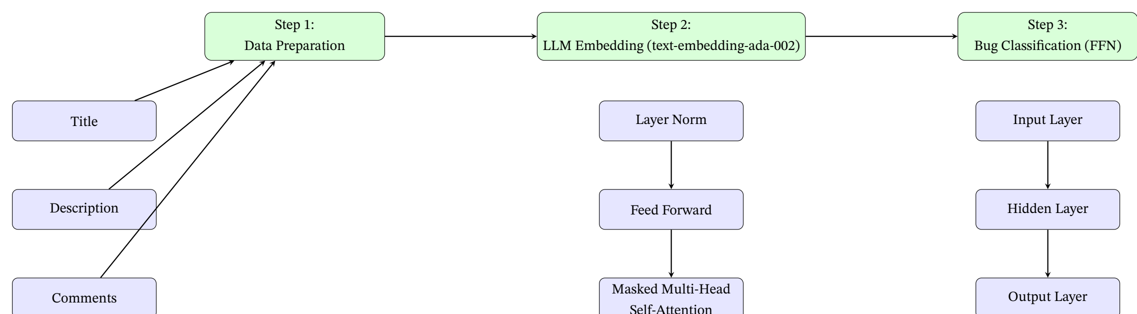

Figure 2.5: Overview of the LLM-BRC framework: bug report preparation, embedding using text-embedding-ada-002, and classification via feed-forward neural network.

The framework is evaluated through a three-level classification pipeline as depicted in Figure 2.5. At the first level, LLM-BRC filters non-bugs such as enhancement or duplicate reports. Then, it distinguishes Bohrbugs from Mandelbugs, and finally classifies Mandelbugs into aging-related (ARBs) and non-aging (NAMs) based on error propagation complexity.

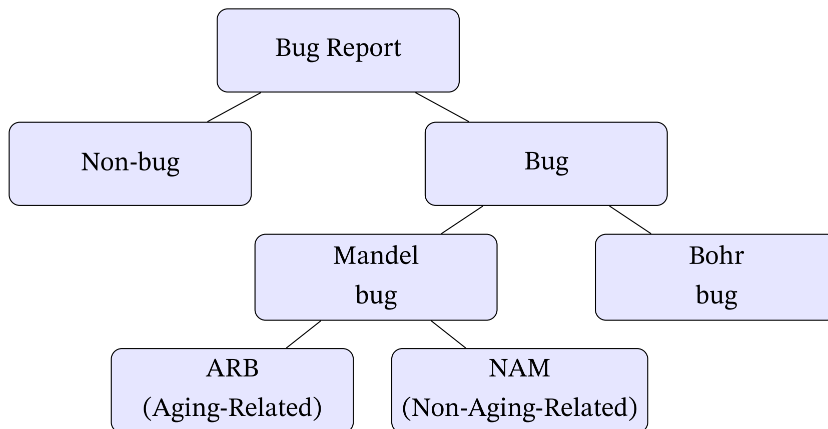

Figure 2.6: Three-level hierarchical classification of bug reports in LLM-BRC.

The results of LLM-BRC outperformed DeepSIM, the prior state-of-the-art, by 17.21%–69.15% in accuracy, precision, recall, and F1-score across all categories. For example, in the TensorFlow bug/non-bug classification task, LLM-BRC achieved 92.56% accuracy and 92.73% F1-score. Additionally, the study revealed that incorporating all report components (title, description, and comments) leads to the best performance, especially for complex bug types.

Table 2.2: LLM-BRC Classification Performance across Frameworks and Tasks

Framework	Task	Accuracy	Precision	Recall	F1-score
TensorFlow	Bug/Non-Bug	92.56%	92.46%	93.02%	92.73%
TensorFlow	BOH/MAN	98.47%	98.70%	99.47%	99.08%
TensorFlow	ARB/NAM	98.75%	98.75%	98.75%	98.75%
MXNET	Bug/Non-Bug	94.74%	94.86%	94.94%	94.88%
Paddle	ARB/NAM	92.00%	92.50%	96.67%	94.29%

To further analyze model effectiveness, the authors conducted ablation studies using multiple classifiers and embedding models. They found that:

- The FFN classifier significantly outperformed SVC, RFC, LOG, GNB, DTC, KNN, and even MLP.

- The `text-embedding-ada-002` embeddings led to performance improvements of up to 44.3% over BERT and 36.07% over word2vec.

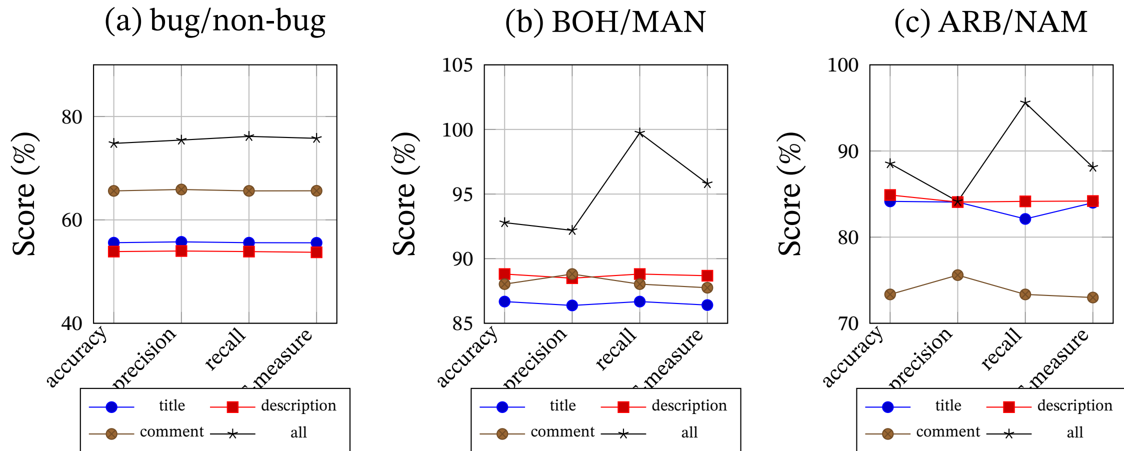

Figure 2.7: Ablation study comparing bug report components on different classification tasks (LLM-BRC)

Overall, LLM-BRC provides compelling evidence that OpenAI's embedding models combined with FFNs can serve as an effective and scalable solution for real-world bug classification in deep learning ecosystems.

2.2 Automatic Bug Classification Techniques

Automatic Bug Classification System using Hybrid NLP and ML Methods

Kelin et al. [12] proposed a comprehensive and structured automatic bug classification framework, known as the Automated Bug Tracking System (ABTS), which integrates natural language processing (NLP) with machine learning (ML) techniques to improve software maintenance productivity. The authors argue that the high volume and diverse nature of bug reports, particularly in large-scale or globally distributed development environments, necessitate the adoption of automated bug classification mechanisms to effectively assign issues to the appropriate development teams [12].

The ABTS framework operates on both textual and categorical attributes extracted from bug reports, including fields such as title, summary, severity, priority, platform, and product version. It supports classification into multiple bug types—logical, GUI, enhancement, and analysis bugs—allowing fine-grained assignment and prioritization.

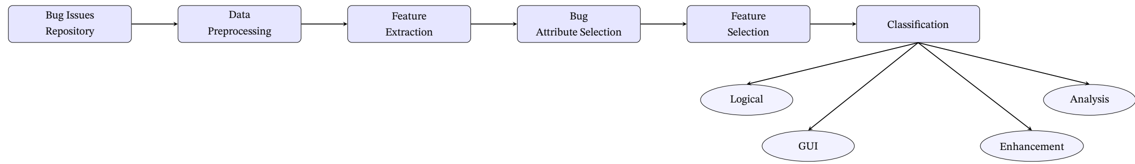

Figure 2.8: Architecture of the ABTS system: from bug report preprocessing to classification into logical, GUI, enhancement, and analysis bugs.

Methodology Overview

The classification pipeline consists of the following components:

- **Preprocessing:** Bug reports are segmented into sentences, tokenized, stopwords removed, and terms lemmatized using NLP tools.
- **Feature Extraction:** Using TF-IDF and information gain methods, more than 200 frequent terms were selected as features. Terms like *button*, *pixel*, and *border* identify GUI bugs, while *optimize* and *compile* relate to enhancement bugs.
- **Attribute Selection:** Eight attributes are used: three textual (title, detailed summary, expert suggestion) and five categorical (priority, product, version, severity, platform).
- **Feature Selection:** A two-phase selection (filtration + wrapper) identifies optimal subsets to improve classifier performance.
- **Classification:** Three machine learning models—Naïve Bayes, Random Forest, and XGBoost—are applied for classification.

Experimental Results

The evaluation used 40,000 bug reports from Bugzilla and Google Chromium repositories (20,000 each). A comparative analysis across three ML classifiers and various feature sets was performed both before and after feature selection.

Table 2.3: Performance evaluation using textual, categorical, and combined attributes (before feature selection)

Classifier	**Textual**				**Categorical**				**Combined**			
	Acc	Prec	Rec	F1	Acc	Prec	Rec	F1	Acc	Prec	Rec	F1
Naïve Bayes	0.67	0.774	0.77	0.768	0.57	0.572	0.57	0.562	0.652	0.662	0.64	0.682
Random Forest	0.817	0.826	0.818	0.818	0.572	0.568	0.572	0.566	0.808	0.822	0.818	0.808
XGBoost	**0.882**	**0.898**	**0.852**	**0.814**	0.636	0.646	0.696	0.590	0.856	0.862	0.856	0.844

Post feature selection, performance improved markedly across all models, with XGBoost showing superior results:

Table 2.4: Performance with selected features using wrapper and filtration technique

Classifier	**Accuracy**	**Precision**	**Recall**	**F-Measure**
Naïve Bayes	0.768	0.789	0.772	0.773
Random Forest	0.893	0.885	0.902	0.891
XGBoost	**0.952**	**0.965**	**0.958**	**0.942**

Comparative Benchmarking

The method was benchmarked against other state-of-the-art techniques, including [14], and [8]. XGBoost outperformed all others in every metric, achieving the highest accuracy, precision, recall, and F-measure.

Table 2.5: Comparison with state-of-the-art bug classification techniques

Approach	**Accuracy**	**Precision**	**Recall**	**F-Measure**
Otoom et al., 2019	0.930	0.830	0.850	0.830
Sarkar et al., 2019	0.790	0.780	0.790	0.780
Fang et al., 2022	0.937	0.912	0.911	0.901
Proposed (XGBoost)	**0.952**	**0.965**	**0.958**	**0.942**

The ABTS demonstrates a highly scalable and practical framework for automating bug classification in real-world settings. Its hybrid approach, leveraging both structured and unstructured data, and its use of ensemble methods like XGBoost ensure robustness, generalizability, and high accuracy. These findings make ABTS a strong candidate for integration into industrial bug tracking and triaging systems.

Bug Report Classification with Ensemble Learning

Yılmaz et al. introduced a significant advancement in automated bug classification by addressing one of the most underexplored areas in the field—commercial, closed-source software [18]. Their work presents the Turkish Software Bug Reports (TSBR) datasets, a rare collection of commercial bug reports compiled and curated for academic research. These datasets serve as the foundation for evaluating multiple classification tasks, including module-level prediction, developer assignment, and issue/feature categorization.

Unlike open-source repositories, closed-source environments pose unique challenges: testers often operate with limited access to internal code structures and rely heavily on black-box testing. To address these limitations, the authors enriched the TSBR

datasets with both textual features (e.g., bug descriptions) and categorical fields (e.g., module name, configuration unit). They emphasized practical realism during dataset construction by tackling issues such as label imbalance and reporting noise.

Methodology Overview

The study evaluates a diverse set of classifiers including traditional machine learning (SVM, Random Forest, Logistic Regression), deep learning architectures (GRU, CNN, LSTM), and transformer-based models (BERTurk, mBERT).

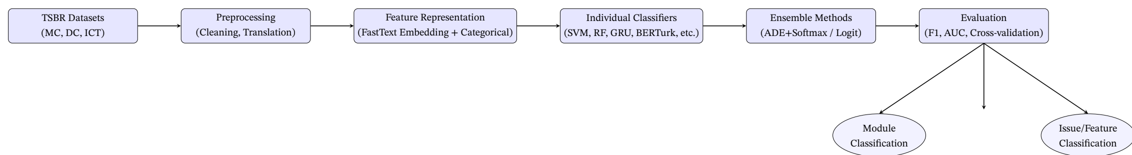

Figure 2.9: Workflow of ensemble-based classification approach using ADE for TSBR datasets (module, developer, and issue/feature classification).

