BACHELOR OF SCIENCE IN SOFTWARE ENGINEERING

Bug Report Summarization with Large Language Models

Shaira Sadia Karim
200042122
Lamia Alam
200042162
Abrar Mahmud Rahim
200042168

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

Bug Report Summarization with Large Language Models

Shaira Sadia Karim
200042122
Lamia Alam
200042162
Abrar Mahmud Rahim
200042168

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 **Shaira Sadia Karim**, **Lamia Alam**, and **Abrar Mahmud Rahim** under the supervision of **Dr. Abu Raihan Mostofa Kamal**, Professor, Department of Computer Science and Engineering and co-supervision of **Dr. Md. Azam Hossain**, Associate Professor, Department of Computer Science and Engineering, **Lutfun Nahar Lota**, Assistant Professor, Department of Computer Science and Engineering, and **Ishmam Tashdeed**, Lecturer, Department of Computer Science and Engineering, 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 Information derived from the published and unpublished work of others have been acknowledged in the text and a list of references is given.

Dr. Abu Raihan Mostofa Kamal
Professor
Department of Computer Science and Engineering
Islamic University of Technology (IUT)
Date: September 30, 2025

Dr. Md. Azam Hossain
Associate Professor
Department of Computer Science and Engineering
Islamic University of Technology (IUT)
Date: September 30, 2025

Lutfun Nahar Lota
Assistant Professor
Department of Computer Science and Engineering
Islamic University of Technology (IUT)
Date: September 30, 2025

Ishmam Tashdeed
Lecturer
Department of Computer Science and Engineering
Islamic University of Technology (IUT)
Date: September 30, 2025

Shaira Sadia Karim
Student ID: 200042122
Date: September 30, 2025

Lamia Alam
Student ID: 200042162
Date: September 30, 2025

Abrar Mahmud Rahim
Student ID: 200042168
Date: September 30, 2025

Dedicated to our supervisors, whose constant support, insightful feedback, and encouragement have made this work possible. Their guidance has profoundly shaped the direction and success of this research.

Contents

List of Figures

List of Tables

List of Abbreviations

LLM	Large Language Model
NLP	Natural Language Processing
SDLC	Software Development Life Cycle
LoRA	Low-Rank Adaptation
RTA	Represent Them All
GPT	Generative Pre-trained Transformer
TSM	Two-Layer Semantic Model
CA	Crowd-Attribute
PRST	PageRank-based Summarization Technique
LOC	Lines of Code
PEFT	Parameter-Efficient Fine-Tuning
FP16	16-bit Floating Point Precision
CUDA	Compute Unified Device Architecture
ROUGE	Recall-Oriented Understudy for Gisting Evaluation
BERT	Bidirectional Encoder Representations from Transformers
SSBRSSS	Summarization of Software Bug Report based on Sentence Semantic Similarity
SDS	Summary Dataset
ADS	Authorship Dataset
DDS	Duplicate Dataset
BRC	Bug Report Corpus
APBRC	Apache Projects Bug Report Corpus
KSCLP	Knowledge-Specific and Contrastive Learning Pre-Training
TF-IDF	Term Frequency-Inverse Document Frequency

Acknowledgement

We would like to express our heartfelt gratitude to our supervisor, Dr. Abu Raihan Mostofa Kamal, Professor, Department of Computer Science and Engineering, and to Dr. Md. Azam Hossain, Associate Professor, Department of Computer Science and Engineering. Their guidance, advice, and feedback have been invaluable throughout this work, and we are truly grateful for their constant support.

We also express our gratitude to our co-supervisor, Lutfun Nahar Lota, Assistant Professor, Department of Computer Science and Engineering, for her encouragement and helpful suggestions. Her guidance helped us make important decisions during challenging times.

We are deeply indebted to our co-supervisor, Ishmam Tashdeed, Lecturer, Department of Computer Science and Engineering, who supported us at every stage of the research. He provided us with his valuable insights and practical knowledge to shape our research in the current form.

We would also like to express our sincere appreciation to all the teachers at the Network and Data Analysis Group (NDAG) for their support and knowledge sharing throughout our research.

Abstract

The unstructured and verbose nature of bug reports often impedes developers from quickly comprehending the context of a problem and fixing underlying issues. While bug report summarization can facilitate faster comprehension, existing methods often rely on surface-level textual cues, leading to broken or disorganized summaries and failing to capture deeper semantic nuances. Moreover, these methods often neglect supporting code samples, which are critical for accurately detecting and understanding software issues. In this work, we propose Chunk-and-Fuse, a novel progressive code integration framework for LLM-based abstractive bug report summarization. Chunk-and-Fuse addresses the challenge of lengthy bug-related code snippets that exceed typical large language model (LLM) context windows by incrementally integrating segmented code and textual content. We evaluate our approach on four benchmark datasets across eight LLMs, achieving 7.5%–58.2% improvements over extractive baselines and performance comparable to leading abstractive techniques. Our findings demonstrate that jointly leveraging textual and code information can improve bug comprehension and accelerate software maintenance workflows.

Chapter 1

Introduction

Bugs are an inevitable part of software development and maintenance [17], [93]. They are unintended errors in a software program that lead to incorrect and unexpected behavior [36]. These bugs can arise from logic errors, incorrect assumptions, overlooked edge cases or wrong requirements [20], [22], [54], [87]. The process of addressing bugs begins with the creation of a bug report, which is a formal record submitted by users, testers, or developers that describes an issue encountered during software usage [46]. Based on this report, developers analyze the problem and apply changes to fix the issue [75].

A typical bug report includes several components, such as a summary or title, a detailed description of the issue, steps to reproduce the bug, expected versus actual behavior, environment details (e.g., operating system, browser version), severity level, stack traces, partial code snippets, comments from users or developers, and, in many cases, attached logs or screenshots [19], [42], [59]. A sample bug report is illustrated in Figure 1.1.

In collaborative environments like open-source projects or large software firms, these reports are usually managed through issue tracking systems like JIRA, Bugzilla, or GitHub Issues [14], [71], [90]. Bug repositories such as Bugzilla are widely adopted and have become essential tools for managing development tasks. For instance, Bugzilla has been publicly adopted by over 134 organizations and projects, including Mozilla, Eclipse, Gnome, and GCC [38]. These repositories act as a centralized knowledge base, storing information about defects, investigations, and resolutions for future reference [7], [79].

Understanding the life cycle of a bug report is essential in maintaining software systems [23]. The life cycle usually begins with the creation of the bug report, which

Title: Application crashes when saving a new user in the system

Description: When I try to add a new user to the system by filling out the form and clicking the "Save" button, the application crashes without any error message. This happens consistently every time I attempt to save a new user.

Steps to Reproduce:

1. Go to the "User Management" section in the admin panel.
2. Click on "Add New User."
3. Fill out all required fields (name, email, etc.).
4. Click "Save."

Expected Result:: The new user should be saved and displayed in the list of users.

Actual Result: The application crashes, and no new user is added.

Environment:

- Browser: Chrome 95
- OS: Windows 10
- Version: v2.3.1

Error Logs: Error: Uncaught TypeError: Cannot read property 'save' of null at UserController.js:45

Code Snippet (Buggy):

```
function saveUser(user) {
  if (user != null) {
    user.save();  // Line 45
  }
}
```

Figure 1.1: An example bug report

must be carefully read and understood by developers and testers. This is followed by triaging, where it is categorized, prioritized, and assigned [8]. Developers then analyze the issue, possibly request more information, and finally work on a fix [91]. Once resolved, the fix is tested and verified by the tester, and the report is eventually closed. As such, bug reports serve as a primary medium for communication between developers and testers. Figure 1.2 shows the sequential life cycle of a bug report from creation, triaging, analysis, fixing, testing, verification, to closure.

Over time, these reports accumulate valuable historical data, enabling knowledge transfer, process improvement, and duplicate issue prevention [33], [48]. For example, earlier studies show that 200 Mozilla bug reports contained 275 references to other bug reports, demonstrating the extent to which developers rely on historical reports to resolve new issues [62]. In industrial projects, archived bug reports are indispensable since many reported issues do not require code changes but stem from configuration

Figure 1.2: A linear workflow depicting the life cycle of a bug report from creation to closure.

or usage problems [63]. Quick access to similar past cases helps developers avoid redundant work [18] and accelerates issue resolution.

As software projects grow, so does the number of bug reports they generate [73]. In many cases, these reports are long, inconsistently written, and filled with redundant or even unrelated information [12], [62], [98]. Developers and testers often have to go through hundreds or even thousands of reports to find the information they need [32], [38], [74]. This process can be repetitive, tiring, and time-consuming [63], especially when a single bug goes through several revisions or involves multiple contributors and unrelated discussions [9]. Since not all bug reports are written clearly, understanding the actual issue takes effort. Past studies indicate that comprehending historical bug reports is particularly challenging because of the sheer amount of knowledge involved in each report and the need to cross-reference multiple historical bugs for context [62]. Sometimes, the searching process incurs even more human-hours to filter out redundant information or duplicates [86], [95]. Although bug report titles provide a high-level summary [12], they are often insufficient for understanding the actual issue [38]. In such situations, having a concise and informative summary of the report can make a big difference by aiding comprehension [75]. It helps developers quickly grasp the problem and decide what to do next, ultimately saving time and improving productivity.

Text summarization is the process of creating a condensed version of a longer body of text while preserving its essential meaning [83]. It helps readers quickly grasp the main ideas with less effort and in less time. In the context of software engineering, bug report summarization aims to produce a brief yet informative summary that captures the core issue and key details of a bug report [59]. This helps developers quickly understand the nature, context, and main problem of a bug without having to read through long and detailed reports. As an example, Figure 1.3 illustrates the summarized version of the bug report presented earlier in Figure 1.1. While automatic text summarization has been successfully applied in domains such as documents [96], [102], news [67], [103], and emails [72], [88], bug report summarization presents unique challenges due to technical jargon, inconsistent structure, and interlinked his-

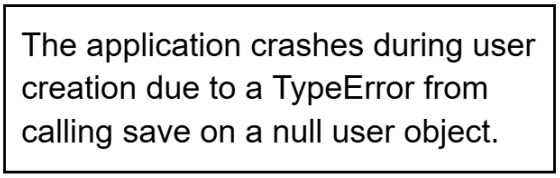


Figure 1.3: An example abstractive bug report summary

torical references, motivating the need for specialized approaches.

There are two major types of summarization approaches [31]:

- **Extractive summarization:** selects and compiles key sentences or phrases directly from the original text, as illustrated in Figure 1.4.
- **Abstractive summarization:** generates new sentences to paraphrase and condense the content while preserving its original meaning [51], resulting in more natural and human-like summaries, as illustrated in Figure 1.3.

Although straightforward, extractive methods often lead to summaries that are overly verbose, fragmented, and sometimes incoherent [26]. They also tend to suffer from redundancy and fail to resolve pronouns or maintain logical flow, limiting their usefulness in practical settings. In contrast, abstractive summarization has gained increasing attention due to its ability to understand context, semantics, and conversation flow [21], [53]. However, despite these advantages, abstractive approaches remain challenging to implement due to the complexity of accurately capturing both content and context [25], [80]. They require a deeper level of language understanding and generation, making them more difficult to design and evaluate compared to extractive techniques.

Natural Language Processing (NLP) techniques have increasingly been applied to automate various bug report tasks such as summarization, categorization, and localization [11], [75], [100]. Earlier NLP-based approaches made use of machine learning models, feature engineering, and rule-based systems [68]. Although these techniques improved structure and relevance to some extent, they often struggled with under-

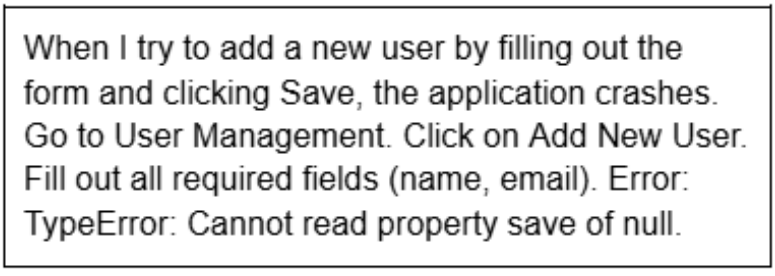


Figure 1.4: A example extractive bug report summary

standing deeper semantic relationships within the text.