Each model was first assessed individually, and then integrated into ensemble configurations. The authors proposed two novel ensemble strategies under the Advanced Deep Ensemble (ADE) framework:

- **ADE+softmax:** Combines output probabilities using weighted softmax voting.
- **ADE+logit:** Utilizes raw logits for weighted fusion based on each model's performance (e.g., F1 score).

These ensemble methods yielded superior performance in most scenarios, demonstrating higher classification robustness and resilience to class imbalance. BERTurk, in particular, achieved the best performance among individual models, while ADE+logit consistently produced the highest F1 scores in ensemble settings.

Empirical Results

The authors evaluated the proposed methods on three TSBR datasets: TSBR-MC (module classification), TSBR-DC (developer classification), and TSBR-ICT (issue classification task). Across all tasks, ADE-based ensembles outperformed baseline and standalone models, particularly in complex multi-class environments. The results were supported by:

- ROC AUC analysis for model discrimination capability.
- Precision, recall, and F1-score for accuracy and robustness.
- 5-fold cross-validation for generalization.

- Statistical t-tests for significance validation.

Insights and Future Directions

A noteworthy insight from the study is the trade-off between ensemble complexity and effectiveness. While ensemble methods generally improved performance, including too many weak learners degraded the final accuracy. This emphasizes the importance of careful ensemble curation.

The paper concludes by identifying future directions, including:

- Integration of generative models for low-resource scenarios.
- Data augmentation to mitigate class imbalance.
- Domain-specific pretraining for improved representation learning in commercial software contexts.

Overall, this study is one of the first to systematically apply ensemble deep learning to closed-source bug classification. It offers a valuable benchmark dataset and demonstrates how ensemble methods can bridge gaps in multilingual and domain-specific classification tasks, setting a strong foundation for future research in industry-scale bug triaging systems.

2.3 Bug Taxonomy in Deep Learning Software

A Comprehensive Study on Deep Learning Bug Characteristics

Johirul , Nguyen, Rangeet , et al. present one of the most comprehensive investigations into bug characteristics in deep learning software [11]. Unlike most prior works that focus solely on open-source repositories or implementation-level bugs, this study bridges Stack Overflow discussions and GitHub bug-fix commits across five widely used deep learning frameworks: TensorFlow, Keras, Theano, Torch, and Caffe .

Dataset and Scope

The authors curated a dataset of 2,716 Stack Overflow posts and 500 bug-fix commits from GitHub. These were manually annotated by trained raters to classify bug types, root causes, and impacts using an extended version of existing taxonomies. The classification was validated through multiple pilot studies, achieving over 90% inter-rater agreement (Cohen's Kappa) Rau and Shih [16] after iterative refinement.

Methodology Overview

The study focuses on six research questions, covering bug types (RQ1), root causes (RQ2), effects (RQ3), pipeline stages (RQ4), bug commonality across libraries (RQ5), and bug evolution over time (RQ6). The authors leveraged a 7-stage deep learning pipeline (data collection, preparation, model choice, training, evaluation, tuning, and prediction) to contextualize bug occurrences across development phases.

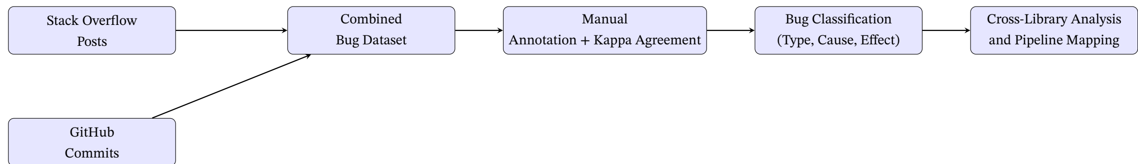

Figure 2.10: Workflow for analyzing deep learning bug characteristics using Stack Overflow and GitHub data.

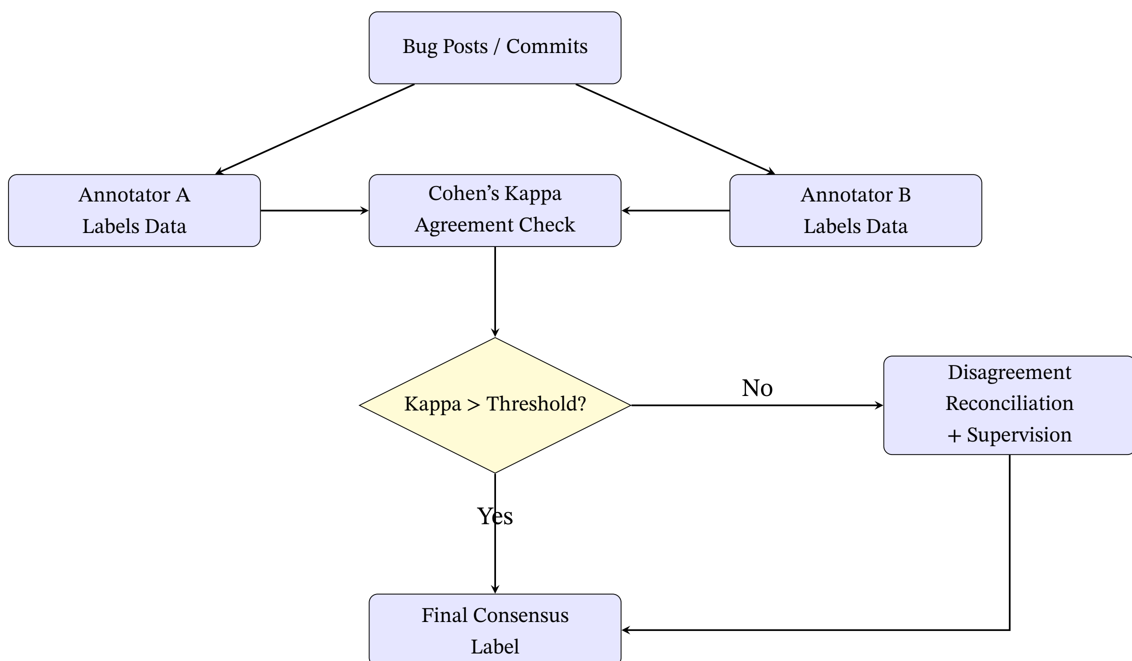

Figure 2.11: Workflow of bug labeling and Cohen's Kappa agreement resolution used in [11] [16].

Key Findings

- **Bug Types (RQ1):** Data Bugs and Structural Logic Bugs dominate, appearing in 26–43% of samples. Structural Bugs typically arise from incorrect model design or misalignment between data shapes.
- **Root Causes (RQ2):** The two most common root causes are Incorrect Model Parameters (IPS, 24–62%) and Structural Inefficiency (SI, 3–53%), followed by issues like API misuse and absence of type checking .

- **Effects (RQ3):** Over 66% of bugs lead to crashes, followed by incorrect functionality and poor performance. Tensorflow and Keras were most crash-prone.
- **Pipeline Stages (RQ4):** The Data Preparation stage is the most bug-prone (32%), followed by Training (27%) and Model Choice (23%).
- **Common Patterns (RQ5):** A strong correlation (up to 0.96) exists between TensorFlow and Keras bug types. Antipatterns such as Input Kludge, Copy-Paste Programming, and Spaghetti Code are frequent.
- **Bug Evolution (RQ6):** Structural logic bugs have increased steadily from 2015–2018. Data bugs are declining, likely due to better data-handling libraries like Pandas and NumPy.

The authors conclude that a majority of bugs in deep learning software arise from either improper data handling or flawed model specification. Their findings strongly advocate for improved data validation tooling and model-architecture guidance tools. The study also sets the stage for future work in bug repair, automated debugging, and root cause mitigation strategies in deep learning software development.

2.4 Comparative Analysis of Bug Classification Approaches

Research Timeline and Technological Progression

Bug classification research has evolved markedly from 2019 to 2025. Figure 2.12 illustrates this progression. Early works focused on manual categorization and conventional ML. With time, transformer models and fine-tuned large language models (LLMs) have redefined the state of the art.

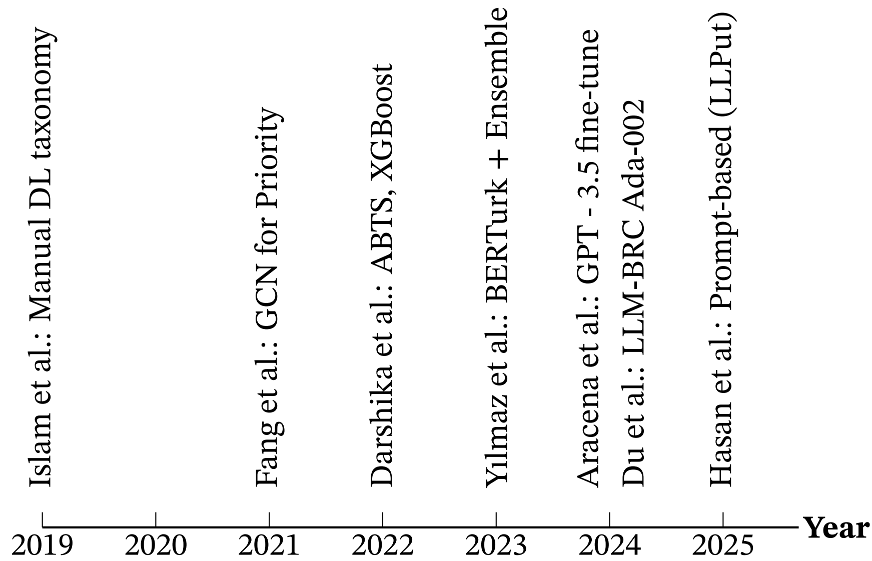

Figure 2.12: Timeline of prominent bug classification research.

Methodology Matrix Overview

To compare these approaches systematically, Table 2.7 summarizes the key methodological aspects of representative studies. It contrasts dataset source, model type, domain context, and evaluation metrics.

Table 2.6: Comparative Summary of Bug Classification Methods

Approach	Year	Model Type	Data Domain	F1-score
Islam et al.	2019	Manual Categorization	Stack Overflow + GitHub Commits	–
Fang et al.	2021	Graph Conv. Net	OSS bug reports	93%
Kelin et al.	2022	XGBoost + TF-IDF	Bugzilla, Chromium	94.2%
Yılmaz et al.	2023	BERTurk + ADE	Closed-source Turkish (TSBR)	>90%
Aracena et al.	2024	GPT-3.5 (Fine-tuned)	GitHub Issues	89.3%
Du et al.	2024	Ada-002 + FFN	DL framework bugs	98.7%
Hasan et al.	2025	LLaMA/Qwen Prompting	Bug reports (input extraction)	BLEU 0.5 (62%)

2.4.1 Unique Features of Author's Approach

While existing works achieve respectable performance on narrow tasks, they often suffer from limited scope, reliance on hand-crafted features or domain-specific training data, and lack of robustness to new issue types.

Table 2.7: Comparative Summary of Bug Classification Methods

Approach	Year	Model Type	Data Domain
Zero-shot Prompting	2025	GPT-4.1, Claude-3.5-Sonnet, Gemini-2.5-pro	GitHub Issues + Own Dataset (LLM-specific issues, 1,500 entries)
RAG	2025	GPT embedding	GitHub Issues
Few-shot Prompting	2025	GPT-4.1, Claude-3.5-Sonnet, Gemini-2.5-pro	GitHub Issues
Zero Shot multi-Peer 1 Judge	2025	GPT-4.1, Claude-3.5-Sonnet, Gemini-2.5-pro	GitHub Issues + Own Dataset
Zero Shot multi-Peer 1 Judge with K-sampling	2025	GPT-4.1, Claude-3.5-Sonnet, DeepSeek-v3.1	GitHub Issues + Own Dataset

Our method breaks from these constraints by assembling a multi-model generative ensemble, leveraging retrieval augmented prompts and advanced sampling to perform zero-shot labeling on a newly curated dataset of LLM-specific issues, and evaluating consensus with kappa statistics. This combination of innovations positions our approach as a versatile and practical alternative to traditional bug-classification pipelines.

Feasibility and Practicality Comparison

In terms of deployment feasibility, approaches vary across three major dimensions: infrastructure requirements, training and inference cost, and data sensitivity.

Infrastructure: Traditional models like ABTS (XGBoost) are lightweight and easily deployable on commodity hardware. Transformer-based models like BERTurk and mBERT need a GPU for training but can be optimized for inference. LLM-API solutions such as GPT-3.5 require cloud access, while embeddings from Ada-002 offer a middle ground: low-latency API calls with precomputed vector representations.

Training Data: Traditional methods like ABTS require thousands of labeled bug reports, often necessitating manual preprocessing and engineering. In contrast, LLM-based models benefit from few-shot or zero-shot capabilities, reducing annotation overhead. LLM-BRC, for instance, reached SOTA performance using only 3k examples.

Runtime Cost and Speed: ABTS and BERTurk offer near-instant predictions, making them viable for integration in CI/CD systems. Generative LLMs like LLPut incur higher latency (1–3s per input) and may require GPU acceleration. GPT-based fine-tuning adds cloud service cost per inference, potentially limiting scalability for startups.

Privacy and Portability: LLM-API models raise concerns for sensitive data unless hosted on private endpoints. BERTurk and similar models offer greater control for enterprises by enabling on-premise deployment. Prompt-based open-source models like LLaMA offer a cost-effective alternative but may require expertise to manage.

Maintenance and Complexity: Ensemble models such as ADE require managing multiple learners, adding to maintenance load. LLM APIs shift maintenance to the provider but introduce black-box dependencies. Prompt-based LLMs reduce maintenance but require careful prompt design.

Future Research Outlook

Future work can address several open challenges:

- **Multimodal Pipelines:** Integrating code snippets, stack traces, and screenshots with text-based classification.
- **Prompt Engineering Automation:** Using reinforcement learning or search to optimize prompts for generative tasks.

- **LLM-Based Explanation:** Coupling classification with natural language rationale to improve developer trust.
- **Online Learning:** Incorporating human-in-the-loop updates via bug reclassification feedback.
- **Distillation and Quantization:** Compressing large LLMs to smaller models for edge or embedded deployment.
- **Synthetic Data Generation:** Using LLMs to balance class distribution and simulate rare bug types.
- **Cross-Project Transferability:** Building generalizable models that retain performance across repositories.
- **Low-Resource Languages:** Expanding beyond English-language bug reports with multilingual embeddings.
- **Continuous Deployment Readiness:** Packaging models into microservices or Docker containers with REST APIs.

These directions aim to improve model performance, reduce cost, and ensure seamless integration of intelligent classifiers in real-world software maintenance workflows.

3 Methodology

This chapter outlines the methodology followed in our research, including dataset collection, labeling pipelines, evaluation design, and modeling approaches. We present three major experimental pipelines: zero-shot labeling of GitHub issues using multiple LLMs, Stack Overflow bug-type labeling, and a comparative run with fine-grained labels from our very own dataset. Our goal is to evaluate the feasibility and accuracy of LLM-based labeling from community-sourced and real-world unstructured data.

3.1 Github Issues Dataset

To test pipelines, we ran auto labeling on a curated GitHub issues dataset[13]. We extracted 433 unique rows and 4 columns: ID, Body, Title, and Labels.

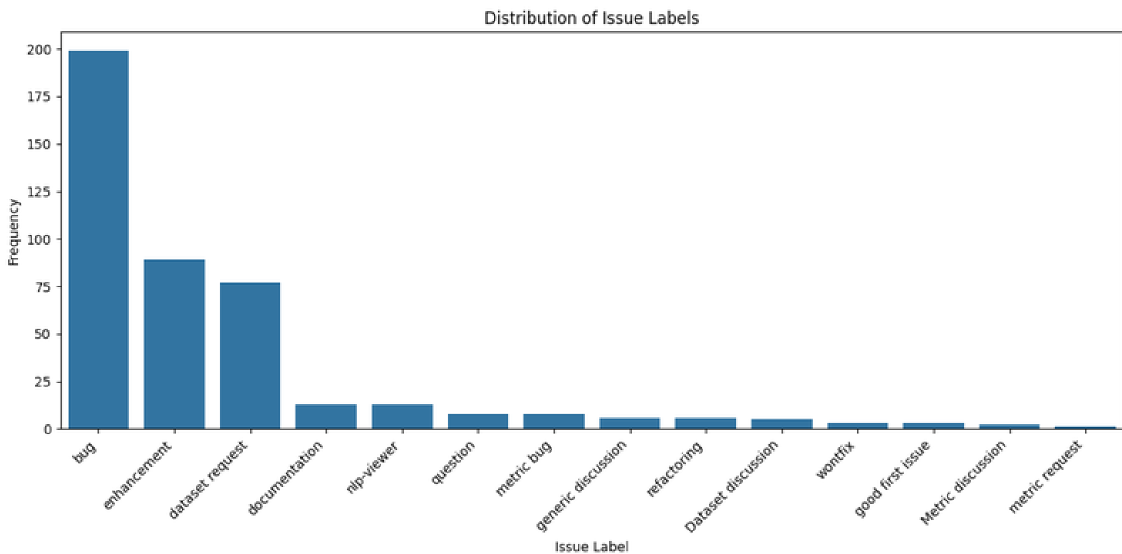

Figure 3.1: Distribution of Issue Labels

3.2 Pipeline

The labeling pipeline developed in this study was designed to progressively evaluate and refine the auto-labeling capabilities of large language models (LLMs) under a variety of configurations.

We began by assessing the individual performance of state-of-the-art models through zero-shot prompting, providing a baseline for how well generative models can classify

issue reports without domain-specific fine-tuning. Building upon this baseline, we incorporated retrieval-augmented generation (RAG) to contextualize prompts with relevant external knowledge, thereby testing whether supplemental information improves classification fidelity. Recognizing the bias inherent in single-model outputs, we subsequently advanced to ensemble-based adjudication frameworks, specifically the Zero-Shot 2-Peer-1-Judge and 3-Peer-1-Judge paradigms, where multiple peer models provide independent labels and a dedicated judge model resolves conflicts.

Finally, to better understand and control the stochastic nature of generative decoding, we conducted systematic experiments with temperature scaling and top-k sampling, tuning these parameters to balance determinism with diversity in predicted labels. Collectively, these stages form a comprehensive pipeline that integrates baseline evaluation, contextual augmentation, ensemble decision-making, and parameter sensitivity analysis.

Individual LLMs

To establish a baseline, we evaluated the performance of individual large language models (LLMs) under a zero-shot prompting regime, without fine-tuning or additional contextual augmentation. Three state-of-the-art models—GPT-4.1, and Claude—were selected for this experiment. Model outputs were directly compared against ground-truth labels using standard classification metrics: precision, recall, and F1-score.

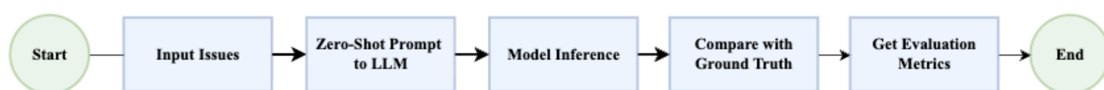

Figure 3.2: Baseline Zero-Shot Prompting

However, this gain came with trade-offs: the retrieval process occasionally introduced irrelevant or noisy passages, which diluted the prompt and reduced precision for certain categories.

Zero Shot 2 Peer 1 Judge

In this pipeline, two peer models independently labeling a given dataset, while a judge model arbitrates differences in labeling between peers. Multiple iterations are executed by increasing the temperature slightly every time to increase the Cohen Kappa

which is measured to determine the similarity between two raters. Also the LLMs only re-label those issues with which the LLMs dont́ agree. The effectiveness of the judge model in combining peer predictions ensures the final labeling has higher accuracy through the reduction of errors that might have been made by any contributing peer.

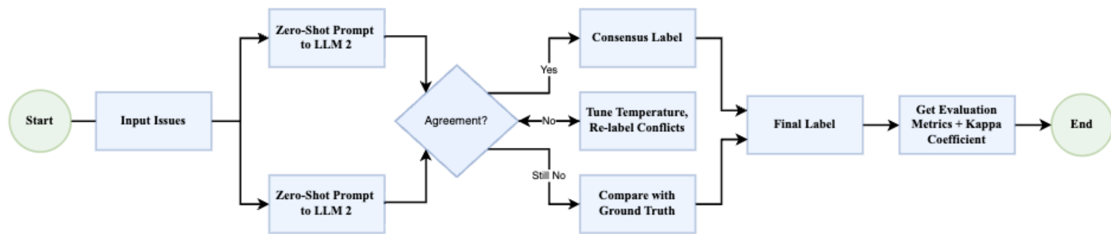

Figure 3.3: Zero-Shot Two Peer One Judge Ensemble

2 Peer 1 Judge with K-sampling

In this setup, we employed an ensemble architecture consisting of two peer LLMs and one judge, enhanced with k-sampling to introduce diversity in candidate predictions. Each issue was independently classified by the two peer models using enriched zero-shot prompts. Instead of producing a single deterministic label, both peers performed k-sampling (e.g., $k = 3$ with temperature 0.3), generating multiple plausible labels for the same input. This stochastic process allowed us to capture uncertainty and variation in model predictions.

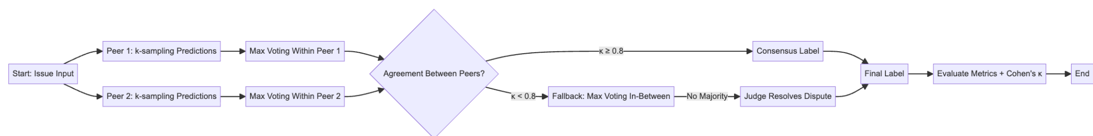

Figure 3.4: Two Peer One Judge with K-sampling

To resolve these multiple outputs, we implemented a two-stage majority voting strategy:

Max Voting Within. Each peer model generated three candidate labels. A max voting rule was applied within these samples to determine the peer's final output. If two or more of the three predictions agreed, that label was selected. In the rare case

where all three predictions differed, the system defaulted to the first generated label to ensure determinism.

Max Voting In-Between. After consolidating predictions from both peers, inter-peer agreement was evaluated using Cohen's κ. When κ reached a threshold of 0.8 or higher, the system considered the peers sufficiently consistent and selected the majority label across their outputs. If κ failed to converge beyond multiple iterations (i.e., agreement stagnated below threshold), a fallback max voting rule was applied: the label with the highest frequency across the peer outputs was chosen. In the case where all three consolidated peer labels were distinct and no majority existed, the judge LLM was invoked as a tie-breaker to provide the final classification.

Role of the Judge. The judge acted as an arbitration mechanism only when peer consensus could not be reached through intra- or inter-model max voting. This design ensures that the judge is not overused, but rather reserved for the most ambiguous cases where peer disagreement persists. The inclusion of the judge thus mimics a human-in-the-loop adjudication step in annotation pipelines.

The 2 Peer 1 Judge with k-sampling pipeline combines stochastic sampling, majority-based consolidation, and an arbitration mechanism to increase reliability of auto-labeling. By layering voting within peers, voting between peers, and a judge fallback, the approach improves robustness to sampling variance, captures diverse labeling possibilities, and ensures systematic resolution of disagreements. This consensus-driven strategy reduces sensitivity to outlier predictions and enhances the quality of the final labels, particularly in cases with class imbalance and ambiguous issue descriptions.

Zero Shot 3 Peer 1 Judge

This setup builds on the previous by adding a peer, making it a three-peer, one-judge decision-making paradigm. Increasing the number of peers should add greater robustness to the labeling by introducing more diverse opinions before the final judgment. After several iterations with temperature adjustments, Fleiss Kappa is measured every time to determine the similarity between 3 raters. After that, the maximum voting phase is reached, where the label agreed upon by two LLMs is selected. The judge model is then introduced to resolve disagreements among the LLMs, ensuring higher accuracy for the remaining issues.

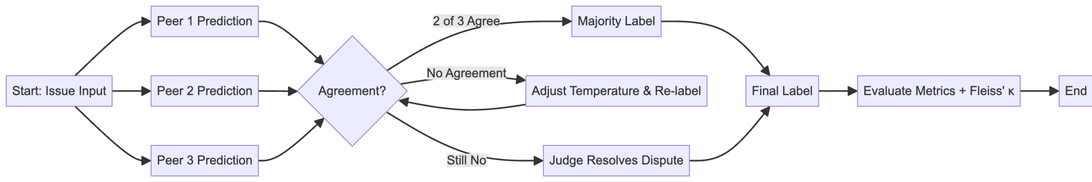

Figure 3.5: Three Peer One Judge with Temperature Tuning

3 Peer 1 Judge with K-sampling

This setup extends the ensemble framework to three peer LLMs with one judge, augmented by k-sampling to increase diversity and robustness. Each issue is independently classified by three parallel LLM agents (the peers) using zero-shot prompts with enriched instructions. Rather than producing a single deterministic label, each peer generates multiple candidate predictions through k-sampling (e.g., $k = 3$ with temperature 0.3). This process introduces stochastic variation and captures alternative plausible interpretations of an issue.