With the emergence of Large Language Models (LLMs), the quality of text summarization has advanced considerably [3], [97]. These models demonstrate strong capabilities in understanding context, generating coherent natural language, and capturing subtle semantic nuances that traditional NLP methods often overlook. Their ability to produce fluent, context-aware, and human-like summaries makes them a compelling solution for handling the complexity and volume of modern bug reports. As a result, LLM-based abstractive summarization holds significant promise for enhancing software maintenance workflows and reducing the time developers spend interpreting lengthy and inconsistent bug reports.

1.1 Motivations and Scope

With the increasing scale and complexity of software systems, developers are often overwhelmed by the volume and inconsistency of bug reports. While this challenge has led to active research in bug report summarization, much of the existing work remains focused on extractive methods [47]. These methods, although simple and computationally efficient, tend to produce summaries that lack coherence, omit context, and often fail to meet the practical needs of developers [26]. Abstractive summarization, in contrast, offers a more flexible and semantically rich alternative [53]. However, it has seen limited adoption in the context of bug reports, likely due to its complexity and the difficulty of preserving both accuracy and clarity in generated summaries [25], [80].

One particularly underexplored yet critical aspect of bug report summarization is the role of code snippets [68]. Many bug reports include partial code examples or references to functions, classes, or variables, which often contain vital clues that the textual description alone may not explicitly convey. Despite this, most existing summarization techniques focus solely on the textual components of bug reports. Even when some methods consider embedded code fragments, they typically treat these snippets in isolation, failing to capture the deeper relationships between the descriptive text and the surrounding source code [56]. Consequently, summaries generated by traditional methods often miss important technical details, limiting their practical usefulness for debugging tasks [13]. Motivated by these challenges, this research seeks to enhance bug report summarization by integrating both the textual content and the relevant code snippets, enabling more context-aware and informative summaries.

At the same time, we leverage the powerful language understanding capabilities of

Large Language Models (LLMs) to perform more nuanced and context-aware summarization. These models are trained not only on natural language but also on code-related content, making them particularly well-suited for software engineering tasks, and thus highly promising for bug report summarization, where understanding both textual and code context is crucial. Although a few recent studies have explored LLM-based bug report summarization, these efforts are still limited and do not incorporate full code snippets, which restricts their ability to capture complete technical context [24], [81], [92].

In addition, prompt engineering has emerged as a promising technique for steering LLM behavior toward desired outputs. Prior work has shown that carefully designed prompts can significantly improve the performance of LLMs across a range of tasks, including reasoning, summarization, and question answering [16], [41], [43], [89], [101]. However, in particular, while prompt engineering has been successfully applied to general text summarization [76] and even to bug reports alone [60], no existing work has leveraged it to explicitly guide models in combining textual bug reports with their associated full code snippets for bug report summarization. Such integration can help models focus on relevant technical details and generate clearer, more accurate outputs.

Our thesis explores the application of LLMs and prompt engineering to generate concise abstractive summaries that incorporate both textual and code-level information from bug reports. Through this approach, we aim to enhance the quality and usefulness of bug report summaries for developers.

1.2 Problem Statement

Effective understanding of bug reports is critical for timely software maintenance and debugging [104]. However, bug reports often vary significantly in quality and structure, making it difficult for developers to quickly grasp the core issues [12]. Most existing bug report summarization techniques rely on extractive methods that merely select parts of the original text [47]. These summaries tend to be fragmented, lack coherence, and frequently overlook essential contextual information that provides crucial insights into the bug [26].

Current summarization approaches largely ignore or inadequately incorporate code snippets, which are crucial for fully understanding the nature and root causes of bugs [68]. Without integrating this code context, summaries fail to provide the depth necessary for effective debugging [13], [65]. Additionally, abstractive summarization,

which can generate novel, flexible summaries while preserving meaning, remains underutilized in this domain, limiting the potential for producing more natural and insightful summaries [25], [80].

The objectives of our research are:

- Develop an approach that effectively incorporates both bug report text and relevant code snippets for summarization.
- Explore the use of abstractive summarization techniques to generate coherent and meaningful summaries beyond simple text extraction.
- Utilize prompt engineering to guide LLMs in producing accurate, concise, and contextually rich summaries.
- Evaluate the proposed method against existing extractive and abstractive summarization approaches to demonstrate improvements in summary quality for developers.

1.3 Research Challenges

Our research addresses several key challenges in the domain of bug report summarization:

- One major challenge is the lack of high-quality datasets that include both bug reports and their corresponding source code [70]. This absence limits the development and evaluation of models capable of jointly understanding natural language descriptions and the related code context.
- Effectively understanding and integrating two fundamentally different modalities, namely natural language descriptions and code snippets, is a significant challenge in bug report summarization. Existing approaches often fail to capture the interaction between these components, and our research seeks to address this gap.
- Abstractive summarization introduces additional complexity. Unlike extractive methods that select portions of the original text, abstractive approaches must generate coherent, concise, and semantically faithful sentences. This requires a deeper understanding of both language and code semantics. We chose this challenging path for our research [49].
- Another challenge lies in designing prompts that can effectively guide LLMs to

generate summaries with the necessary information developers need to understand the bug. This involves a continuous, iterative refinement process to strike the right balance between specificity and generalization.

- The use of large models introduces practical constraints, as they are resource-intensive and require significant computational power for both inference and experimentation, especially when processing combined textual and code inputs.

1.4 Contribution

Our key contributions in this thesis are as follows:

- To the best of our knowledge, we are the first to incorporate full code snippets alongside textual bug reports to enhance the contextual understanding and quality of bug report summaries.
- We introduce the application of prompt engineering to integrate both bug reports and code snippets for abstractive summarization, extending beyond prior work that considered only bug report text.
- We demonstrate significant improvements in summary quality, supported by quantitative evaluation metrics such as BERTScore [99], which measures the semantic similarity between generated and ground truth summaries, highlighting the effectiveness of our approach compared to existing methods.

These contributions advance the current state of research by addressing key limitations in bug report summarization, particularly the lack of integration between textual and code information, and by leveraging recent advances in LLMs through prompt engineering. Our work provides a foundation for more context-aware and developer-friendly summaries, potentially reducing debugging time and improving software maintenance workflows.

1.5 Organization

This report provides a comprehensive overview of our research. We begin with an introduction in chapter 1, which outlines the research context and objectives. Following this, chapter 2 presents a detailed Literature Review, critically examining existing studies and methodologies relevant to our topic. In chapter 3, we describe our Proposed Methodology, including the integration of code and bug reports and the use of prompt engineering. Chapter 4 details the dataset and experimental setup used in our

study, as well as a detailed discussion of our results and analysis. Chapter 5 concludes the paper.

Chapter 2

Related Works

This chapter focuses on a comprehensive analysis of the most relevant research surrounding the domain of bug reports. The aim is to position this thesis within the broader context of existing works, identifying significant trends, emerging methodologies, and gaps in the literature. Rather than presenting a list of individual studies, the chapter adopts a narrative approach that highlights key areas of progress and challenges, ultimately setting the stage for the contributions of this research.

During our exploration of literature related to the Software Development Life Cycle (SDLC) and Natural Language Processing (NLP), we identified an opportunity to extend research in the area of bug report summarization. A review of existing studies revealed that current approaches often exclude code snippets from bug reports and place limited emphasis on abstractive summarization techniques using large language models (LLMs). Based on these gaps, we decided to expand our work in this area.

In addition to summarization, bug reports have also been studied from multiple perspectives, including tasks such as localization, test case generation, bug detection, and report quality improvement [15], [45], [55], [104]. These tasks play an important role in supporting software maintenance and reducing the overall cost of the SDLC. Among them, bug report summarization has become important as it directly addresses the challenge of extracting concise, relevant, and actionable information from lengthy and often noisy reports. Therefore, while this chapter briefly reviews research on related bug report tasks, its primary emphasis is on the evolution of bug report summarization techniques.

2.1 Literature Reviews, Research Trends in Bug Report

Bug reports are a crucial resource in software maintenance and have been studied over the years for various tasks to help developers identify, understand, and resolve issues efficiently [104]. These studies often rely on a variety of datasets, ranging from well-established benchmarks to specialized collections focused on specific projects, domains, or defect types. Table 2.1 provides an overview of widely used bug report datasets in the literature, summarizing their size, purpose, and key components, which range from textual descriptions to metadata.

Table 2.1: Summary of Bug Report Datasets used in Literatures

Dataset	**Count**	**Content**
SDS (Summary Dataset) [39]	36	Bug reports with human-annotated summaries across four open-source projects: Mozilla, Eclipse, KDE, and Gnome.
ADS (Authorship Dataset) [40]	96	Bug reports with human-annotated summaries across four open-source projects: Mozilla, Eclipse, KDE, and Gnome.
DDS (Duplicate Dataset) [66]	35	Duplicate bug reports and duplicate link.
BRC (Bug Report Corpus) [75]	36	General bug reports across multiple open-source projects.
BRS_BL (Tomcat, SWT, Eclipse, JDT) [100]	800	Bug reports for bug localization experiments.
APBRC (Apache Projects Bug Report Corpus) [42]	21	Apache project bug reports.
DB2-Bind Corpus	19	Bug reports from DB2 software.
Signal Repository Subset	30	Subset of signal repository used for bug report evaluation.
BugZilla	275, 639	Bug reports across multiple open-source projects: Mozilla, Eclipse, Netbeans, and GCC.
OSCAR [32]	59	Open-source software bug reports.

Building on these datasets, researchers have proposed diverse approaches to analyze, process, and extract information from bug reports. This section provides a review of existing studies, highlighting the methods and techniques used to support different bug report tasks, with particular attention to approaches for summarization.

2.1.1 LLM Approaches

These approaches leverage large pre-trained language models (LLMs) to process and analyze bug reports. These models can extract relevant information, highlight key patterns, and generate concise, actionable outputs for software maintenance tasks.

The emergence of large pre-trained language models (LLMs) has opened new possibilities for automating software maintenance tasks such as bug report summarization, evaluation, reproduction, and quality enhancement [15], [43], [45], [52]. At first, research focused on whether these models, trained primarily on natural language and code, could be adapted to the specific demands of bug reports. For instance, RTA [24] pre-trained a generative LLM on a large corpus of bug reports and then fine-tuned it for summarization, achieving state-of-the-art performance on task-specific datasets. Building on this foundation, SumLLaMA [92] extended CodeLLaMA [78] by distinguishing relevant from irrelevant content through contrastive pre-training. By drawing related information closer in its internal representation space and separating irrelevant details, it improved the ability to prioritize key content in bug reports. Crucially, instead of updating all parameters, SumLLaMA applied parameter-efficient fine-tuning with Low-Rank Adaptation (LoRA), updating only a small subset of parameters to reduce computational and memory costs [34]. This line of work illustrated the trade-off between specialization and efficiency, proving that even large models could be effectively tuned for bug report summarization without prohibitive expense.

Once models could generate fluent summaries, the next challenge became assessing their quality. Traditional metrics like ROUGE or BLEU are difficult to interpret in a software engineering context and often do not correlate with human judgment [77]. To address this, researchers began to repurpose LLMs themselves as evaluators [52]. By prompting models such as GPT-4o [3], LLaMA-3 [28], and Gemini [84] to judge summaries along dimensions like coherence, informativeness, and completeness, studies showed that LLMs could provide nuanced evaluations that aligned well with human judgments, even on complex tasks. This not only introduced a scalable evaluation alternative but also highlighted a new dependency: evaluation outcomes themselves were sensitive to prompt phrasing and model choice, underscoring the importance of careful design.

However, generating fluent summaries came with a downside: hallucination [35]. LLMs could invent plausible but false details or omit crucial steps, creating outputs that misled rather than helped developers. This limitation shifted attention toward prompt engineering as a mechanism to guide and constrain model behavior. Instead of retraining models for each scenario, researchers explored structured, role-based, and progressive prompting techniques to reduce hallucinations and make outputs more reliable [43].

Several systems demonstrated how well-crafted prompts could reshape LLM behavior in bug report contexts. For example, ChatBR [15] used structured prompts that

assigned roles like “senior software engineer,” specified output formats, and embedded domain-specific instructions. This enabled the model to enhance low-quality bug reports by clarifying ambiguous sentences, suggesting reproduction steps, and adding missing environmental details, significantly improving readability and completeness. Similarly, another study [2] investigated whether instruction-tuned LLMs could standardize casual, unstructured reports by transforming them into high-quality, actionable bug reports. Using prompt templates with error messages, logs, and reproduction steps, these models generated outputs that rivaled human-written reports in accuracy and relevance.

Prompting was also shown to bridge the gap between natural language descriptions and executable artifacts. The LIBRO framework [45] demonstrated how task-specific prompts and few-shot examples could help LLMs reproduce bugs by generating ranked test cases, easing a traditionally time-consuming process. Likewise, LLPut [6] revealed that prompts could guide LLMs to extract failure-inducing input commands from unstructured reports, outperforming earlier BERT-based approaches in translating ambiguous natural language into concrete inputs. Both studies highlighted that LLMs excel when prompts anchor their generation to domain-relevant structure, though they also made clear that verification against real code remains essential.

Other research sought to integrate LLM capabilities with established program analysis methods. LLIFT [55] combined in-context learning with progressive prompting and task decomposition to handle token limitations. By revealing only necessary details step by step and breaking down complex bug detection into smaller tasks, LLIFT allowed LLMs to understand function post-constraints more effectively and improved practical bug detection in large codebases. This integration showed that prompting strategies could complement static analysis, enabling hybrid systems that balance symbolic rigor with language-model flexibility.

Beyond bug reports specifically, a parallel line of work explored how to make LLM reasoning itself more efficient and controllable. Compressed chain-of-thought techniques encoded intermediate reasoning as dense vector representations rather than long textual chains, reducing token usage and computational cost while preserving accuracy [89]. This matters for bug report workflows that require multi-step reasoning, such as diagnosing causes from logs or cross-referencing multiple files. Similarly, foundational work on GPT-3 [16] demonstrated that few-shot prompting could teach models new tasks from minimal examples, while probing studies revealed that model knowledge is highly sensitive to prompt phrasing and format [41], [101]. Together, these findings reinforced the centrality of prompt design: small changes in instruc-

tions or exemplars can determine whether outputs are consistent, accurate, and complete.

Building on this foundation, researchers developed prompt-based approaches for controllable summarization. PromptSum [76] introduced prompts that explicitly specify attributes like summary length, style, or focus, offering parameter-efficient alternatives to fine-tuning. Meanwhile, automatic semantic augmentation enriched prompts with contextual cues such as repository metadata, variable names, and data-flow information, enabling models to generate more meaningful and accurate summaries without additional training [5]. These techniques demonstrated that both the structure and the content of prompts directly shape output quality, showing that prompting is not merely about wording but about embedding contextual signals that ground the model's generation.

Similarly, recent work explored ChatGPT's capabilities on vulnerability management, including bug report summarization [60]. This study highlights challenges such as prompt sensitivity and hallucination, and proposes self-heuristic prompting as a strategy to improve output reliability, demonstrating how careful prompt design can enhance LLM performance in technical domains.

Across these diverse directions, a unifying theme emerges: while LLMs provide the raw capacity to generate, evaluate, and even reason about bug reports, their utility depends heavily on how they are guided. Domain adaptation methods like LoRA and contrastive pre-training enhance specialization; evaluation methods using LLM judges provide nuanced but prompt-sensitive quality checks; and prompt engineering through role assignment, progressive disclosure, semantic augmentation, and task decomposition serves as the bridge between general-purpose models and domain-specific requirements. Together, these strategies illustrate a trajectory from capability to reliability: moving from "what LLMs can do" toward "how to make them consistently useful for developers." Yet important challenges remain, including ensuring faithfulness, developing standardized evaluation pipelines, and balancing efficiency with controllability. The body of work thus suggests that the future of LLMs in software maintenance will rely not only on more powerful models but on increasingly sophisticated ways of structuring and guiding their use.

2.1.2 Graph-Based Approaches

These approaches leverage graph neural networks to capture and model relationships between sentences, entities, or code segments within bug reports. They effectively

capture the structural and relational information within reports.

One representative example is the Bug Report Summarization for Bug Localization (BRS_BL) method [55], which directly ties summarization to the downstream task of localizing buggy code. In this framework, bug reports are transformed into heterogeneous graphs with multiple node types such as sentences, keywords, and code-related entities, with edges that encode their relationships. This design allows the model to capture nuanced interactions across different report components. To handle the challenge of large source files, where only a few lines may actually contain the defect, BRS_BL splits code into smaller chunks, making feature extraction more precise. Summarized bug reports are then matched to these code chunks using not only semantic similarity but also software-specific features, such as the history of prior bug fixes. A fine-grained matching module integrates these signals, and a fusion layer computes a final relevance score that identifies the most likely faulty code segments. In doing so, BRS_BL demonstrates how graph-based summarization can move beyond text compression to actively support developer tasks like bug localization.

2.1.3 Neural Learning Approaches

These approaches leverage specialized deep learning architectures to process bug reports, capturing sentence importance, contextual relationships, and domain-specific patterns to generate more accurate and informative outputs.

Early neural methods emphasized identifying the most significant parts of a report. One approach introduced the idea of computing sentence significance factors [50], where a model assigns numerical importance scores to each sentence based on semantic content, position within the report, and contextual links with surrounding sentences. This mechanism naturally prioritizes sentences that align with the report's title or appear early in the description, producing concise summaries that capture essential information such as the problem statement, reproduction steps, and environmental details.

While this represented an important step, later research recognized that simply ranking sentences was not enough. Bug reports often include multiple perspectives from the initial description to developer comments, and extracting these in isolation risks losing the coherence of the report as a whole. To address this, BugSum [59] was developed as a neural architecture that encodes different report components jointly. By learning how titles, descriptions, and comments relate to each other, BugSum is able to filter redundancy and highlight truly relevant content, resulting in summaries that

are not only informative but also contextually coherent.

Building on this foundation, researchers explored how pre-training could further improve summarization models. The Knowledge-Specific and Contrastive Learning Pre-Training (KSCLP) framework [81] exposed models to large corpora of bug reports and software development text, allowing them to develop familiarity with domain-specific terminology and concepts. In addition, contrastive learning was used to train the system to detect subtle differences between bug reports, improving its ability to discriminate important details from irrelevant noise. This pre-training strategy allowed the model to generate more accurate and discriminative summaries that better serve the needs of developers.

Finally, a different line of research focused on ensuring that all critical attributes of a bug report are preserved in the summary. The ClaSum [66] framework approached the problem through multidocument summarization, treating elements such as the description, reproduction steps, and environmental details as related but distinct sources of information. By classifying and ranking sentences across these components, ClaSum produced structured summaries that retained essential attributes often overlooked in earlier methods. This approach ensured that summaries were not only concise but also complete enough to support effective debugging.

Taken together, these neural learning approaches trace a progression in the field: from identifying significant sentences, to modeling contextual relationships across report components, to leveraging domain-specific pre-training, and finally to ensuring completeness through multidocument integration.

2.1.4 Information Retrieval Approaches

These approaches extract key information from bug reports by ranking sentences or segments based on features like term frequency and relevance. These methods provide concise, interpretable summaries but may struggle to capture deeper contextual relationships.

To address the challenge of extracting relevant information from lengthy bug reports, an effective summarization approach combines keyword extraction with sentence-level analysis [42]. Key terms are identified using the Term Frequency–Inverse Document Frequency (TF-IDF) method, highlighting words that are both frequent and significant within the report. Sentences containing these high-value keywords are then scored, ranked, and combined to generate concise summaries that emphasize the core aspects of a bug report. By integrating keyword detection with sentence-level

ranking, this approach improves bug review, prioritization, and developer comprehension. Its main limitation, however, is reliance on textual features alone, which leaves out contextual information such as stack traces, runtime logs, and source code that are often crucial for debugging.

2.1.5 Semantic Analysis Approaches

These approaches leverage semantic information to identify key sentences, using similarity measures or semantic classes. Unlike information retrieval methods, which rely on keyword frequency and sentence ranking and often fail to capture deeper meaning, semantic analysis approaches focus on the relationships between sentences and their roles in conveying critical information.

One strategy measures the semantic similarity between sentences using vector-based text representations, allowing the model to identify which sentences are most closely related in meaning [27]. Sentences with the highest centrality in this semantic network are then selected to form a summary, ensuring that the core aspects of the bug are preserved while redundancy is minimized. This method effectively captures important contextual information that purely statistical approaches may overlook, though it remains limited to textual descriptions and does not incorporate additional data such as code snippets.

Building on this idea, more advanced techniques have explored the use of semantic classes to better identify informative content. For instance, a Two-Layer Semantic Model (TSM) [95] distinguishes between sentences that describe actions performed by people, labeled as Anthropogenic, and those that detail steps to reproduce a bug, labeled as Procedural. The first layer of the model filters out less informative sentences, while the second layer applies a supervised logistic regression model to rank and select the most relevant sentences. By combining semantic class distinctions with a structured ranking process, this approach produces concise summaries that highlight the information most critical for debugging, offering a more targeted and context-aware alternative to earlier methods.

2.1.6 Supervised Approaches

These approaches leverage supervised learning, a method where models are trained on labeled data to learn which sentences are important. They often use logistic regression with engineered features or attributes to rank and select key sentences from bug reports.

A key insight in this domain is that duplicate bug reports often contain overlapping or complementary information, which can be leveraged to generate more informative summaries. PageRank-based Summarization Techniques (PRST) [32] exploit this by computing sentence importance using similarity metrics across both master and duplicate bug reports. A supervised logistic regression model then predicts the likelihood that each sentence belongs in the summary, and these scores are combined to select the top-ranked sentences, ensuring that critical details are retained while redundancy is minimized.

Another direction involves incorporating human insight to enhance attribute design. Crowd-Attribute (CA) method [38] systematically infers new features from summary sentences elicited from volunteers. By reviewing bug reports, volunteers identify sentences they consider important and provide reasoning for their choices. Heuristic rules are then applied to extract, filter, and merge these candidate attributes, which are fed into a logistic regression model to predict sentence importance. This approach demonstrates that leveraging diverse human perspectives can improve the quality and accuracy of automatically generated summaries, especially in capturing subtleties that purely statistical features might miss.

Beyond duplicate reports and crowdsourced attributes, supervised methods have also explored applying general conversation-based summarization techniques to bug reports [75]. Classifiers trained on bug report corpora use logistic regression over a wide range of features, including structural elements, participant involvement, sentence length, and lexical content. Summaries are generated by selecting the top-ranked sentences up to a fixed portion of the report's length. Task-based evaluations with developers have shown that these summaries facilitate tasks such as duplicate detection, enabling faster completion without sacrificing accuracy. This confirms the practical usefulness of supervised approaches in real-world software maintenance scenarios.

2.1.7 Unsupervised Approaches

These approaches leverage unsupervised learning, where models identify important sentences without relying on labeled data, making them particularly useful when annotated corpora are scarce. Techniques include clustering based on semantic and syntactic features, cognitive-inspired heuristics to mimic human reading patterns, and deep representation learning to capture contextual relationships.

One early insight is that developers often do not read bug reports line by line but instead scan them quickly to locate critical details [62]. Emulating this "hurried" read-

ing process, some methods incorporate cognitive-inspired heuristics to prioritize sentences that are more likely to be noticed under time constraints. Positional bias, which favors sentences appearing earlier, and keyword density are used to highlight the most salient content, producing concise summaries that retain essential elements such as problem descriptions and reproduction steps.

Other unsupervised techniques leverage deep learning to capture richer semantic and contextual relationships. By representing sentences as embeddings, models can compute pairwise similarities and identify clusters of related content, ensuring that summaries reflect both centrality and diversity of information. Approaches like DeepSum [56] illustrate this strategy, modeling deeper representations that are robust to variations in vocabulary and domain-specific phrasing. Clustering then selects representative sentences that cover the most important aspects of the report, producing summaries that are more informative and comprehensive than those generated by traditional extractive methods, though some redundancy may persist when semantically similar sentences are scattered throughout the report.