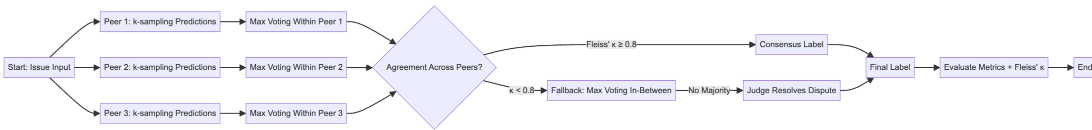

Figure 3.6: Three Peer One Judge with K-sampling

Max Voting Within. For each peer, the three sampled outputs were consolidated using majority voting. If at least two of the sampled predictions agreed, that label was taken as the peer's final decision. In cases where all three predictions differed, the first sampled label was selected as a deterministic fallback. This ensured that each peer contributed one consolidated label to the ensemble.

Max Voting In-Between. After within-peer consolidation, the three peers' outputs were compared. To evaluate agreement among them, Fleiss' κ was used as the inter-rater reliability measure. When κ exceeded the threshold of 0.8, the peers were considered to have reached sufficient consensus, and the majority label across the three outputs was selected. If κ stagnated below this threshold, a fallback max voting rule was applied, where the most frequent label among the three peers was chosen. In situations where all three peers disagreed (i.e., three distinct labels), the system escalated the decision to the judge.

Role of the Judge. The judge LLM acted as an arbitration layer when no consensus could be achieved across the peers. By evaluating the peers' conflicting predictions in context, the judge selected the most consistent and semantically appropriate label. This mechanism prevents deadlock in the ensemble and ensures that each issue receives a final classification, even under maximum disagreement.

The 3 Peer 1 Judge with k-sampling pipeline provides an additional layer of redundancy and robustness compared to the two-peer design. By combining within-peer consolidation, inter-peer agreement evaluation using Fleiss' κ, and judge-based arbitration, the pipeline mimics a consensus-driven multi-annotator process. This setup reduces sensitivity to stochastic variance, mitigates bias toward frequent categories, and produces more reliable labels in cases of ambiguity or class imbalance.

Retrieval-Augmented Generation (RAG)

To address the limitations of pure zero-shot prompting, we designed a Retrieval Augmented Generation (RAG) pipeline that integrates external knowledge into the labeling process. Specifically, we constructed a semantic index using GPT-based embeddings, which allowed us to retrieve contextually relevant information for each issue. For every classification query, the prompt was enriched with synthetic contextual data corresponding to the candidate bug categories, including repository descriptions, related issue discussions, and documentation excerpts. This augmentation was intended to ground the LLMs in domain-relevant context, reducing ambiguity in category assignment.

Few-Shot Prompting and RAG Experiments

In addition to zero-shot prompting, we also experimented with few-shot prompting and retrieval-augmented generation (RAG) on the GitHub issues dataset. The intuition behind few-shot prompting is that providing the model with examples of input-output label pairs should help it generalize better to new unseen issues. Similarly, RAG augments the prompt with retrieved contextual passages to ground the classification in external knowledge.

However, in our experiments, both approaches yielded very limited improvements. In fact, few-shot prompting performed worse than zero-shot. This is likely because the GitHub issues dataset contains highly diverse and open-ended issue descriptions that do not follow consistent patterns. Few-shot prompting, by construction, depends on the model's ability to identify and generalize from recurring patterns in the examples provided. Since the issues are heterogeneous and context-specific, the provided few-

shot exemplars did not generalize effectively to the rest of the dataset, thereby leading to poor accuracy.

RAG also failed to produce significant gains. While retrieval occasionally surfaced useful information, it often introduced irrelevant or noisy passages into the prompts, which diluted the classification signal. This resulted in confusion for the model, lowering precision for several categories. Since our experiments showed that these methods neither improved agreement scores nor accuracy beyond the zero-shot baselines, we do not include detailed metrics or tables here.

Overall, these findings highlight that in tasks such as GitHub issue labeling, where issue descriptions are highly varied and do not follow repetitive patterns, few-shot prompting and RAG are not reliable approaches. Instead, ensemble-based adjudication pipelines with zero-shot prompting proved to be more effective and consistent.

3.3 GitHub Issue Labeling with Multiple LLMs

To test zero-shot [4] labeling, we ran multiple rounds of LLM-based labeling on a curated GitHub issues dataset [13]. The following taxonomy describes the 15 standardized categories identified within the dataset. Table 3.1 outlines each category along with a representative example to clarify its scope.

Table 3.1: Label Taxonomy Standardization

Category	**Description**	**Example**
Dataset discussion	Discussions about existing datasets or dataset usage.	*Should we update the dataset to include newer samples?*
Metric discussion	Conversations regarding evaluation metrics or performance measures.	*Should we add F1-score in addition to accuracy?*
Bug	Reports about broken functionality or unintended behavior.	*Training script crashes at epoch 5.*
Dataset request	Requests for new datasets or data sources to be added.	*Please add a Spanish language dataset.*
Documentation	Issues related to missing, incorrect, or unclear documentation.	*The README is missing setup instructions.*
Enhancement	Suggestions to improve or add new functionality.	*Add support for TensorFlow 2.x.*
Generic discussion	Open-ended conversations not fitting into specific categories.	*Thoughts on migrating the codebase to Rust?*
Good first issue	Simple tasks suitable for first-time contributors.	*Fix typo in homepage title.*
Metric bug	Problems specifically with evaluation or performance metrics.	*Accuracy calculation doesn't exclude padding tokens.*
Metric request	Requests for additional metrics to be implemented.	*Add BLEU score computation for translation tasks.*
NLP viewer	Issues related to visualization tools for NLP models or data.	*Model attention maps not displaying correctly.*
Question	Users asking for clarification or help.	*How do I fine-tune this model with a custom dataset?*
Refactoring	Restructuring code without adding features.	*Split utility functions into a separate module.*
Wontfix	Acknowledged issues that maintainers decided not to fix.	*Feature not aligned with project goals.*
Other	Miscellaneous announcements or experimental ideas.	*Miscellaneous announcements or experimental ideas.*

3.4 Results & Observations

Table 3.2: Zero Shot: 2 Peers, 1 Judge (GitHub Issues)

Metric / Step	Value
Peer Models	GPT-4.1, Claude-3.5-Sonnet
Judge Model	Claude-3-Opus
Initial Cohen's Kappa	0.8468
Temperature Adjustments	0.22, 0.28, 0.32
Final Cohen's Kappa	0.8890
Final Consensus Accuracy	**81.29%**

Table 3.3: Zero Shot: 2 Peers, 1 Judge with K-sampling (GitHub Issues)

Metric / Step	Value
Peer Models	GPT-4.1, Claude-3.5-Sonnet
Judge Model	Claude-3-Opus
Initial Cohen's Kappa	0.8400
Temperature Adjustments	0.22, 0.28, 0.32
Final Cohen's Kappa	0.8790
Final Consensus Accuracy	**83.37%**

Table 3.4: Zero Shot: 3 Peers, 1 Judge (GitHub Issues)

Metric / Step	Value
Peer Models	GPT-4.1, Claude-3.5, GPT-3.5-turbo
Judge Model	Claude-3-Opus
Initial Fleiss' Kappa	0.7637
Temperature Adjustments	0.22, 0.28
Final Fleiss' Kappa	0.7870
Final Dataset Accuracy	**81.99%**

Table 3.5: Zero Shot: 3 Peers, 1 Judge with K-sampling (GitHub Issues)

Metric / Step	Value
Peer Models	GPT-4.1, Claude-3.5, DeepSeek-v3.1
Judge Model	Claude-3-Opus
Individual Accuracy (GPT / Claude / DeepSeek)	82.22 / 81.76 / 81.09
Initial Cohen's Kappa	0.8468
Final Cohen's Kappa	0.8890
Final Consensus Accuracy	**82.68%**

Table 3.6: Zero Shot: 2 Peers, 1 Judge with K-sampling (StackOverflow)

Metric / Step	Value
Initial Cohen's Kappa (OpenAI vs. Claude)	0.4738
Iteration 1 Kappa	0.7698
Iteration 2 Kappa	0.7844
Iteration 3 Kappa	0.7970
Final Disagreements (Iteration 3)	208
Matching Original Labels	**51.97%**

3.5 Prepared Dataset

To construct the Stack Overflow dataset, we implemented a targeted crawling strategy focused on Large Language Model (LLM) and Natural Language Processing (NLP) frameworks.

Crawling Strategy and Filters

The data collection was guided by a set of technical keywords including: *transformers, langchain, llamaindex, huggingface, llama, bert, gpt, text generation, sentence-transformers, rag, finetune, quantization, model deployment, openai, chatgpt, prompt engineering, embedding, inference, peft, auto-gpt, retrieval augmented generation, ollama, mistral, mixtral, and falcon.*

To ensure high-quality data, the following filters were applied during the collection process:

- **Keyword Requirement:** The post title must contain at least one of the specified keywords.
- **Temporal Constraint:** Posts must have been created after January 1, 2023.
- **Quality Threshold:** Posts must have a score ≥ 1 and at least one answer.
- **Deduplication:** All duplicate questions were removed to ensure data uniqueness.

Final Dataset Composition

The final dataset consists of **1,885 unique posts**, with **1,547 entries manually labeled** and stored in CSV format. Each entry in the dataset contains the following metadata fields:

Table 3.7: Stack Overflow Dataset Fields

Category	Fields Included
Identification	question_id, link, owner
Content	title, description, tags
Metrics	score, view_count, creation_date
Solutions	is_answered, answer_count, top_3_answers

Target Classes

- Model Loading Error
- Inference Failure
- Installation Issue
- Token Limit Exceeded
- CUDA/Device Error
- API Misuse
- Authentication/API Key
- Model Architecture Issue
- Other / Unknown

Class Descriptions, Examples, and Justification

We elaborate on each target class selected for this study, providing a description, a real-world example, and a justification for its inclusion:

- **Model Loading Error**
  Description: Errors that occur when trying to load a saved model file into memory.
  Example: "Error: Failed to load weights from checkpoint."
  Justification: Prevents any model functionality and is critical for operational readiness. One of the most appearing bug type in recent years.
- **Inference Failure**
  Description: Failures encountered during model prediction or evaluation.
  Example: "RuntimeError: tensor size mismatch during inference."
  Justification: Directly impacts usability and reliability of LLM applications. Impact direct business and use cases of LLM applications.
- **Installation Issue**
  Description: Problems arising during software installation or environment setup.
  Example: "pip install error: cannot find dependency for torch==2.0.x"
  Justification: Hinders adoption by practitioners trying to deploy or experiment with models.

- **Token Limit Exceeded**
  Description: Input text exceeds model's maximum token length limitation.
  Example: "Input exceeds 4096 token limit for GPT-3.5."
  Justification: Common constraint faced during real-world LLM use.

- **CUDA/Device Error**
  Description: GPU-related failures like memory overflows or device incompatibility.
  Example: "CUDA out of memory error during batch training."
  Justification: Crucial for any deployment that depends on hardware acceleration.

- **API Misuse**
  Description: Errors from incorrectly calling APIs or using wrong function arguments.
  Example: "TypeError: 'NoneType' object is not callable."
  Justification: Reflects user education gaps, implementation faults and usability flaws.

- **Authentication/API Key**
  Description: Problems with missing, invalid, or expired access credentials.
  Example: "401 Unauthorized: Invalid API key for OpenAI API."
  Justification: Directly affects system integration and access to essential services.

- **Model Architecture Issue**
  Description: Instability or failures due to incorrect model design.
  Example: "Mismatch between LSTM hidden size and input dimension."
  Justification: Important for researchers experimenting with custom models.

- **Other / Unknown**
  Description: Miscellaneous or ambiguous issues not fitting predefined classes.
  Example: "Unexplained segmentation fault when fine-tuning."
  Justification: Captures edge cases to maintain flexibility and completeness.