Industrial bug reports often contain additional noise, including email dumps, chat transcripts, which can reduce the effectiveness of summarization. To address this, certain unsupervised frameworks classify sentences into categories such as questions, investigative statements, code snippets, and others, filtering out irrelevant content before applying clustering or selection algorithms [63]. This noise reduction ensures that the final summaries focus on meaningful, actionable information.

Finally, some approaches adopt a multi-view, multi-objective perspective [64], recognizing that both semantic and syntactic features contribute to sentence importance. Sentences are clustered independently based on semantic embeddings, such as those generated by BERT, and syntactic features like TF-IDF. Multi-objective optimization then reconciles these different views, forming consensus clusters from which key sentences are selected. This strategy allows summaries to capture diverse and relevant content across the report, balancing coverage, centrality, and relevance, and representing an advanced evolution of unsupervised summarization techniques.

2.2 Critical Analysis and Synthesis of Existing Approaches

In this section, we move beyond a simple overview to critically examine the literature on bug report summarization in software development. By assessing the strengths,

limitations, and underlying methodologies of existing approaches, we identify recurring themes, methodological patterns, and gaps in the field. This critical synthesis not only provides a deeper understanding of how bug report summarization has evolved but also sets the stage for situating the contributions of this thesis within the broader research landscape, highlighting where novel insights and improvements are needed.

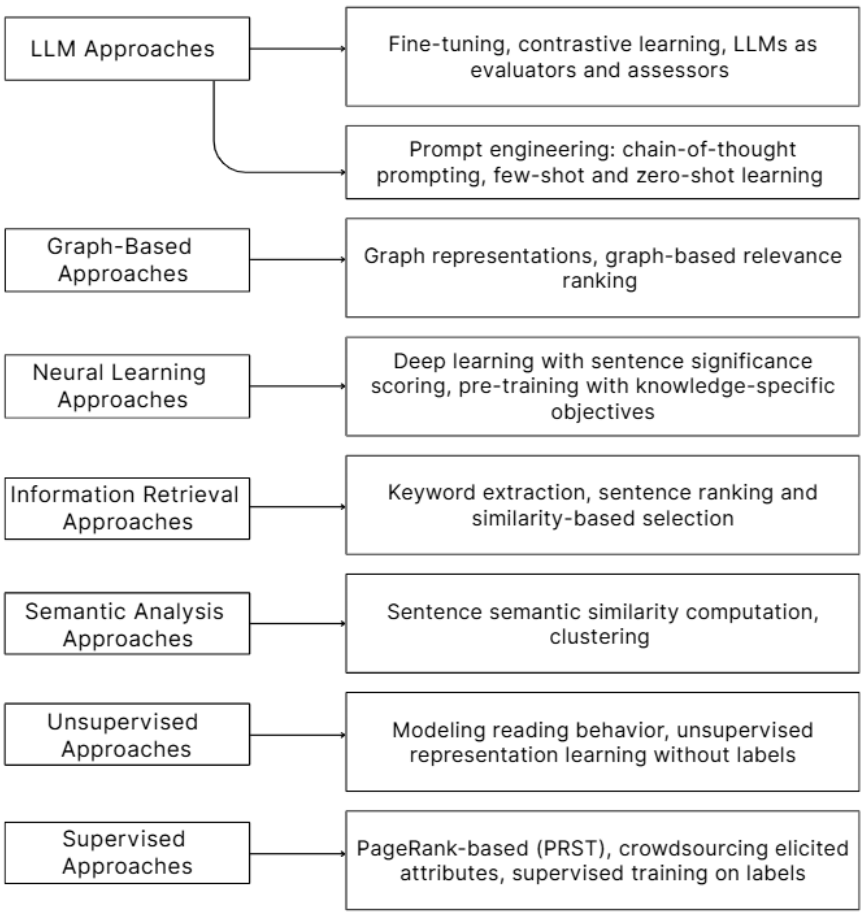


Figure 2.1: A taxonomy of bug report summarization approaches, categorizing existing techniques into LLM-based, graph-based, neural learning, information retrieval, semantic analysis, and supervised/unsupervised methods

2.2.1 Comparison of Different Approaches

A dominant trend in bug report summarization is the prevalence of extractive methods [47], alongside the emerging use of LLM-based abstractive approaches [97]. Extractive methods such as SSBRSS [27], TSM [95], Hurried [62], AUSUM [63], DeepSum [56], BugSum [59], ClaSum [66], Automatic Keyword and Sentence-Based [42], PRST [32], Crowd-Attribute [38], Automatic Summarization [75], Multi-View Multi-Objective Optimization Framework [64], and Sentence Significance Factor-based approaches [50] focus on selecting the most relevant sentences from bug reports rather

than generating new text. While SSBRSS emphasizes semantic similarity and sentence centrality, TSM extends this paradigm by introducing Anthropogenic and Procedural classes, allowing it to capture critical sentences that other extractive techniques tend to overlook. Similarly, cognitive and human-in-the-loop strategies, such as Hurried and Crowd-Attribute, simulate developer reading behavior or incorporate crowdsourced insights. This makes them more adaptive than purely statistical approaches like the Automatic Keyword and Sentence-Based method, which are prone to missing nuanced contextual details.

Within extractive methods, there is a clear distinction between supervised and unsupervised strategies. PRST, Automatic Summarization, and Crowd-Attribute exemplify supervised approaches, where annotated datasets and engineered features enable higher accuracy. In contrast, unsupervised techniques such as Hurried, DeepSum, AUSUM, and the Multi-View Multi-Objective Optimization Framework avoid the need for labeled data by relying on embeddings, clustering, or heuristics, offering scalability at the cost of occasional precision loss. Neural extractive models, such as DeepSum and Sentence Significance Factor-based approaches, further improve coverage by embedding contextual information, outperforming simpler keyword-driven methods like Automatic Keyword and Sentence-Based. Hybrid models like BRS_BL [100] go a step further by integrating semantic, software-specific, and code-contextual features. Unlike most extractive methods, however, BRS_BL does not use summarization as its primary goal but rather as a supporting mechanism to enhance bug localization, demonstrating the broader utility of summarization in downstream software engineering tasks.

In contrast, abstractive approaches such as RTA [24], SumLLaMA [92], KSCLP [81], and PromptSum [76] represent a shift toward generating fluent, human-like summaries. RTA serves as an early benchmark, pre-training a generative LLM pre-trained on bug reports and achieving strong task-specific performance. SumLLaMA enhances this by combining contrastive pre-training with LoRA fine-tuning, efficiently adapting to bug-report summarization. KSCLP, however, emphasizes knowledge-specific pre-training and contrastive learning, capturing domain-specific semantics that are difficult for generic models. Unlike both, PromptSum reduces the need for fine-tuning by focusing on prompt engineering, controlling aspects like length, style, and content focus. In comparison, recent work exploring ChatGPT's capabilities in vulnerability management [60] also leverages prompt-based techniques for bug report summarization, but highlights additional challenges such as prompt sensitivity and hallucination. The study proposes self-heuristic prompting as a strategy to mitigate these issues, illustrating how prompt design can influence both effectiveness and reliability. Com-

pared to extractive approaches, these LLM-based methods offer superior fluency and readability but struggle with ensuring technical accuracy and balancing brevity with detail, an especially critical concern in technical domains where minor omissions can obscure root causes or mislead developers.

Several interconnections can be observed across these approaches. For example, both TSM and BRS_BL demonstrate the advantages of integrating semantic modeling with supervised ranking or multi-source features such as code awareness, thereby improving precision. Likewise, unsupervised models like DeepSum and Hurried conceptually align with SSBRSS in leveraging semantic similarity for sentence selection, though they differ in whether human feedback is included. Neural extractive models and hybrid strategies often outperform simpler statistical or keyword-based systems, reinforcing the theme that semantic richness and contextual embeddings enhance coverage and informativeness of summaries. Beyond traditional extractive approaches, LLM-based and prompt-driven techniques also share conceptual links. PromptSum and recent studies exploring ChatGPT's capabilities demonstrate that controlling summary content through prompts or self-heuristic strategies can focus on relevant information, without requiring extensive model retraining.

Table 2.2 provides a comparative overview of the major bug report summarization approaches, highlighting their year of introduction, techniques, and focus areas. Overall, extractive methods continue to dominate due to their reliability, accuracy, and interpretability, whereas abstractive approaches, particularly LLM-based ones, show promise in improving fluency and readability. However, abstractive methods face the critical challenge of maintaining technical accuracy. Building on insights from hybrid extractive approaches like BRS_BL, which leverage code-contextual information to enhance downstream tasks such as bug localization, future work should emphasize abstractive prompting strategies that integrate code context alongside textual content to produce summaries that are both readable and technically comprehensive. Such directions hold the potential to close the gap between informativeness and accuracy in bug report summarization.

Table 2.2: Comparison of Bug Report Summarization Approaches

Category	Paper Name	Year	Technique	Key Focus
LLM Approaches	RepresentThemAll: A Universal Learning Representation of Bug Reports [24]	2023	Abstractive	Pre-trained LLM for bug reports with multi-task capabilities using masked language modeling and contrastive learning
	SumLLaMA: Efficient Contrastive Representations and Fine-Tuned Adapters for Bug Report Summarization [92]	2024	Abstractive	LLM fine-tuning with adapters
	PromptSum: Parameter-Efficient Controllable Abstractive Summarization [76]	2023	Abstractive	Prompt-based controllable summarization
	Exploring ChatGPT's Capabilities on Vulnerability Management [60]	2024	Abstractive	Prompt-based general-LLM
Graph-Based Approaches	Enhancing Bug Localization through Bug Report Summarization [100]	2023	Extractive	Graph-based bug localization aided by summarization
Neural Learning Approaches	Deep Learning-Based Bug Report Summarization Using Sentence Significance Factors [50]	2022	Extractive	DL model with significance factor weighting

Category	Paper Name	Year	Technique	Key Focus
	BugSum: Deep Context Understanding for Bug Report Summarization [59]	2020	Extractive	Contextual deep learning for summaries, believability score
	Enhancing Bug Report Summaries Through Knowledge-Specific and Contrastive Learning Pre-Training [81]	2024	Abstractive	Contrastive and knowledge-specific pretraining
	A Multidocument Summarization Technique for Informative Bug Summaries [66]	2024	Extractive	Rule-based scoring, classification
Information Retrieval Approaches	Automatic Keyword and Sentence-Based Text Summarization for Software Bug Reports [42]	2020	Extractive	Keyword extraction, sentence scoring, statistical features
Semantic Analysis Approaches	Summarization of Software Bug Report based on Sentence Semantic Similarity (SSBRSS) [27]	2024	Extractive	Semantic similarity-based summarization
	Towards an Improvement of Bug Report Summarization Using Two-Layer Semantic Information [95]	2018	Extractive	Extending semantic model with anthropogenic and procedural information

Category	Paper Name	Year	Technique	Key Focus
Unsupervised Approaches	Hurried: Modelling the 'Hurried' Bug Report Reading Process to Summarize Bug Reports [62]	2012	Extractive	Cognitive-inspired unsupervised summarization
	DeepSum: Unsupervised Deep Bug Report Summarization [56]	2018	Extractive	Unsupervised DL model for bug report summarization
	AUSUM: Approach for unsupervised bug report summarization [63]	2012	Extractive	Noise reducer, unsupervised algorithms
	Bug Report Summarization using Multi-View Multi-Objective Optimization Framework [64]	2022	Extractive	Multi-view (semantic and syntactic) multi-objective optimization approach
Supervised Approaches	PRST: A PageRank-Based Summarization Technique for Summarizing Bug Reports with Duplicates [32]	2017	Extractive	PageRank-based scoring for duplicate bug reports
	Towards Better Summarizing Bug Reports with Crowdsourcing Elicited Attributes [38]	2018	Extractive	Crowd-sourced attributes for heuristic scoring
	Automatic Summarization of Bug Reports [75]	2014	Extractive	Feature-engineered supervised learning

2.2.2 Strengths and Weaknesses of Existing Approaches

Bug report summarization has evolved from basic keyword-based extraction to sophisticated neural and LLM-based methods, each with distinct strengths, limitations, and application contexts.