Rationale for Class Selection

The selection of these specific bug types was driven by a combination of empirical analysis and experience consensus among the authors. In the course of reviewing recent Stack Overflow posts, GitHub issues, and developer forums, we identified the most recurrent and impactful categories of problems faced during LLM development,

deployment, and usage.

Our selection criteria were guided by the following principles:

- **Severity:** Classes like "Model Loading Error" and "Inference Failure" represent issues that immediately halt system functionality and hence have high severity.
- **Frequency:** Types such as "Installation Issue" and "Token Limit Exceeded" occur frequently across LLM workflows, making them highly relevant.
- **Practical Impact:** Problems like "CUDA/Device Error" and "Authentication/API Key" directly affect the usability and scalability of LLM applications in production environments.
- **Developer-Centric Viewpoint:** From a practitioner's perspective, these categories align with the most common bottlenecks and pain points encountered during real-world development and maintenance cycles.
- **Coverage and Completeness:** The inclusion of "Other / Unknown" ensures that the classification scheme remains flexible enough to handle rare, unexpected, or evolving types of bugs.

Thus, our taxonomy reflects a balance between theoretical completeness and practical relevance, offering a robust basis for evaluating LLM behavior in realistic settings.

3.6 Workflow Summary

This section summarizes our end-to-end experimental pipelines using illustrative diagrams. Each approach (GitHub Issue Zero Shot labeling, StackOverflow bug labeling, and sample run with custom bug types) is visualized independently for clarity.

GitHub Zero-Shot Labeling via 2 peers

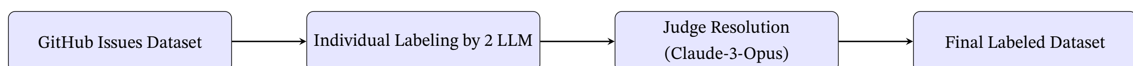

Figure 3.7: Zero-shot GitHub issue labeling using 2 LLMs and arbitration.

GitHub Zero-Shot Labeling via 2 peers with k-sampling

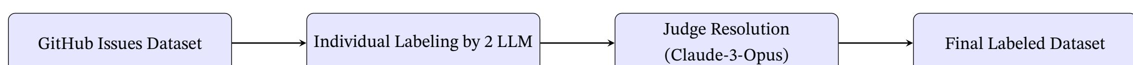

Figure 3.8: Zero-shot GitHub issue labeling using 2 LLMs and arbitration.

GitHub Zero-Shot Labeling via 3 peers

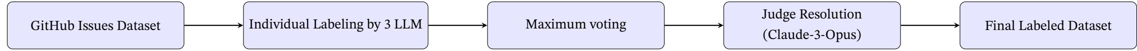

Figure 3.9: Zero-shot GitHub issue labeling using 3 LLMs and arbitration.

3.7 Mathematical Formula and Theories

3.7.1 Cohen's Kappa Coefficient

Introduction

Cohen's Kappa (κ) is a robust statistical measure that evaluates inter-rater agreement for categorical classifications, beyond what would be expected by chance [16]. It is widely used in labeling problems to assess the consistency between two annotators or models.

Mathematical Definition

Cohen's Kappa is mathematically defined as:

$$\kappa = \frac{p_o - p_e}{1 - p_e} \tag{3.1}$$

where:

- p_o = Observed agreement ratio (empirical probability of agreement)
- p_e = Expected agreement ratio assuming random labeling based on empirical priors

Implementation Details

In this study, we utilized the `cohen_kappa_score` function from the `sklearn.metrics` module in Python's Scikit-Learn library [5]. This function computes Cohen's kappa score based on the empirical probabilities of agreement.

- **Function Used:** `sklearn.metrics.cohen_kappa_score(y1, y2, labels=None, weights=None, sample_weight=None)`
- **Parameters:**
  - `y1`: Labels assigned by the first annotator.
  - `y2`: Labels assigned by the second annotator.

- `labels`: Optional list of labels to consider.
- `weights`: Type of weighting (`None`, `linear`, `quadratic`).
- `sample_weight`: Optional sample weights.

- **Return:** A float value between -1 and 1.

Example Usage

```
from sklearn.metrics import cohen_kappa_score

y1 = ["negative", "positive", "negative", "neutral", "positive"]
y2 = ["negative", "positive", "negative", "neutral", "negative"]

score = cohen_kappa_score(y1, y2)
print(score)  # Output: 0.6875
```

Interpretation

The resulting κ value can be interpreted as follows:

- $\kappa = 1$: Perfect agreement
- $\kappa = 0$: Agreement equivalent to chance
- $\kappa < 0$: Less than chance agreement

Interpretation scale:

Kappa Value	Strength of Agreement
< 0	Poor
$0.01 - 0.20$	Slight
$0.21 - 0.40$	Fair
$0.41 - 0.60$	Moderate
$0.61 - 0.80$	Substantial
$0.81 - 1.00$	Almost Perfect

Key Motivations for Cohen Kappa

- Traditional percent-agreement metrics do not consider that raters might agree by guessing. Cohen's kappa was introduced to address this problem
- Kappa scores range from 1 (less than chance agreement) to 1 (perfect agreement) and can be categorized into levels of agreement (poor, slight, fair, etc.).

- Since the two-peer pipeline involves exactly two LLMs, Cohen's kappa is the appropriate metric. It directly measures how much the two LLMs agree after adjusting for chance.
- By recalculating kappa after each iteration, the researchers can tell whether adjusting hyperparameters (e.g., sampling temperature) leads to better agreement.

Applications in This Research

Cohen's Kappa was applied extensively to measure the inter-annotator and model agreement in the labeling tasks. Specifically, it was used to validate consistency among LLM-based classifiers and resolve disagreements during label adjudication.

3.7.2 Fleiss' Kappa Coefficient

Introduction

Fleiss' Kappa (κ) is a statistical measure that evaluates the reliability of agreement between a fixed number of raters when classifying items into categories [17]. Unlike Cohen's Kappa, which is limited to two raters, Fleiss' Kappa generalizes the concept to multiple raters, making it ideal for scenarios involving more than two annotators [16].

Mathematical Definition

Fleiss' Kappa is mathematically defined as:

$$\kappa = \frac{\bar{P} - \bar{P}_e}{1 - \bar{P}_e} \tag{3.2}$$

where:

- $\bar{P}$ = Mean observed agreement across all items
- $\bar{P}_e$ = Mean expected agreement by chance

Implementation Details

In this study, we utilized the `fleiss_kappa` function from the `statsmodels.stats.inter_rater` module in Python's Statsmodels library. This function computes Fleiss' Kappa based on the raters' categorical assignments.

- **Function Used:** `statsmodels.stats.inter_rater.fleiss_kappa(table, method='fleiss')`
- **Parameters:**
  - `table`: A 2-D array-like object where rows represent subjects and columns represent the count of ratings per category.
  - `method`: Method to calculate chance agreement (`'fleiss'` for Fleiss' kappa, `'randolph'` for Randolph's uniform distribution-based kappa).
- **Return:** A float value representing Fleiss' or Randolph's Kappa statistic.

Example Usage

```
from statsmodels.stats.inter_rater import fleiss_kappa
import numpy as np

table = np.array([
    [0, 0, 10],
    [0, 2, 8],
    [1, 0, 9],
    [2, 2, 6]
])

score = fleiss_kappa(table, method='fleiss')
print(score)
```

Interpretation

The resulting κ value can be interpreted as follows:

- $\kappa = 1$: Perfect agreement
- $\kappa = 0$: Agreement equivalent to chance
- $\kappa < 0$: Less than chance agreement

Interpretation scale:

Kappa Value	Strength of Agreement
< 0	Poor
0.01 – 0.20	Slight
0.21 – 0.40	Fair
0.41 – 0.60	Moderate
0.61 – 0.80	Substantial
0.81 – 1.00	Almost Perfect

Key Motivations for Fleiss Kappa

- Fleiss' kappa extends the concept of Cohen's kappa to any fixed number of raters. This makes it ideal for the three-peer pipeline.
- It computes the average observed agreement across items and compares it to the average agreement expected by chance. A high Fleiss' kappa indicates that the group of LLMs is reliably producing consistent labels.
- Like Cohen's kappa, Fleiss' kappa ranges from 1 to 1 and can be interpreted using similar agreement categories.

Applications in This Research

Fleiss' Kappa was employed in scenarios where multiple LLM models were used to classify the same bug reports. It served as a metric to evaluate multi-model agreement and validate the robustness of the consensus-building process among multiple annotators.

3.8 Rationals behind the Coefficients

Cohen's kappa is used to evaluate agreement between two raters (or models) on categorical labels while correcting for chance agreement. In our pipeline, when comparing outputs of two LLMs (e.g., GPT-4.1 vs. Claude), Cohen's kappa is more informative than raw accuracy because it accounts for the probability that both models might agree just by guessing common categories (such as bug or enhancement). This provides a chance-corrected measure of consistency.

Fleiss's kappa generalizes this idea to more than two raters. In our 3-peer-1-judge pipeline, three LLMs independently assign labels before adjudication. Fleiss's kappa is therefore the most appropriate reliability coefficient, as it quantifies multi-rater consistency across categorical outputs. Like Cohen's kappa, it controls for agreement due

to chance, but it is specifically designed to handle the ensemble setting where three or more annotators label the same dataset.

Using kappa statistics allows the authors to quantitatively assess inter-annotator reliability in their auto-labeling framework. Instead of assuming that multiple LLMs agreeing on labels implies quality, they use Cohen's and Fleiss' kappa to determine whether agreement exceeds what would be expected by chance. Measuring kappa after each iteration helps them tune model parameters and select the best approach for producing a reliable labeled dataset. By integrating these metrics, the authors demonstrate that their approach yields a scalable and trustworthy labeling process.

3.9 Evaluation Metrics

In order to assess the quality and reliability of our auto-labeling framework, we employ standard classification metrics widely used in information retrieval and machine learning. These metrics allow us to quantify the performance of individual models as well as ensembles under various sampling and prompting strategies.

3.9.1 Precision

Precision measures the proportion of correctly predicted positive instances among all instances predicted as positive. It quantifies how many of the predicted labels are actually correct. Formally:

$$\text{Precision} = \frac{TP}{TP + FP}$$

where TP denotes true positives and FP denotes false positives. High precision indicates that the model makes few false positive errors.

3.9.2 Recall

Recall (also known as sensitivity) measures the proportion of correctly predicted positive instances out of all actual positive instances. It captures how well the model identifies all relevant cases. Formally:

$$\text{Recall} = \frac{TP}{TP + FN}$$

where FN denotes false negatives. High recall indicates that the model successfully retrieves most relevant items, though it may allow more false positives.

3.9.3 F1

The F1-score is the harmonic mean of precision and recall, balancing both measures into a single metric. It is particularly useful when the dataset is imbalanced, as it penalizes models that perform well on one metric but poorly on the other. Formally:

$$\text{F1} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

3.9.4 Confusion Matrix

The confusion matrix provides a complete breakdown of model predictions across all categories. It is a tabular representation where rows correspond to the actual labels and columns correspond to the predicted labels. Each cell (i, j) contains the number of instances with true label i but predicted as j. This visualization helps identify systematic misclassifications, class imbalance issues, and categories where the model struggles.

3.9.5 Relevance to This Research

These metrics are crucial for our study because they provide complementary perspectives on labeling performance. Precision ensures that the automatically assigned labels are reliable, while recall ensures that rare but critical categories (e.g., *Token Limit Exceeded*) are not overlooked. The F1-score balances the trade-off between the two, especially important in our imbalanced dataset. Finally, the confusion matrix offers interpretability, revealing which categories are commonly confused and guiding further improvements in retrieval, sampling, and ensemble strategies. Together, these metrics provide a robust evaluation framework for understanding the effectiveness of LLM-based auto-labeling pipelines.

4 Results and Discussion

4.1 GitHub Issue labeling Using individual LLM

Before moving to ensemble-based pipelines, we first evaluated the performance of individual LLMs in zero-shot labeling of GitHub issues. Each model was tasked with assigning issue categories without additional retrieval or ensemble support. Performance was assessed against the ground truth using accuracy, weighted F1-score, and Cohen's κ to capture inter-rater reliability.