Supervised extractive approaches [32], [75], [95] provide higher accuracy but rely on labeled data, whereas unsupervised approaches [27], [42], [56], [63], [64] offer scalability but sometimes at the cost of precision. Human-in-the-loop and cognitive methods [38], [62] contribute valuable insights but are limited in automation and scalability. Neural and hybrid methods [50], [59], [66], [81] show that contextual embeddings and code-awareness enhance coverage and downstream utility. Abstractive LLM-based methods [24], [60], [76], [81], [92] prioritize fluency and readability but require careful design to preserve technical correctness. BRS_BL [100] exemplifies how summarization can serve broader goals, bridging content extraction with software engineering tasks like bug localization. To provide a more detailed perspective, Table 2.3 summarizes the key strengths and limitations of prominent bug report summarization approaches. The table highlights not only the methodological diversity from rule-based and statistical methods to neural and LLM-based approaches, but also the practical trade-offs in terms of accuracy, scalability, computational cost, and applicability to downstream software engineering tasks.

Table 2.3: Strengths and Limitations of Bug Report Summarization Approaches

Paper	**Strength**	**Limitations**
Summarization of Software Bug Report based on Sentence Semantic Similarity (SSBRSS) technique [27]	Captures sentence importance using semantic similarity and centrality; Lightweight and easy to implement without deep models	May miss deeper contextual nuances compared to transformer-based embeddings
Automatic Keyword & Sentence-Based Text Summarization for Software Bug Reports [42]	Fast, simple extraction	Focuses on surface features; fuzzy clustering may misgroup sentences; redundancy removal may discard semantically important sentences

Paper	Strength	Limitations
Hurried: Modelling the 'Hurried' Bug Report Reading Process to Summarize Bug Reports [62]	Human-like cognitive modeling; light-weight and useful in real-time settings	Simplified heuristics; lacks semantic/contextual representation
Towards Better Summarizing Bug Reports with Crowdsourcing Elicited Attributes [38]	Incorporates crowd-sourced insights; heuristic-based approach integrates human understanding	Performance depends on volunteer quality; no deep learning
Towards an Improvement of Bug Report Summarization Using Two-Layer Semantic Information [95]	Dual-layer semantic modeling (Anthropogenic & Procedural); better context capture	May be computationally more complex; depends on proper classification of sentences
AUSUM: Approach for unsupervised bug report summarization [63]	Noise reducer to filter content	May not generalize across projects; noise-reduction heuristics require reconfiguration for each subject
PRST: A PageRank-Based Summarization Technique for Summarizing Bug Reports with Duplicates [32]	Supervised; high precision with annotated data	Requires high-quality labels; limited generalization
Automatic Summarization of Bug Reports [75]	Supervised; engineered features improve accuracy	Label-dependent; may not generalize well
Deep Learning-Based Bug Report Summarization Using Sentence Significance Factors [50]	Contextual embedding-based ranking	Outdated architecture; may not consider bug report-specific characteristics
DeepSum: Unsupervised Deep Bug Report Summarization [56]	Embedding-based clustering improves coverage; goes beyond lexical similarity	Time-consuming (5.66 min on average per report); ignores inter-sentence semantic flow

Paper	Strength	Limitations
Enhancing Bug Localization through Bug Report Summarization [100]	Integrates semantic, software-specific, code features; enhances bug localization	Computationally intensive; summarization is secondary objective
A Multidocument Summarization Technique for Informative Bug Summaries [66]	Rule-based scoring ensures consistency	Lacks generalization; limited semantic capture
BugSum: Deep Context Understanding for Bug Report Summarization [59]	Designed specifically for bug-related threaded discussions; believability scoring from discussion replies	Depends on availability and quality of discussion replies
Bug Report Summarization using Multi-View Multi-Objective Optimization Framework [64]	Multi-objective optimization; diversity (semantic and syntactic representations) and relevance	Computationally intensive
SumLLaMA: Efficient Contrastive Representations and Fine-Tuned Adapters for Bug Report Summarization [92]	Contrastive pre-training with LoRA; fluent summaries	High computational cost; lacks standard zero/few-shot prompting
Enhancing Bug Report Summaries Through Knowledge-Specific and Contrastive Learning Pre-Training [81]	Knowledge-specific pretraining; contrastive learning; domain semantics	Computationally expensive contrastive pretraining
PromptSum: Parameter-Efficient Controllable Abstractive Summarization [76]	Prompt-based control over summary style/length	Requires careful prompt design; may miss technical details; not specifically adapted to bug reports

Paper	Strength	Limitations
Exploring ChatGPT's Capabilities on Vulnerability Management [60]	Flexible across tasks; fluent summaries	Sensitive to prompt design; hallucination; requires self-heuristic prompting

2.3 Identifying Gaps and Opportunities

Despite significant progress in bug report summarization, several limitations remain in the existing literature. A recurring issue is the reliance on narrow or domain-specific datasets such as BRC [75], SDS [39], and ADS [40]. While these datasets have enabled model development and evaluation, their limited scope reduces generalizability, causing models to struggle with bug reports from diverse software environments or with greater complexity.

Another key gap is the lack of multimodal integration. Most studies focus solely on textual content and neglect code snippets, which are critical for understanding the full context of a bug [13], [65], [68]. Traditional extractive methods and neural approaches typically emphasize semantic or statistical features derived from text [61], [68], producing summaries that may miss essential technical details necessary for accurate bug reproduction and resolution.

Abstractive LLM-based approaches address fluency and readability but face additional challenges [4], [57]. They are sensitive to prompt design, prone to hallucinations, and often struggle to maintain technical precision [35], [41], [101]. Furthermore, these models are rarely evaluated on heterogeneous datasets, limiting confidence in their generalizability and reliability. Existing LLM approaches also largely ignore code context [24], [60], [76], [81], [92], leaving the multimodal challenge unresolved. Due to their computational requirements, they may be impractical for real-time or large-scale bug report summarization [25], [80]. Compared to extractive methods, research on abstractive LLM-based summarization remains limited, leaving many questions about their effectiveness and applicability open [30].

Algorithmic and methodological limitations persist across other approaches as well. Methods such as Deep Learning-Based Sentence Significance Factors [50], ClaSum [66], and the MVMO Optimization Framework [64] can be computationally intensive, affecting scalability. Human-in-the-loop or crowd-sourced approaches, like CrowdAttribute [38], provide valuable insights but are constrained in automation and re-

producibility, making them difficult to deploy at scale.

Overall, the major gaps include limited datasets, insufficient focus on abstractive methods, inadequate integration of code context, high computational costs, and challenges in maintaining technical precision and generalizability. Addressing these gaps presents opportunities for future research and motivates this thesis's focus on developing bug report summarization methods that effectively combine textual and code information to produce accurate, informative, and reliable summaries.

2.4 Summary

Bug report summarization has progressed from simple extractive methods to neural and LLM-based approaches. Extractive techniques are reliable and interpretable but often miss context, while semantic, graph-based, and hybrid models improve coverage and informativeness by capturing relationships among sentences and code. Supervised methods achieve high accuracy, whereas unsupervised methods offer scalability. Abstractive LLM-based approaches provide fluent, human-like summaries and multi-task capabilities but face challenges in technical accuracy, prompt sensitivity, code integration, and computational cost. Key gaps remain in dataset diversity, multimodal integration, and balancing readability with correctness. These insights motivate this thesis to develop a method that combines textual and code information to generate concise, accurate, and informative bug report summaries.

Chapter 3

Proposed Methodology

In this chapter, we present the methodology that we developed to generate abstractive summaries of software bug reports. Our approach combines information from both textual bug descriptions and corresponding code snippets, allowing us to produce summaries that are contextually meaningful and technically precise. We first provide a high-level overview of the methodology and then describe its components in chronological order in the subsequent sections.

3.1 Overview of the Methodology

The goal of our work is to design a system that generates summaries of software bugs by combining natural language bug reports with their corresponding buggy code snippets. These summaries aim to capture the essential aspects of the bug, with emphasis on the root cause.

To achieve this goal, we developed a staged pipeline in which each component progressively refines the input into a concise, meaningful summary. The pipeline integrates both textual and code-based information, enabling the system to provide outputs that are technically precise yet easy for developers to interpret during debugging.

Our methodology consists of four sequential stages: data pre-processing, prompt engineering, fine-tuning, and evaluation. In the pre-processing stage, we clean, filter, and align bug reports with the corresponding code snippets to ensure data quality. We then apply prompt engineering to leverage the capabilities of pre-trained large language models (LLMs) in a cost-effective way, enabling us to obtain early results without the computational expense of full retraining. Building on this, we fine-tune the models to incorporate domain-specific knowledge of software engineering, thereby

improving their ability to understand bug reports and code changes. Finally, we evaluate the generated summaries using automated metrics to ensure that the outputs are accurate, fluent, and relevant. Figure 3.1 illustrates an overview of our proposed approach.

When designing this methodology, we were motivated by several considerations. First, we combined bug reports with code snippets because these two sources provide complementary perspectives: bug reports capture the context of the issue, while code snippets reveal the technical details of the fault. Second, we chose abstractive summarization over extractive methods to generate summaries that are not simple fragments of the original text, but instead coherent and informative narratives. Third, we adopted a staged pipeline to balance efficiency with performance. By starting with prompt engineering, we were able to quickly adapt LLMs to the task, while fine-tuning allowed us to inject domain-specific knowledge for more accurate results.

3.2 Chronological Breakdown of Components

3.2.1 Data Preprocessing

The first step in our methodology involved preparing the dataset for use in the summarization pipeline. Although Defects4J [44] provides links to bug reports and code snippets [82], the raw data was not structured for immediate processing. We therefore designed a preprocessing procedure to extract, clean, and organize the data.

Extraction

The initial task was to extract relevant information from Defects4J [44]. While the dataset provides URLs pointing to bug reports and code snippets [82], we implemented a scraping process to automatically parse, retrieve and store content locally. Due to variations in how different projects present their bug reports and code changes via different interface layouts, the scraping procedure required project-specific handling. We ensured that both bug reports and their corresponding buggy and fixed code snippets were successfully collected.

Figure 3.2 illustrates the extraction process for both bug reports and code snippets. Subfigures 3.2(a) and 3.2(b) show a bug report in its original URL form and the corresponding content obtained after scraping, respectively. Similarly, subfigures 3.2(c) and 3.2(d) display a patch code snippet as provided in URL form and the structured content extracted from it. These examples demonstrate how the scraping process

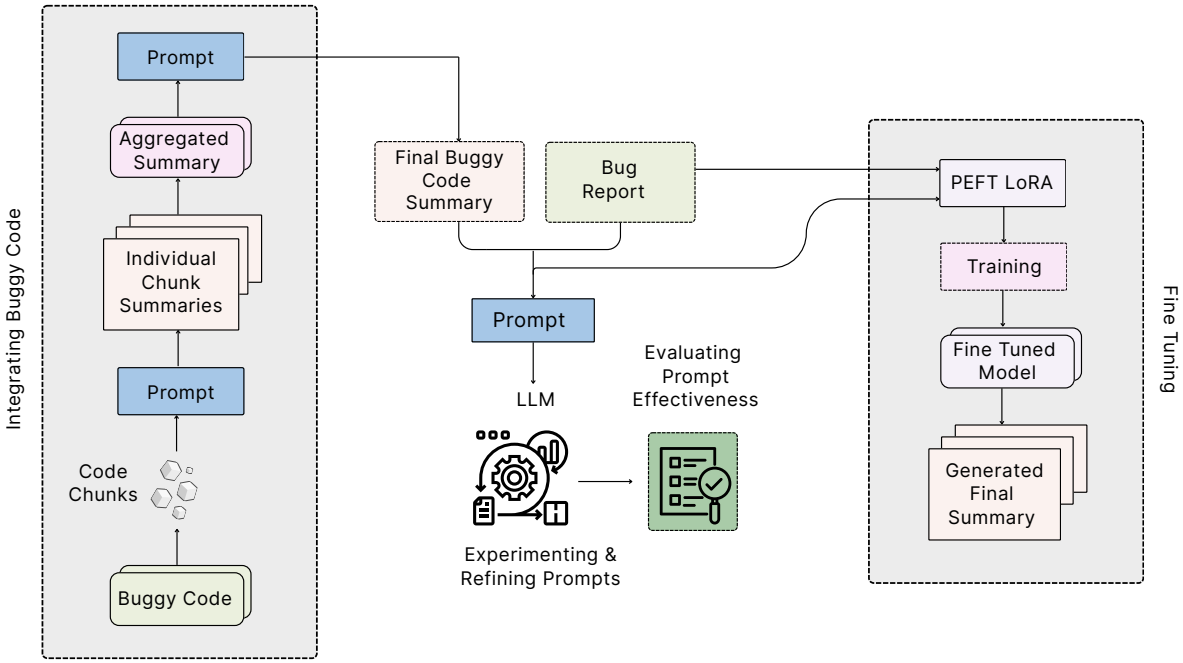
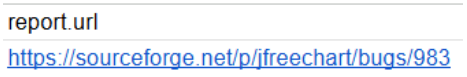


Figure 3.1: End-to-end pipeline for bug report summarization. The methodology consists of three stages: (1) Chunking and summarizing buggy code – a long buggy code snippet is divided into smaller code chunks that fit within an LLM's context window. Each chunk is independently summarized, and the chunk-level summaries are aggregated into a final code-level summary. (2) Integrating bug reports and code summaries – the bug report and the aggregated code summary are combined through structured zero-shot, one-shot, or few-shot prompts, producing the complete abstractive summary. (3) Fine-tuning with parameter-efficient adaptation – an LLM is adapted using Low-Rank Adaptation (LoRA), yielding two fine-tuned variants: one trained on bug reports only and another trained on bug reports with code summaries. This pipeline enables systematic experimentation with prompting strategies and efficient model adaptation, ensuring concise summaries that capture both natural-language and code-level bug contexts.

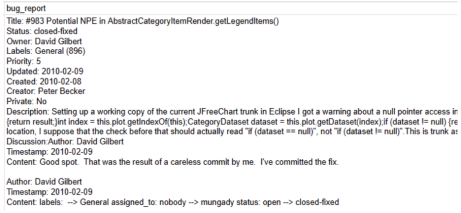
transforms raw web-based data into organized, locally stored text and code, making it suitable for subsequent preprocessing and model input.

Cleaning

Although the scraped data was generally well-structured, we verified that all textual content and code snippets were consistently formatted and free from extraneous HTML tags or other artifacts. This ensured that the data remained organized and compatible with subsequent model processing steps.

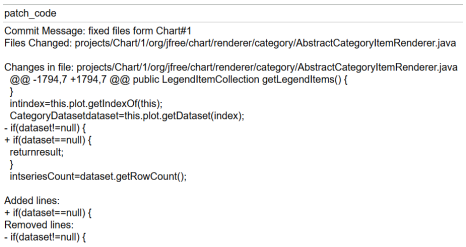


(a) An example of a bug report in URL form



(b) Bug report content after scraping

(c) An example of a patch code snippet in URL form



(d) Patch code content after scraping

Figure 3.2: An example of an extracted bug report and a code snippet.

Processed Dataset

After extraction and cleaning, we compiled a structured dataset, pairing each bug report with its associated buggy code snippets. During this stage, we observed that only about half of the records contained complete pairs (bug report with code snippets); incomplete entries were therefore discarded. The final dataset consists of high-quality text-code pairs suitable for model input.

- Input: Raw Defects4J [44] dataset (URLs pointing to bug reports and code snippets [82]).
- Output: Cleaned and structured text-code pairs.

We selected Defects4J over other datasets because it includes real-world bug reports and code changes across a variety of Java projects, making it highly relevant for this task.

3.2.2 Prompt Engineering

After data preprocessing, we performed prompt engineering to prepare the inputs for our models. Since the design of prompts strongly affects the quality of summaries produced by large language models, we experimented with multiple prompting strategies to identify the most effective approach for summarizing bug reports and code snippets. Instead of modifying the models themselves, we relied on carefully structured prompts to shape how the models generated outputs. This approach allowed us to im-

prove summary quality without additional training or fine-tuning, thereby reducing computational cost.

Zero-shot prompting

We began with zero-shot prompting by providing the model only with the bug report and task instructions, without any examples of prior summaries. We structured the prompt to clearly highlight the underlying issue of the bug while keeping the wording concise. Figure 3.3 illustrates an example of a zero-shot prompt we used.

```
prompt = f"""Given the bug report, Write a one-sentence summary of the core issue using no more than 10 words.


Bug Report:
{bug_report}

Summary :"""
```

Figure 3.3: An example of a zero-shot prompt for bug reports

One-shot prompting

We implemented one-shot prompting by including a single example of a bug report and its corresponding summary alongside the task instructions. We carefully structured the prompt to guide the model without biasing it toward the wording or structure of the example. This helped the model generate summaries that were consistent in style but not overly influenced by a single example. Figure 3.4 shows an example of a one-shot prompt we used.

```
prompt = f"""Here is an example of a bug report and its summary:

Example Bug Report:
{example_bug_report}

Example Summary:
{example_summary}

Now, Write a one-sentence summary of the core issue using no more than 10 words.
Avoid copying example text unless they naturally apply ; tailor the summary to the new bug report.\n\n

Bug Report:
{bug_report}

Summary :"""
```

Figure 3.4: An example of a one-shot prompt for bug reports

Few-shot prompting

We then extended our experiments to few-shot prompting, providing the model with multiple randomized example summaries along with the bug report. This approach

allowed the model to identify patterns and generate more accurate and coherent summaries. Figure 3.5 illustrates a few-shot prompt we designed.

```
few_shot_examples = filtered_df.sample(3, random_state=42)

example_prompt = "Here are some examples of bug reports and their summaries:\n\n"

for _, row in few_shot_examples.iterrows():
    example_prompt += f"Example Bug Report:\n{row['bug_report']}\n\n"
    example_prompt += f"Example Summary:\n{row['ground_truth_summary']}\n\n"

example_prompt += "Now, Write a one-sentence summary of the core issue using no more than 10 words.\n
Avoid copying example text unless they naturally apply; tailor the summary to the new bug report.\n\n"
```

Figure 3.5: An example of a few-shot prompt for bug reports

Chunking and Summarizing Code

When we incorporated code snippets, we observed that most exceeded the input limit of the models. To address this, we divided each code snippet into smaller chunks and prompted each of our models to summarize the chunks individually in a zero-shot setting, assuming the role of a senior software engineer. Figure 3.6 illustrates a zero-shot prompt we designed.

```
prompt = f"""You are a senior software engineer helping to analyze this buggy code.\n
Summarize this piece of buggy code in 1-2 sentences:

Buggy Code:
{chunk}

Summary:"""
```

Figure 3.6: An example of a zero-shot prompt for code chunks

We then aggregated all chunk summaries into a single summary using a zero-shot prompt, capturing the key technical aspects of the code relevant to the bug while keeping the wording concise. Figure 3.7 illustrates a zero-shot prompt we designed.

```
combined_summary_prompt = (
        "Write a summary describing the main context of the bug using minimal words in 1 sentence.\n\n"
        f"Chunk Summaries:\n{chunk_summaries_text}\n\nSummary:"
    )
```

Figure 3.7: An example of a zero-shot prompt for final code summary

Combining Bug Reports and Code

Finally, we designed prompts in zero-shot, one-shot, and few-shot settings that combined the bug report with the aggregated code summaries. By providing the models with both sources of information, we ensured that the final summary integrated the

context from the bug report with the technical details from the code, resulting in a coherent and developer-friendly output. Figures 3.8, 3.9, and 3.10 illustrate the zero-shot, one-shot, and few-shot prompts we designed.

```
final_prompt = f"""
 Now, Given a Bug Report with Buggy Code Summary, Write a one-sentence summary of the core issue using no more than 10 words.\n

Bug Report:
{bug_report}

Buggy Code Summary:
{combined_code_summary}

Final Summary:"""
```

Figure 3.8: An example of a zero-shot prompt for combined bug report and code snippet summary

```
    final_prompt = f"""Here is an example of a bug report, summarized buggy code and its summary:
    Example Bug Report:
    {example_bug_report}

    Example Buggy Code Summary:
    {example_combined_code_summary}

    Example Summary:
    {example_summary}

 Now, Given a Bug Report with Buggy Code Summary, Write a one-sentence summary of the core issue using no more than 10 words.\n
 Avoid copying example text unless they naturally apply ; tailor the summary to the new bug report.\n

Bug Report:
{bug_report}

Buggy Code Summary:
{combined_code_summary}

Final Summary:"""
```

Figure 3.9: An example of a one-shot prompt for combined bug report and code snippet summary

```
few_shot_examples = code_summary_gemma_df.sample(3, random_state=42)


example_prompt = "Here are some examples of bug reports, buggy code summaries and their summaries:\n\n"


for i, (_, row) in enumerate(few_shot_examples.iterrows(), 1):
    bug_report = row['bug_report']
    ground_truth = row['ground_truth_summary']
    combined_code_summary = row['code_summary']


    example_prompt += (
        f"Example {i}:\n"
        f"Bug Report:\n{bug_report}\n\n"
        f"Buggy Code Summary:\n{combined_code_summary}\n\n"
        f"Summary:\n{ground_truth}\n\n"
        + "="*10 + "\n\n"
    )


example_prompt = example_prompt.strip()
```

Figure 3.10: An example of a few-shot prompt for combined bug report and code snippet summary

Evaluating Prompt Effectiveness

We evaluated the prompts qualitatively by examining the coherence, relevance, and completeness of the generated summaries. Through iterative testing, as illustrated in 3.11, we found that prompts guiding the model to generate concise summaries generally performed best. These experiments confirmed that well-designed prompts can significantly improve summary quality even before fine-tuning the model.

- **Input:** Preprocessed bug reports and code snippets.
- **Output:** Prompted samples ready for model fine-tuning.

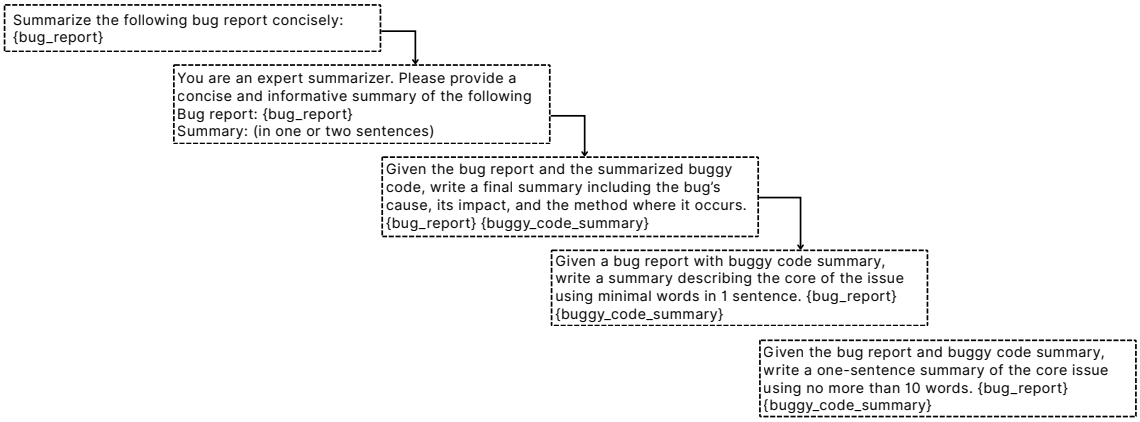


Figure 3.11: Overview of the iterative prompt refinement process. Prompts were progressively adjusted based on the quality of model-generated summaries to improve accuracy, completeness, and readability. The figure illustrates representative examples; for clarity, not all prompt permutations, including zero-, one-, and few-shot settings with bug reports and bug reports plus code, are shown.

In summary, we carefully experimented with zero-shot, one-shot, few-shot, and code chunk summarization strategies. These efforts allowed us to optimize the prompts and prepare the model to produce high-quality abstractive summaries in the subsequent stages of our methodology. We applied this entire pipeline consistently across all models to ensure comparability and to achieve high-quality outputs for each approach.

3.2.3 Fine-Tuning

Following preprocessing and prompt engineering, we extended the pipeline by fine-tuning pre-trained LLMs, adapting them specifically for abstractive bug report summarization. The goal was to make the models more context-aware, enabling them to capture the technical characteristics of bug reports and code snippets effectively. To achieve this efficiently, we employed parameter-efficient fine-tuning with Low-Rank

Adaptation (LoRA) [34], which updates only a small fraction of parameters while retaining pre-trained knowledge.