Table 4.1: Individual LLM accuracies and agreement for GitHub issue labeling

Model	Accuracy (%)	Weighted F1
GPT-4.1	78.0	0.84
Gemini-2.0-flash	75.0	0.81
Claude-3.5-Sonnet	82.0	0.85

4.2 GitHub Issue labeling Using 2 LLMs and 1 Judge

We first evaluated GitHub issue labeling using an ensemble of two LLMs (GPT-4.1 and Claude-3.5) with another LLM as judge for conflict resolution. Each model independently labeled the set of GitHub issues, and any conflicting predictions were resolved by a designated tie-breaker model (Claude-3-Opus) acting as a judge.

Table 4.2: Individual LLM accuracies for GitHub issue labeling (2-model pipeline)

Model	Accuracy (%)
GPT-4.1	81.10
Claude-3.5	78.90
Ensemble (with judge)	81.29

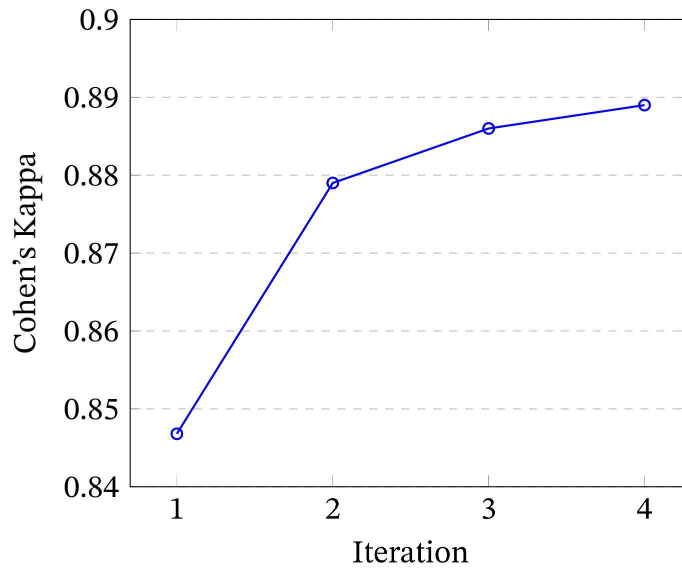

Figure 4.1: Cohen's Kappa across four labeling iterations for 2 LLMs

4.3 GitHub Issue labeling Using 2 LLMs and 1 Judge (tuning K-sampling)

In the ensemble we tried to incorporate a self-judging mechanism within the peer LLMs with K-sampling technique. We kept the ensemble combination intact like the previour 2 peer and 1 judge but here we implemented the peer to consider its top three inferences before providing the final output.

First we tried with top-3 k samples to understand its performance:

Table 4.3: Zero Shot 2 peers, 1 judge with k-sampling (top-3)

Model	Accuracy (%)
GPT-4.1	82.217
Claude-3.5-Sonnet	81.76
Ensemble (with judge)	83.37

We tried with top-7 k samples to understand how it differs:

Table 4.4: Zero Shot 2 peers, 1 judge with k-sampling (top-7)

Model	Accuracy (%)
Ensemble (with judge)	78.34

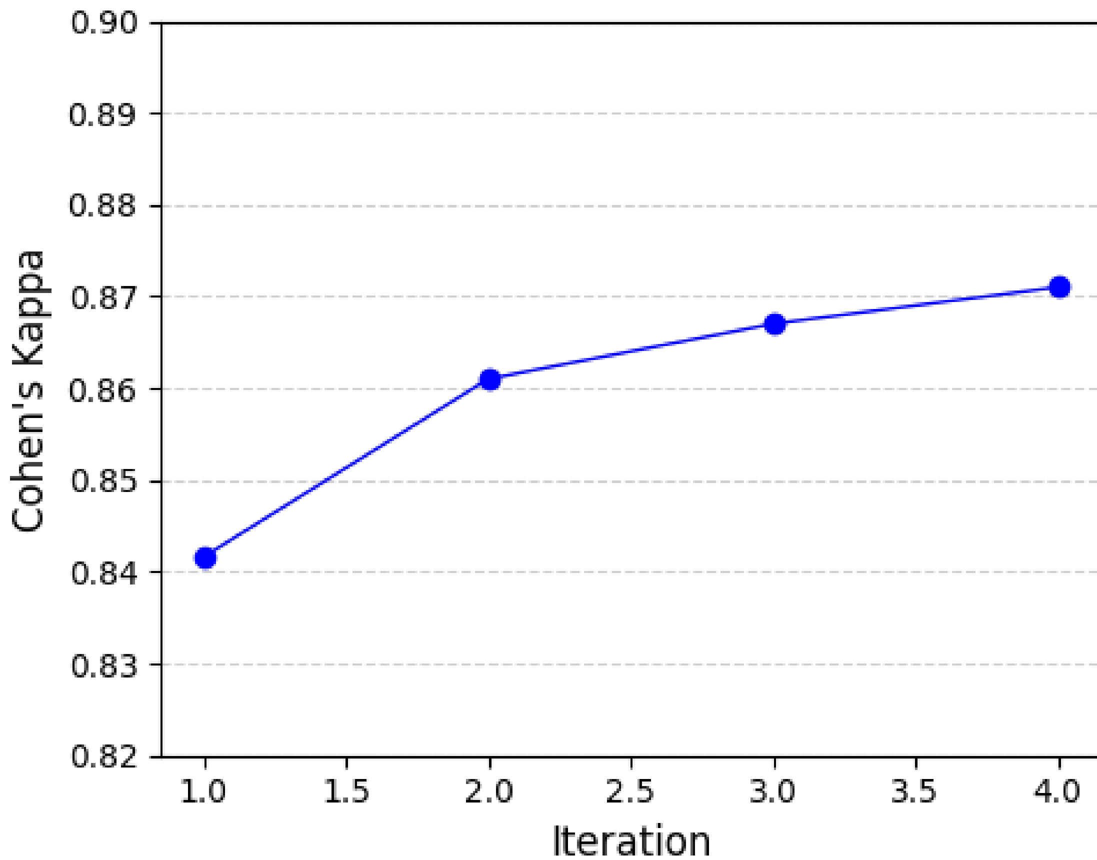

Figure 4.2: Cohen's Kappa across Iterations

4.4 GitHub Issue labeling Using 3 LLMs and 1 Judge

In the ensemble we incorporated a third LLM (GPT-3.5 Turbo). For labeling, we adopted a majority voting scheme after running a few iterations. Lastly, when the output of all three models differed, the designated judge model (Claude-3-Opus) settled the issue.

Table 4.5: Individual LLM accuracies for GitHub issue labeling (3-model pipeline)

Model	**Accuracy (%)**
GPT-4.1	81.10
Claude-3.5	78.90
GPT-3.5 Turbo	75.98
Ensemble (majority + judge)	81.99

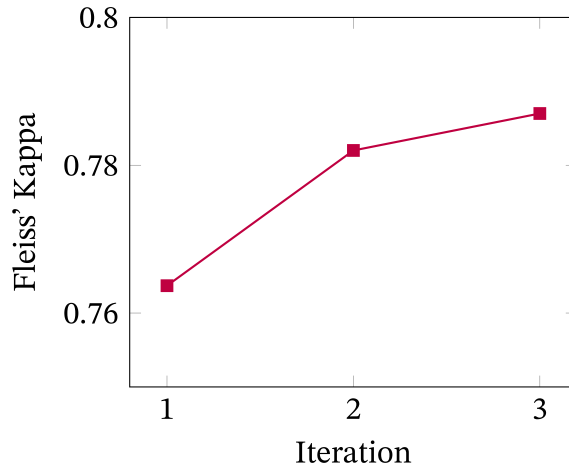

Figure 4.3: Fleiss' Kappa across three labeling iterations for 3 LLMs

4.5 GitHub Issue labeling Using 3 LLMs and 1 Judge (tuning K-sampling)

Table 4.6: Individual LLM accuracies for GitHub issue labeling (3-model pipeline)

Model	**Accuracy (%)**
GPT-4.1	81.10
Claude-3.5	78.90
GPT-3.5 Turbo	75.98
Ensemble (majority + judge)	81.99

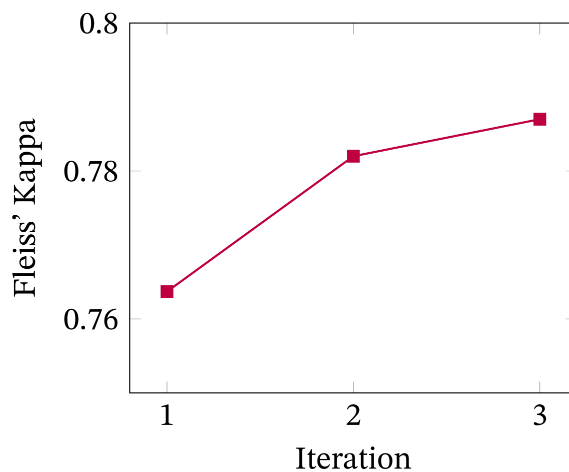

Figure 4.4: Fleiss' Kappa across three labeling iterations for 3 LLMs

4.6 LLM Issues labeling using 2 peers 1 judge with K-sampling on Prepared Dataset

Overall Accuracy: 57.01

Classification Report

Table 4.7: Classification Report on Cleaned Dataset (1547 samples)

Class	Precision	Recall	F1-score	Support
API Misuse	0.59	0.90	0.71	594
Authentication/API Key	0.71	0.77	0.74	61
CUDA/Device Error	0.71	0.35	0.47	48
Inference Failure	0.19	0.08	0.12	118
Installation Issue	0.76	0.71	0.73	154
Memory Management Issue	0.00	0.00	0.00	0
Memory/GPU Error	0.00	0.00	0.00	0
Model Architecture Issue	0.24	0.37	0.29	83
Model Loading Error	0.54	0.48	0.51	67
Other/Unknown	0.65	0.20	0.30	386
Token Limit Exceeded	0.58	0.78	0.67	36
Accuracy			0.57	1547
Macro avg	0.41	0.39	0.38	1547
Weighted avg	0.58	0.57	0.53	1547

Coefficient Scores

- Initial Cohen's Kappa Score for OpenAI vs. Claude: 0.4738 ' Kappa score is: 0.4738152055371392
- Cohen's Kappa Score for Iteration 1 (OpenAI vs. Claude): 0.7698
- Cohen's Kappa (Iteration 2 - OpenAI vs. Claude): 0.7844
- Cohen's Kappa (Iteration 3 - OpenAI vs. Claude): 0.7970
- Number of disagreements between the two models (Iteration 3): 208
- Percentage of final labels matching the original 'label' column: 51.97