Dataset Formulation

We constructed two dataset variants for fine-tuning, allowing a direct comparison between text-only and code-augmented inputs:

- **Variant 1 (Bug Report Only):** Each input consisted of a bug report paired with its ground-truth summary.
- **Variant 2 (Bug Report + Code):** Each input combined a bug report and its corresponding code-level summary, paired with the ground-truth summary.

Training and Learning

We fine-tuned two model variants using structured prompt–target pairs to ensure consistency with the earlier prompt engineering stage.

- **Variant 1 (Bug Report Only):** Trained solely on bug reports and ground-truth summaries.
- **Variant 2 (Bug Report + Code):** Trained on bug reports combined with code summaries, paired with ground-truth summaries.

This design allowed us to compare the effectiveness of summarization when relying only on bug reports versus when integrating both bug and code contexts.

- **Input:** Bug report (and optionally code snippet summaries), formatted into structured prompts.
- **Output:** Fine-tuned models adapted for bug report summarization with or without code context.

Prediction

Once fine-tuned, both variants were applied to unseen bug reports for summarization.

- **Input:** New bug report and (optionally) code snippet summaries, structured as prompts.
- **Output:** Summaries that captured textual bug context, with Variant 2 additionally leveraging code information for more precise and technically aligned outputs.

While large pre-trained models are already capable of general text summarization, our fine-tuning process enhanced their ability to focus on the unique characteristics of software bugs. By comparing bug-only and bug+code variants, we were able to highlight the contribution of code context in producing precise, developer-friendly summaries.

3.2.4 Evaluation

To assess the generalization ability of our proposed approach, we evaluated the fine-tuned models on both the primary dataset and additional datasets that were not used in the initial analysis. This step ensured that the models were not only effective on the prepared dataset but could also handle unseen bug reports and code contexts.

The evaluation involved both fine-tuned model variants:

- **Variant 1 (Bug-Only):** Tested on bug reports without any accompanying code summaries.
- **Variant 2 (Bug + Code):** Tested on bug reports with integrated code snippet summaries.

By comparing these two configurations across unseen datasets, we aim to assess the contribution of code context to the summarization process. This evaluation framework supports the practical application of the models to new bug reports, ensuring that the methodology is capable of handling real-world scenarios where both textual and code information may be available.

3.3 Integration and Interconnections

The methodology is structured as a cohesive pipeline, where each stage builds upon the outputs of the previous component, as illustrated in Figure 3.1 and Algorithm 1. Data preprocessing transforms raw dataset URLs into a clean, well-structured dataset of bug reports and associated code snippets, ensuring that subsequent stages operate on reliable and consistent inputs.

Prompt engineering prepares the inputs for the models, using zero-shot, one-shot, and few-shot strategies to guide the models' understanding of textual bug reports. Processed code summaries are integrated alongside bug reports, allowing the models to consider both the natural language description and the relevant technical details when generating outputs. This integration ensures that the models capture the full

Algorithm 1 Chunk-and-Fuse for Abstractive Bug Report Summarization

Input: Bug report, buggy code, instructions, examples (optional), pre-trained LLM, bug report dataset, bug report + code dataset
Output: Abstractive summary, fine-tuned LLMs
Split the buggy code into smaller chunks that fit within the LLM's context window
for each code chunk **do**
    Generate a chunk-level summary using the pre-trained LLM
end for
Aggregate all chunk summaries into a single code-level summary
Combine the bug report and the code-level summary using structured prompts to produce the final abstractive summary
for each LLM **do**
    Fine-tune on the bug report dataset
    Fine-tune on the bug report + code dataset
end for
return Final abstractive summary and fine-tuned LLMs

context of each bug.

Fine-tuning adapts the models to generate context-aware summaries by training on structured prompt–target pairs. The two variants, bug-only and bug plus code, demonstrate the impact of including code context, highlighting how code summaries complement bug reports to improve clarity and technical accuracy.

Evaluation applies the models to unseen datasets, testing their ability to generalize. By comparing outputs from both variants, we can observe how the inclusion of code information enhances the informativeness and developer relevance of the summaries.

The pipeline forms an interconnected framework where the output of one component directly informs the next. For example, the quality of prompt engineering depends on the structure of the preprocessed dataset, and the quality of the final summaries depends on the integration of bug report and code information.

The overall data flow can be summarized as:

Raw dataset → Structured and cleaned dataset → Prompted bug reports with integrated code summaries → Fine-tuned models (with 2 variants) → Generated summaries

Figure 3.1 presents an architectural flowchart of the methodology, showing how each module integrates to achieve accurate and informative bug summaries, with bug reports and code summaries contributing complementary information.

3.4 Summary of the Methodology

The proposed methodology integrates three key components: data preprocessing, prompt engineering, and model fine-tuning. Preprocessing organizes and cleans raw bug reports and code snippets from dataset, creating a structured dataset suitable for model input. Prompt engineering guides the models using zero-shot, one-shot, and few-shot strategies, explicitly combining bug reports with code summaries to enhance summary quality. Fine-tuning then adapts pre-trained language models to generate coherent, developer-friendly summaries that leverage both textual and code information.

Each stage builds on the previous one, forming a cohesive pipeline where data flows seamlessly from raw input to prompted representations and finally to generated summaries. This structured approach ensures that the system produces high-quality abstractive bug summaries while maintaining consistency and reliability across different datasets.

Chapter 4

Results and Discussion

This chapter presents the experimental findings of the proposed approach and provides a critical discussion of their implications. The results demonstrate the performance of large language models (LLMs) in generating summaries for bug reports, both with and without code context, under various prompt strategies. The discussion interprets these outcomes in relation to the research objectives and compares them with existing literature, highlighting the strengths and limitations of the approach while emphasizing its practical relevance.

4.1 Datasets and Experimental Setup

4.1.1 Datasets

We evaluate our approach using four benchmark datasets that offer different perspectives. Defects4J [44] serves as the primary dataset because it provides both textual bug reports and associated code snippets, enabling evaluation of code-aware summarization through prompt engineering. In contrast, SDS [39] and ADS [40] primarily support extractive benchmark comparisons, while Fang et al.'s corpus [24] enables large-scale generalization. While all datasets include textual information such as bug descriptions, statuses, and technical metadata, only Defects4J contains complete code snippets, whereas the others include partial or none.

Defects4J [44]: A benchmark dataset containing real-world Java bugs across 17 projects, including both bug reports and corresponding code snippets. It has 854 entries in total, of which 372 include both bug reports and code. Only 133 entries from a single project contain ground-truth summaries. These 133 samples form the core of our

experiments, as they provide complete information: bug reports, code snippets, and reference summaries. On average, bug reports in this subset are 289 words long, and buggy code snippets span about 1,037 lines of code (LOC). The dataset primarily provides textual information, such as bug descriptions, statuses, comments, and technical metadata. No major class imbalance was observed, as each report was treated as an independent instance. However, since this subset was selected based on the availability of summaries, it may be biased toward better-documented bugs with richer context. This could lead to slightly optimistic performance estimates compared to real-world settings where bug reports are often less detailed.

Summary Dataset (SDS) [39]: An extractive benchmark dataset containing 36 manually annotated bug reports from four open-source projects: Mozilla, Eclipse, KDE, and Gnome. These reports contain a total of 2,361 sentences, averaging 65 sentences per report. Each report was selected to include conversational content with at least two contributors, avoiding reports dominated by long stack traces or large code snippets. Each report was annotated by three graduate students to create gold standard summaries (GSS), where sentences marked with a score of two or more were included. The GSS serves as the reference for evaluating generated summaries.

Authorship Dataset (ADS) [40]: An extractive benchmark dataset containing 96 manually annotated bug reports from the same four projects as SDS. Reports are shorter than in SDS, averaging 39 sentences, and include diverse authorship, enabling evaluation on more concise inputs and varied writing styles. Each summary was annotated similarly to SDS.

Public Bug Report Corpus by Fang et al. [24]: A large-scale corpus of 275,639 bug reports from four open-source projects: Mozilla, Eclipse, NetBeans, and GCC. The source reports an average length of 171 per report, but the unit is unspecified. For the purpose of our study, we did not utilize the entire corpus. Instead, we randomly sampled 500 bug reports from this dataset to create a manageable yet representative subset for experimentation. While random selection helps reduce systematic bias, the smaller sample size may not fully capture the diversity of the original dataset in terms of bug complexity, language style, and project characteristics. This introduces a potential sampling bias, which should be considered when generalizing the findings.

4.1.2 Experimental Setup

Research Questions

We first compare our proposed approach with state-of-the-art bug report summarization techniques to evaluate its effectiveness (RQ1). Next, we perform an ablation study to investigate the impact of integrating full code snippets with bug reports (RQ2). Finally, we examine the effect of prompt engineering compared to fine-tuning on the quality of summarization across different LLMs and datasets (RQ3).

- **RQ1:** How effective is our approach compared to state-of-the-art bug report summarization approaches?
- **RQ2:** How does the integration of full code snippets influence the quality of abstractive bug report summaries?
- **RQ3:** How do prompt engineering and fine-tuning approaches affect summarization performance across different LLMs and datasets?

Models

We selected eight large language models (LLMs) representing diverse architectures, parameter scales, and specialization levels to evaluate their performance in bug report summarization, both with and without code snippets. In selecting these models, we aimed to balance general-purpose reasoning capabilities with code-aware understanding.

CodeLlama [78]: It is a generative text model designed for code understanding and generation tasks. We used the 7B-Instruct-hf version, which contains 6.74 billion parameters. We included it to evaluate how strong code comprehension aids in generating bug report summaries including code context.

Llama-3.1 [28]: It is a general-purpose LLM designed for multilingual and reasoning tasks. We used the 8B-Instruct version, which contains 8.03 billion parameters. We used it as a baseline for general language understanding to see whether pure language models can compete in structured technical summarization tasks.

Mistral [37]: We used the 7B-Instruct-v0.3 version, which contains 7.25 billion parameters. We selected it to explore how an instruction-optimized model performs on bug report summarization across various prompt settings.

Phi-3 [1]: It is a small-scale model designed for lightweight deployments, with competitive reasoning and instruction-following abilities given its size. We used the mini-

4k-Instruct version, which contains 3.82 billion parameters. We included it to assess whether small models can generate usable summaries under resource constraints.

Gemma [85]: It is a lightweight model built on Google's Gemini technology. We used the 7B-Instruct version, which contains 8.54 billion parameters. We selected it for its strong performance on a variety of text generation tasks, including summarization.

Qwen3 [94]: It is a non-thinking model, meaning it does not have self-reflective or agentic capabilities, but is designed for improved reasoning, text comprehension, coding, and long-context understanding. We used the 4B-Instruct-2507 version, which contains 4.02 billion parameters. We included it because its balanced design for reasoning and coding contexts is relevant to our multi-modal summarization experiments.

DeepSeek Coder [29]: It is a fine-tuned language model with coding capabilities. We used the 6B-Instruct version, which contains 6.74 billion parameters. We included it to analyze whether models explicitly trained for code tasks help us outperform general-purpose models in bug report summarization when code context is available.

GPT-3.5 Turbo [69]: It is a proprietary OpenAI model, optimized for fast inference and robust reasoning at scale. Although its parameter count is undisclosed, it represents a strong commercial baseline for natural language tasks. We included it as a benchmark to compare open-weight models against widely used proprietary solutions.

Experimental Configurations

We conducted our experiments on a cloud-based GPU rented via RunPod, using a single NVIDIA RTX A6000 GPU with 48 GB VRAM, 50 GB system memory, and 9 vCPUs. The environment was set up in a Docker container, running PyTorch version 2.8.0, Python version 3.11, CUDA version 12.8.1, cuDNN version 90800, on Ubuntu version 22.04, providing a ready-to-use development environment with JupyterLab. We loaded all models in 8-bit precision using bitsandbytes (v0.47.0) to reduce memory usage. We used Transformers version 4.55.4 and PEFT version 0.17.1 for fine-tuning.