Confusion Matrix

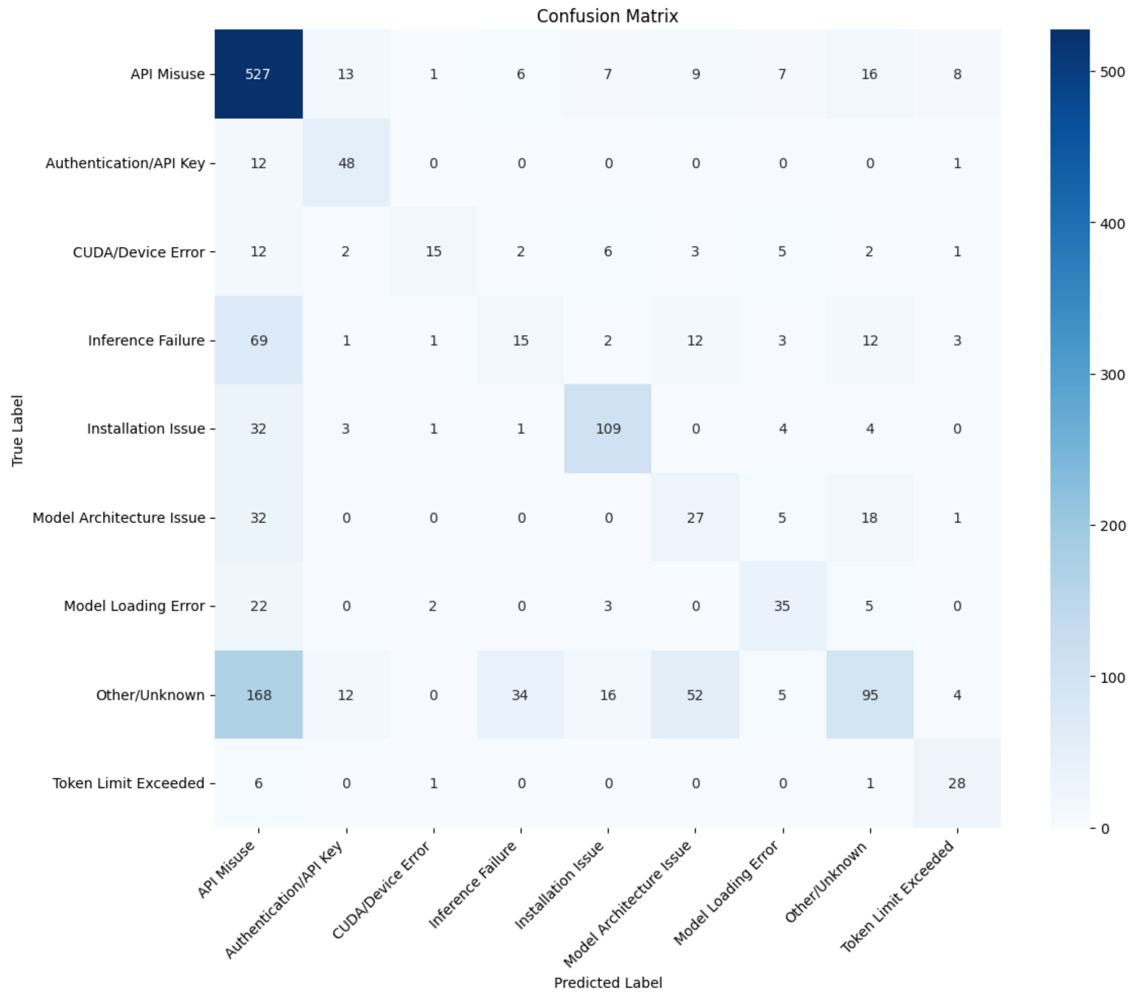

Figure 4.5: Confusion Matrix of 2 LLM and 1 Judge with K-sampling

4.7 Comparative Discussion

This section synthesizes results across all pipelines to explain when and why each design choice (zero-shot, RAG, k-sampling, LLM-as-judge, and peer count) improves or harms performance. We emphasize three axes: overall accuracy/F1, reliability (Cohen's/Fleiss' κ), and behavior on minority classes.

4.7.1 Overall Trends: Single Models vs. Ensembles

Individual zero-shot labeling demonstrated that modern LLMs can produce competitive baselines without task-specific finetuning. Among single models, Claude-3.5-Sonnet achieved the highest accuracy (82%), followed by GPT-4.1 (78%) and Gemini-2.0-flash (75%). However, accuracy alone masks instability: pairwise agreement is sensitive to prompt wording and decoding parameters. Introducing a judge yields small but consistent gains and, more importantly, stabilizes outcomes. The 2-peer-

1-judge pipeline reached 81.29% accuracy, closely tracking the stronger peer while filtering obvious mislabels from the weaker one.

4.7.2 Effect of the Judge: Convergence and Stability

The adjudication layer is most useful exactly where zero-shot models diverge: ambiguous phrasing, overlapping taxonomies (e.g., documentation vs. enhancement), and long issues with multiple root causes. In the two-peer pipeline, iterative temperature tuning coupled with adjudication raised Cohen's κ from 0.8468 to 0.889, indicating not only better accuracy but also stronger consistency beyond chance. In practice, the judge curbs idiosyncratic biases of either peer and reduces variance across re-runs, which is crucial for building a dependable labeled corpus.

4.7.3 Impact of k-Sampling: Finding the Sweet Spot

Diversity helps—up to a point. With top-3 k-sampling inside each peer, the 2-peer-1-judge ensemble improved to 83.37% accuracy. The within-peer majority consolidates stochastic samples, dampening random misfires while retaining beneficial exploration that rescues rare classes. In contrast, increasing to top-7 degraded ensemble accuracy to 78.34%. Beyond a moderate k, additional samples add low-utility variance that dilutes within-peer consensus, increases cross-peer dispersion, and burdens the judge with noisier candidates. The takeaway is a **moderate** k (e.g., 3) with low temperature (e.g., 0.3) as a robust default: it balances exploration (recall on sparse labels) and consolidation (precision on frequent labels).

4.7.4 Two vs. Three Peers: Marginal Accuracy, Higher Reliability

Adding a third peer (GPT-3.5-Turbo) produced an ensemble accuracy of 81.99% with majority+judge, roughly on par with the 2-peer setup. The primary benefit was not raw accuracy but improved group reliability: Fleiss' κ rose from 0.7637 to 0.787 after temperature iterations, indicating stronger multi-rater consistency. Three peers reduce the probability that a single model's systematic bias dominates the final label—useful in safety-critical or auditing contexts. That said, cost and latency increase; for throughput-oriented pipelines, **2 peers + judge + top-3 k-sampling** offered the best accuracy–efficiency trade-off in our runs.

4.7.5 Reliability Trajectories with Temperature Tuning

Across both 2-peer and 3-peer settings, gradually raising temperature for re-labeling only the disagreements improved agreement metrics (Cohen's and Fleiss' κ) without re-processing the entire dataset. This selective re-annotation concentrates compute where uncertainty is highest, encouraging peers to explore alternative but plausible labels and converge when the initial pass was brittle. The monotonic κ gains (e.g., 0.8468 $\rightarrow$ 0.8790 $\rightarrow$ 0.8860 $\rightarrow$ 0.8890) support an iterative, disagreement-focused schedule as a practical knob for reliability.

4.7.6 Error Analysis and Class Imbalance

Confusion matrices and per-class metrics on the prepared dataset (1,547 samples) show a long-tail pattern. Frequent classes such as API Misuse (F1 = 0.71) and Installation Issue (0.73) are learned reliably, while under-represented categories degrade (Inference Failure, F1 = 0.12; Model Architecture Issue, 0.29). Other/Unknown shows reasonable precision (0.65) but low recall (0.20), suggesting the ensemble remains conservative in assigning the catch-all label. Moderate k-sampling helps recall on sparse classes by letting peers surface minority hypotheses that would be pruned under greedy decoding, but excessive k reverses the benefit by proliferating off-topic candidates.

4.7.7 Prepared Dataset vs. GitHub Issues: Why Accuracy Differs

While GitHub issue ensembles reached ~81–83% accuracy, the prepared Stack Overflow dataset attained 57.01% accuracy in the 2-peer-1-judge with k-sampling setting. Two factors explain the gap. First, the SO taxonomy is finer-grained (Authentication/API Key, Token Limit Exceeded, CUDA/Device Error, etc.) and labels root causes rather than surface themes, raising difficulty. Second, class imbalance is more severe (e.g., 36 instances of Token Limit Exceeded vs. 594 API Misuse), increasing the penalty for false negatives on rare types. Notably, several rare but well-defined categories still showed competitive F1 (Token Limit Exceeded, 0.67; Authentication/API Key, 0.74), indicating that when the textual cues are explicit, zero-shot+ensemble can perform strongly even without finetuning.

4.7.8 On RAG and Few-Shot: Limited Gains in Heterogeneous Issues

We observed marginal or negative returns from RAG and few-shot prompting on GitHub issues. RAG occasionally retrieved irrelevant passages that diluted prompts and hurt precision; few-shot examples did not generalize because issue phrasing is highly heterogeneous and context-specific, so pattern-matching on a handful of exemplars was unreliable. In contrast, ensemble diversity plus adjudication consistently improved agreement and accuracy without relying on brittle exemplars or noisy retrieval.

4.7.9 Practical Guidance

- **Default setting:** 2 peers + judge with **top-3 k-sampling** at low temperature (e.g., 0.3) offers the best accuracy–efficiency trade-off in our results.
- **When to add a third peer:** choose 3 peers if you prioritize reliability and auditability (higher Fleiss' κ) over throughput or cost.
- **Iterative schedule:** re-label only disagreements while slightly increasing temperature across iterations to raise κ without reprocessing the entire corpus.
- **Rare classes:** use moderate k to boost recall on under-represented categories; avoid large k (e.g., 7) to prevent variance inflation and accuracy drop.
- **RAG/few-shot:** reserve for domains with stable, well-scoped schemas and high-quality corpora; for open-ended issue text, ensemble methods are more reliable.

4.7.10 Threats to Validity and Limitations

First, zero-shot judgments remain sensitive to prompt phrasing and decoding, though our peer+judge framework mitigates this. Second, agreement metrics can be inflated by frequent classes; we therefore report macro/weighted summaries and examine minority labels explicitly. Third, judge bias is model-dependent; swapping the judge or adding rubric-style instructions could further stabilize decisions. Finally, our results may vary with newer LLM releases or domain shifts; nonetheless, the relative findings—moderate k beats large k, judge improves stability, and 3 peers raise reliability—are likely to hold.

Summary

Ensembles with adjudication deliver consistent improvements over single LLMs. The most robust setting we observed is **2 peers + judge + top-3 k-sampling with iterative temperature for disagreements**, which combines high accuracy with good agreement and manageable cost. Adding a third peer yields higher reliability (Fleiss' κ) with modest accuracy changes. For heterogeneous, real-world issues, diversity via k-sampling and consensus via a judge outperform RAG.

5 Dataset Construction

5.1 Sources and Raw Ingest

We curated a corpus of developer issues/questions drawn from *GitHub Issues* and *Stack Overflow* that describe practical problems encountered when building or operating LLM systems. The initial raw ingest contained **1885** records with heterogeneous metadata (URLs, timestamps, owners, tags, vote counts, etc.). Each record comprised a short `title`, a free-form `description`, optionally the `top_3_answers` from community replies, and a canonical `label` indicating the failure mode.

5.2 Cleaning and Filtering

We performed schema normalization and removed entries with missing or unusable labels. Concretely, we dropped rows with empty values in the `label` field (**335** rows), resulting in a cleaned dataset of **1547** labeled entries:

$$1885_{\text{raw}} - 335_{\text{empty label}} = 1547_{\text{cleaned}}.$$

The resulting dataset was persisted as `cleaned_dataset.csv` and used for all subsequent experiments.

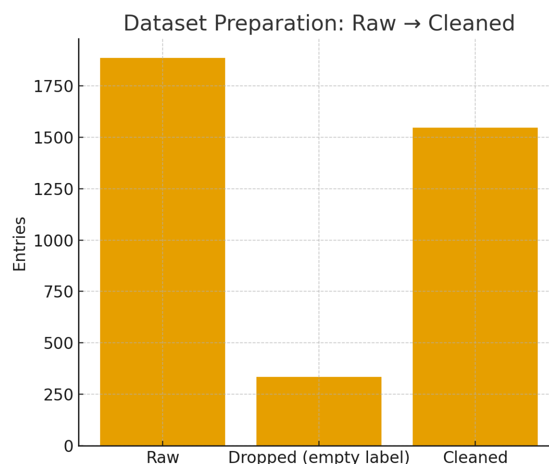

Figure 5.1: Post-cleaning Dataset

5.3 Final Schema

To reduce noise and focus on fields directly informative for classification, we retained the following columns:

- `question_id`: unique identifier of the issue/question.
- `title`: short summary of the problem.
- `description`: detailed problem statement (free text).
- `top_3_answers`: textual concatenation of the three most helpful answers/snippets.
- `label`: gold label.

5.4 Data Distributions

The distribution reveals that **API Misuse** (594 instances) and **Other/Unknown** (386 instances) are the most frequent categories, while categories such as **Token Limit Exceeded** (36 instances) and **CUDA/Device Error** (48 instances) are significantly underrepresented. This long-tailed distribution highlights the challenges of classification in imbalanced datasets and underscores the need for ensemble and augmentation strategies to improve recall for rare categories.

5.5 Discussion

This dataset is the first curated collection, to our knowledge, that specifically focuses on **LLM-related issues and failure modes**. By capturing diverse problem categories across GitHub and Stack Overflow, it provides a benchmark for evaluating auto-labeling pipelines. The imbalance across labels mirrors real-world usage scenarios, where some issues (e.g., API misuse) are far more common than rare failure cases (e.g., token limit exceeded). This imbalance directly motivates the use of retrieval-augmented generation, ensemble models, and reliability metrics to improve label quality.

6 Conclusion

This work investigated whether large language models (LLMs) can act as *universal, zero-shot labelers* for unstructured developer discussions and how ensemble design choices and decoding strategies shape reliability at scale. We contributed (i) a principled evaluation of zero-shot [4], RAG, k-sampling, and LLM-as-Judge pipelines on GitHub issues [13] and a newly curated Stack Overflow dataset of **1,547** labeled entries, and (ii) a practical ensemble blueprint that trades a small amount of additional compute for markedly higher stability and agreement.

Summary of Findings

- **Zero-shot baselines are strong but brittle.** Individual models achieved competitive accuracy on GitHub issues (e.g., Claude-3.5-Sonnet $\approx$ 82%, GPT-4.1 $\approx$ 78%), yet pairwise agreement was sensitive to prompt/decoding choices.
- **Judged ensembles stabilize outcomes.** A *2-peer-1-judge* pipeline lifted agreement from $\kappa = 0.8468$ to $\kappa = 0.889$ across iterations, producing consistent gains with modest overhead and filtering idiosyncratic errors from either peer [16].
- **Moderate k-sampling helps; excessive k hurts.** Top-3 k-sampling inside each peer improved ensemble accuracy to **83.37**%; pushing to top-7 increased variance and reduced accuracy to **78.34**%. The sweet spot balanced exploration (recall on sparse classes) with consolidation (precision on frequent classes).
- **Third peer improves reliability more than accuracy.** The *3-peer-1-judge* setting achieved accuracy on par with 2-peer variants (~82%), but raised Fleiss' κ (0.7637→0.787), which is valuable for auditability and safety-critical curation [17].
- **Domain difficulty matters.** On our fine-grained Stack Overflow dataset, the best judged ensemble reached **57.01**% accuracy overall. Long-tail labels and root-cause taxonomies are harder; still, well-signaled classes (e.g., *Authentication/API Key*, *Token Limit Exceeded*) achieved competitive F1, showing zero-shot ensembles can succeed when cues are explicit.
- **RAG and few-shot offered limited benefit here.** In highly heterogeneous, open-ended issues, retrieved context frequently added noise and few-shot exemplars failed to generalize; ensemble diversity plus adjudication proved more

reliable than retrieval or templated examples.

Practical Blueprint

Our results support a pragmatic default:

1. **Two peers + judge**, zero-shot prompts, re-label *only disagreements*.
2. **Top-3 k-sampling** at low temperature (e.g., $T \approx 0.3$) within each peer; majority *within* and *between* peers; escalate remaining ties to the judge.
3. **Iterative schedule:** slowly increase temperature across rounds for the disputed subset, monitoring Cohen's/Fleiss' κ and stopping when gains plateau [16], [17].

This configuration consistently delivered strong accuracy with higher chance-corrected agreement and manageable cost/latency. Add a third peer when *reliability and auditability* outweigh throughput constraints.

Limitations

- **Prompt and decoding sensitivity.** Although ensembles mitigate variance, outputs still depend on instructions and sampling settings; different judges may encode different biases.
- **Class imbalance and sparse regimes.** Minority classes remain challenging; even with k-sampling, extreme sparsity limits recall.
- **External validity.** Results may shift with newer model releases, policy updates, or domain drift; conclusions emphasize *relative* trade-offs rather than absolute scores.

Bibliography

[1] G. Aracena, K. Luster, F. Santos, I. Steinmacher, and M. A. Gerosa, "Applying large language models api to issue classification problem," in *Proceedings of the 46th International Conference on Software Engineering (ICSE)*, Lisbon, Portugal: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2401.04637

[2] J. Bai et al., *Qwen technical report*, 2023. arXiv: 2309.16609 [cs.CL]. [Online]. Available: https://doi.org/10.48550/arXiv.2309.16609

[3] R. H. Bugzilla, "The red hat bug tracking system," in *Bug Report*, Red Hat Bugzilla, 2025. [Online]. Available: https://bugs.launchpad.net/bugs/bugtrackers/redhat-bugs

[4] M. Cate, "Understanding zero-shot and few-shot learning in llms," Jun. 2023.

[5] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and Psychological Measurement*, vol. 20, no. 1, pp. 37–46, 1960. DOI: 10.1177/001316446002000104 eprint: https://doi.org/10.1177/001316446002000104. [Online]. Available: https://doi.org/10.1177/001316446002000104

[6] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, *Bert: Pre-training of deep bidirectional transformers for language understanding*, 2019. arXiv: 1810.04805 [cs.CL]. [Online]. Available: https://doi.org/10.48550/arXiv.1810.04805

[7] X. Du, Z. Liu, C. Li, X. Ma, Y. Li, and X. Wang, "Llm-brc: A large language model-based bug report classification framework," *Software Quality Journal*, vol. 32, no. 3, pp. 985–1005, 2024. DOI: https://doi.org/10.1007/s11219-024-09675-3

[8] Z.-g. Fang, S.-q. Yang, C.-x. Lv, S.-y. An, and W. Wu, "Application of a data-driven xgboost model for the prediction of covid-19 in the usa: A time-series study," *BMJ Open*, vol. 12, no. 7, e056685, 2022. DOI: 10.1136/bmjopen-2021-056685 [Online]. Available: https://bmjopen.bmj.com/content/12/7/e056685

[9] A. A. Hasan, S. Saha, M. M. Imran, and T. S. Zaman, "Llput: Investigating large language models for bug report-based input generation," in *Companion Proceedings of the 33rd ACM Symposium on the Foundations of Software Engineering (FSE '25)*, Trondheim, Norway: Association for Computing Machinery, 2025. [Online]. Available: https://doi.org/10.48550/arXiv.2503.20578

[10] B. Hui, J. Yang, Z. Cui, and J. Yang, *Qwen 2.5 coder*, 2024. arXiv: 2409.12186 [cs.CL]. [Online]. Available: https://doi.org/10.48550/arXiv.2409.12186

[11] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan, "A comprehensive study on deep learning bug characteristics," in *Pro-ceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19)*, New York, NY, USA: ACM, 2019. [Online]. Available: https://doi.org/10.48550/arXiv.1906.01388

[12] A. R. D. Kelin, B. Nagarajan, S. Rajendran, and S. Muthumari, "Automatic bug classification system to improve the software organization product performance," *International Journal of Sociotechnology and Knowledge Development*, vol. 14, no. 1, 2022. DOI: https://doi.org//IJSKD.310066

[13] E. Lewtun, "Github issues dataset," 2022. [Online]. Available: https://huggingface.co/datasets/lewtun/github-issues

[14] A. F. Otoom, S. Al-jdaeh, and M. Hammad, "Automated classification of software bug reports," in *Proceedings of the 18th International Conference on Software Engineering Research and Practice (SERP'19)*, Las Vegas, USA: The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2019. DOI: 10.1145/3357419.3357424 [Online]. Available: https://doi.org/10.1145/3357419.3357424

[15] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, “Bleu: A method for automatic evaluation of machine translation,” in *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, ACL, 2002, pp. 311–318.

[16] G. Rau and Y.-S. Shih, “Evaluation of cohen’s kappa and other measures of inter-rater agreement for genre analysis and other nominal data,” *Journal of English for Academic Purposes*, vol. 53, p. 101 026, 2021, ISSN: 1475-1585. DOI: https://doi.org/10.1016/j.jeap.2021.101026 [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1475158521000709

[17] statsmodels, *Fleiss kappa*, 2025. [Online]. Available: https://www.statsmodels.org/dev/generated/statsmodels.stats.inter_rater.fleiss_kappa.html

[18] E. H. Yılmaz, C. E. Öztürk, and Ö. Köksal, “Bug report classification with ensemble learning for closed-source software,” *Preprint*, 2023, Available on arXiv and SSRN. [Online]. Available: https://doi.org/10.22541/au.169401793.32858063/v1

Appendices

A Prompts used for Evaluations

A.1 Stackoverflow prompt for each iteration:

Prompt: LLM-Bug-Tagger

ROLE & GOAL ### You are an expert AI assistant, ‘LLM-Bug-Tagger‘, designed for Stack Overflow. Your task is to analyze user posts and classify them into one of the following categories.
CATEGORIES

- API Misuse
- Authentication/API Key
- CUDA/Device Error
- Inference Failure
- Installation Issue
- Model Architecture Issue
- Model Loading Error
- Other/Unknown
- Token Limit Exceeded

INSTRUCTIONS
1. **Analyze all context**: Examine the ‘Title‘, ‘Description‘, and the provided ‘Answers‘. 2. **Prioritize the solution**: The answers are the strongest signal for the problem’s root cause. 3. **Determine the root cause**: Based on all the information, select the single best category. 4. **Strict output**: Your response MUST BE **ONLY** the category label. Do not add any explanation, punctuation, or other text.
POST TO CLASSIFY
Title: {row[’title’]}
Description: {row[’description’]}
Answers: {row[’top_3_answers’]}
YOUR CLASSIFICATION

A.2 Stackoverflow initial prompt for each peer:

Prompt: LLM-Bug-Tagger

ROLE & GOAL ### You are an expert AI assistant, ‘LLM-Bug-Tagger‘, designed for Stack Overflow. Your task is to analyze user posts and classify them into one of the following categories.
CATEGORIES

- API Misuse
- Authentication/API Key
- CUDA/Device Error
- Inference Failure
- Installation Issue
- Model Architecture Issue
- Model Loading Error
- Other/Unknown
- Token Limit Exceeded

INSTRUCTIONS
1. **Analyze all context**: Examine the ‘Title‘, ‘Description‘, and the provided ‘Answers‘. 2. **Prioritize the solution**: The answers are the strongest signal for the problem’s root cause. 3. **Determine the root cause**: Based on all the information, select the single best category. 4. **Strict output**: Your response MUST BE **ONLY** the category label. Do not add any explanation, punctuation, or other text.
POST TO CLASSIFY
Title: {row['title']}
Description: {row['description']}
Answers: {row['top_3_answers']}
YOUR CLASSIFICATION

A.3 Stackoverflow Judge prompt:

Prompt: LLM-Bug-Tagger

```
### ROLE & GOAL ### You are an expert AI assistant, ‘LLM-Bug-Tagger‘,
designed for Stack Overflow. Your task is to analyze user posts
and classify them. Two advanced AI models have each classified the
following Stackoverflow posts into different labels. Your role is
to carefully read the issue and these labels, and then choose the
**single most appropriate label** from the initial labels provided.
### INSTRUCTIONS ###
1. **Analyze all context**: Examine the ‘Title‘, ‘Description‘,
and the provided ‘Answers‘. 2. **Consider initial labels**: Take
into account the initial labels provided by the other two AI models.
3. **Determine the root cause**: Based on all the information,
select the single best category *from the provided initial labels*.
4. **Strict output**: Your response MUST BE **ONLY** the category
label. Do not add any explanation, punctuation, or other text.
### POST TO CLASSIFY ###
**Title:** {row['title']}
**Description:** {row['description']}
**Answers:** {row['top_3_answers']}
### INITIAL LABELS (Iteration 3) ###
Peer-1 Label: {row['openai_predicted_labels_it3']}
Peer-2 Label: {row['claude_predicted_labels_it3']}
### YOUR FINAL CLASSIFICATION (Choose from INITIAL LABELS) ###
```

A.4 Github Initial prompt for each peer:

Prompt: LLM-Bug-Tagger

```
You're a issue-type classifier for Github issues. Classify the
following issue into one of these categories:

  • Dataset discussion

  • Metric discussion

  • bug

  • dataset request
```

- documentation
- enhancement
- generic discussion
- good first issue
- metric bug
- metric request
- nlp-viewer
- question
- refactoring
- wontfix

Post Content: {text}
Respond with just the label.

A.5 Github Iteration prompt:

Prompt: LLM-Bug-Tagger

ROLE & GOAL ### You are an expert AI assistant, a issue-type classifier for Github issues. Your task is to analyze github issues and classify them. You are also be provided with the initial labels from three advanced AI models, and you should choose the most appropriate single label based on all the information.
INSTRUCTIONS
1. **Analyze the post**: Carefully read the issue text to understand the root cause of the issue. 2. **Review the conflicting labels**: Examine the three ‘INITIAL LABELS‘ provided by your peers. 3. **Select the best fit**: Choose the single label from the ‘INITIAL LABELS‘ that most accurately represents the root cause you identified. 4. **Strict output**: Your response MUST BE **ONLY** the selected category label. Do not add any explanation or other text.
Post Content: {text}
INITIAL LABELS

```
peer-1 Label: {peer1_label}
peer-2 Label: {peer2_label}
peer-3 Label: {peer3_label}
```

A.6 Github Judge Prompt:

Prompt: LLM-Bug-Tagger

```
Two expert LLMs classified the following GitHub issue differently.
--
Post: {text}
--
First LLM's assigned label: {gpt_label}
Second LLM's assigned label: {claude_label}
Classify the issue into one of the mentioned labels.
Respond with just the label. No explaination or anything, a single
word response with the label.
```