We ensured consistent input formatting for all models, using a default placeholder token where necessary. For LLaMA-3, we applied special handling to account for its unique token configuration. We did not perform any additional preprocessing. Inputs were tokenized with truncation and padding, using maximum lengths of up to 2,048 tokens for both bug reports alone and combined bug reports with code. For large code snippets, we applied a chunking approach (maximum 1,024 tokens per chunk) and

combined individual chunk summaries into a final summary. We applied zero-shot, one-shot, and few-shot prompt settings across all models. Prompts were manually designed and refined iteratively for optimal coverage and clarity. During text generation, we used beam search (beam size = 3), top-k sampling (k = 50), early stopping, and constraints to prevent repeated sequences (n = 2). Maximum generation length was set to 512 tokens.

For fine-tuning, we split the Defects4J [44] dataset into 90% for training and 10% for validation. We conducted training with a batch size of 1 and gradient accumulation of 4, using a learning rate of 5×10^{-5} for two epochs with FP16 precision.

4.2 Presenting the Results

We report results systematically using quantitative evaluation metrics, using ROUGE [58] to evaluate baselines (both extractive and abstractive approaches) and BERTScore [99] to assess semantic similarity between generated summaries and ground-truth summaries. Since our task focuses on abstractive summarization, BERTScore is particularly appropriate, as it captures semantic similarity beyond surface-level overlap, and we report Precision, Recall, and F1. In contrast, ROUGE relies on exact n-gram matches, favoring lexical similarity but often failing to capture true meaning equivalence in abstractive methods that may express the same idea differently [10]. Nevertheless, for consistency with prior work, we also report ROUGE-N and ROUGE-L as familiar baselines for evaluating relative performance. The results are presented in multiple tables, each corresponding to a distinct experimental setup, with all tables clearly labeled and referenced in the text for clarity.

Table 4.1 presents a baseline comparison of ROUGE-1, ROUGE-2, and ROUGE-L scores on the Fang et al. corpus [24], where models generate summaries using only bug reports. This comparison includes prior state-of-the-art approaches alongside various LLMs, most fine-tuned on Defects4J [44] and others evaluated under zero-shot prompting.

Table 4.2 reports BERTScore performance for multiple LLMs under different prompting strategies (zero-shot, one-shot, few-shot). Results are shown both for bug reports alone and for bug reports combined with buggy code, highlighting the effect of code context.

To further examine the role of code input, we conducted an ablation study. Table 4.3 presents results when only buggy code is provided as input, while Table 4.4 considers

Table 4.1: Comparison of ROUGE scores (ROUGE-1, ROUGE-2, ROUGE-L) for baseline models and LLMs on the Fang et al. corpus dataset using solely bug reports as input. All LLMs were fine-tuned on the Defects4J dataset except GPT-3.5 Turbo, which was evaluated in a zero-shot prompt setting. Scores reflect performance on abstractive bug report summarization.

Model	ROUGE-1	ROUGE-2	ROUGE-L
SUMLLAMA [92]	0.4426	0.2515	0.4101
KSCLP [81]	0.4133	0.2202	0.3789
RTA [24]	0.3919	0.2057	0.3597
BugSum [59]	0.2591	0.1166	0.2469
DeepSum [56]	0.1760	0.0805	0.1700
CodeLlama	0.2641	0.0746	0.2206
Llama-3.1	0.1220	0.0293	0.0998
Mistral	**0.2786**	**0.0747**	**0.2316**
Phi-3	0.1246	0.0253	0.0988
Gemma	0.2016	0.0425	0.1596
Qwen3	0.0960	0.0189	0.0764
DeepSeek Coder	0.1049	0.0202	0.0821
GPT-3.5 Turbo	0.3225	0.1146	0.2731

the use of patch code alongside bug reports. The effect of input order is summarized in Table 4.5, where we compare “Code -> Bug Report” versus “Bug Report -> Code” sequences. CodeLlama was specifically evaluated in this ablation study.

Finally, Table 4.6 reports cross-dataset generalization results, comparing models fine-tuned on Defects4J [44] and evaluated on multiple datasets. These experiments assess both bug report–only inputs and combined bug report + code settings, showing the influence of fine-tuning and code context on abstractive summarization performance.

Overall, the results are presented to enable structured comparison across models, prompting strategies, datasets, and input configurations. At this stage, the emphasis is on reporting the data rather than interpreting trends, which will be discussed in the following section.

4.3 Interpreting the Results

4.3.1 Effectiveness of Our Approach Compared to Baseline Approaches (RQ1)

To evaluate the effectiveness of our approach, we compare the bug-report-only component of our chunk-and-fuse framework against extractive and abstractive baselines for

Table 4.2: BERTScore performance (Precision, Recall, F1) on the Defects4J dataset across different prompting strategies (Zero-Shot, One-Shot, Few-Shot) for each LLM. Results are reported for bug reports alone and for bug reports combined with corresponding buggy code, highlighting the effect of including code context.

Model	Prompting Technique	Bug Reports			Bug Reports + Buggy Code		
		Precision	Recall	F1 Score	Precision	Recall	F1 Score
CodeLlama	Zero-Shot	0.5553	0.6408	0.5859	0.4957	0.5350	0.5109
	One-Shot	0.4194	0.5347	0.4618	0.5656	0.5627	0.5615
	Few-Shot	0.6711	0.6880	0.6752	0.6326	0.6398	0.6337
Llama-3.1	Zero-Shot	0.4154	0.6015	0.4895	0.3765	0.5742	0.4509
	One-Shot	0.3955	0.5995	0.4745	0.3815	0.5760	0.4561
	Few-Shot	0.4005	0.6050	0.4795	0.5673	0.6546	0.5974
Mistral	Zero-Shot	0.6574	0.6924	0.6719	0.6104	0.6212	0.6137
	One-Shot	0.6783	0.6895	0.6816	0.6315	0.6082	0.6180
	Few-Shot	0.6874	0.7032	0.6928	0.6593	0.6467	0.6509
Phi-3	Zero-Shot	0.4311	0.6058	0.4992	0.4965	0.5797	0.5304
	One-Shot	0.4220	0.6299	0.5027	0.4758	0.5209	0.4925
	Few-Shot	0.4506	0.6176	0.5171	0.4314	0.5767	0.4905
Gemma	Zero-Shot	0.5662	0.6756	0.6132	0.5271	0.6307	0.5715
	One-Shot	0.4154	0.5862	0.4844	0.5335	0.6054	0.5630
	Few-Shot	0.5459	0.6463	0.5894	0.5477	0.6161	0.5768
Qwen3	Zero-Shot	0.4085	0.5992	0.4846	0.4159	0.5525	0.4709
	One-Shot	0.4135	0.6005	0.4878	0.4202	0.5447	0.4705
	Few-Shot	0.4059	0.5995	0.4829	0.4102	0.5583	0.4690
DeepSeek Coder	Zero-Shot	0.4132	0.6401	0.5002	0.4080	0.5871	0.4737
	One-Shot	0.4050	0.6308	0.4902	0.4389	0.5880	0.4966
	Few-Shot	0.4132	0.6401	0.5002	0.4107	0.5987	0.4789
GPT-3.5 Turbo	Zero-Shot	0.7478	0.7869	0.7640	0.6761	0.6763	0.6740
	One-Shot	0.8398	0.8417	0.8388	0.7893	0.7787	0.7820
	Few-Shot	**0.8696**	**0.8536**	**0.8595**	**0.9108**	**0.8929**	**0.9003**

a fair comparison. Prior studies on bug report summarization largely report ROUGE-based evaluations [24], [56], [59], [81], [92], which tend to favor extractive methods [10]. We adopt ROUGE scores for consistency with previous work, while noting that these metrics may not fully capture the semantic quality of abstractive summaries.

As shown in Table 4.1, extractive baselines underperform on the Fang et al. corpus [24], with BugSum [59] achieving ROUGE-1 of 0.2591 and DeepSum [56] only 0.1760, as these models rely on directly selecting sentences from the original bug reports, which prevents them from producing the concise and paraphrased summaries that the Fang dataset favors. Notably, our models, fine-tuned on solely bug reports, such as Mistral and CodeLlama achieve ROUGE-1 scores of 0.2786 and 0.2641, respectively, outperforming BugSum, while Gemma also surpasses DeepSum. Therefore, our ap-

Table 4.3: BERTScore performance (Precision, Recall, F1) on the Defects4J dataset across Zero-Shot prompting strategy using CodeLlama with solely buggy code as input. Results illustrate the model's ability to generate abstractive summaries without accompanying bug report text, highlighting the context provided by code alone.

Prompting Technique	Buggy Code		
	Precision	**Recall**	**F1 Score**
Zero-Shot	0.4570	0.5005	0.4767

Table 4.4: BERTScore performance (Precision, Recall, F1) on the Defects4J dataset across different prompting strategies (Zero-Shot, One-Shot, Few-Shot) using CodeLlama with bug reports and corresponding patch code. This setup highlights the impact of incorporating only the corrected portions of code alongside textual bug reports for abstractive summarization.

Prompting Technique	Bug Reports + Patch Code		
	Precision	**Recall**	**F1 Score**
Zero-Shot	0.5903	0.5940	0.5888
One-Shot	0.5613	0.5630	0.5592
Few-Shot	**0.6286**	**0.6499**	**0.6368**

proach is able to produce more concise, paraphrased, and contextually coherent summaries compared to extractive baselines.

Abstractive baselines such as SUMLLAMA [92], KSCLP [81], and RTA [24] report higher ROUGE-1 scores (0.4426, 0.4133, 0.3919) and ROUGE-L scores (0.4101, 0.3789, 0.3597) on the Fang corpus as they were fine-tuned directly on this dataset. In contrast, the bug-report-only component of our approach is fine-tuned on Defects4J [44] and evaluated across multiple datasets, including a sampled subset of 500 bug reports from Fang, without dataset-specific adaptation. This difference explains our lower ROUGE-1 and ROUGE-L scores on Fang, though GPT-3.5 Turbo achieved ROUGE-1

Table 4.5: BERTScore performance (Precision, Recall, F1) on the Defects4J dataset across different prompting strategies (Zero-Shot, One-Shot, and Few-Shot) using CodeLlama, comparing different input sequences: Code -> Bug Report versus Bug Report -> Code. Results are reported, highlighting the impact of input order on abstractive summarization performance.

Prompting Technique	Code -> Bug Report			Bug Report -> Code		
	Precision	**Recall**	**F1 Score**	**Precision**	**Recall**	**F1 Score**
Zero-Shot	**0.6029**	0.6087	0.6030	0.4957	0.5350	0.5109
One-Shot	0.5315	0.5667	0.5450	0.5656	0.5627	0.5615
Few-Shot	0.5992	**0.6230**	**0.6087**	**0.6326**	**0.6398**	**0.6337**

score of 0.3225 despite being used only in a zero-shot prompt setting without fine-tuning. Crucially, while the baselines benefited from fine-tuning on Fang, our models maintain strong semantic performance, even without fine-tuning, as reflected in Table 4.6. Moreover, prior baselines did not report semantic measures such as BERTScore, further limiting direct comparison and reinforcing the need to evaluate beyond token-level overlap when assessing abstractive summaries.

Answer to RQ1: The bug-report-only component of our chunk-and-fuse approach generates concise, coherent summaries that outperform extractive baselines and achieve strong semantic performance (BERTScore), making it competitive with fine-tuned abstractive models. Despite lower ROGUE score on the sampled Fang et al. subset, the strong BERTScore demonstrates the effectiveness of our multi-stage summarization framework.

4.3.2 Impact of Code Context (RQ2)

To assess the influence of full code snippets on abstractive bug report summarization, we evaluate eight LLMs on the Defects4J [44] dataset using both bug reports only and bug reports combined with code snippets. Table 4.2 shows that adding code sometimes improves few-shot performance, though in several cases the scores are slightly lower than using only bug reports. This outcome contrasts with our original hypothesis that including code would enhance the model's understanding of the bug context and produce better summaries. This deviation is influenced by both the characteristics of the dataset and the behavior of the models.

The ground truth summaries in Defects4J present unique challenges: many are extremely short and abstract, while others follow inconsistent styles, emphasizing either high-level concepts or specific code constructs and trigger conditions. This variability makes it difficult for models to learn consistent summarization patterns. Additionally, summaries generated with code context occasionally include relevant but unmatched new content, which may be penalized despite being valid. These factors suggest that, while code can enhance context, current evaluation metrics may not fully capture its benefits in abstractive summarization.

With the exception of GPT-3.5 Turbo, Mistral, CodeLlama, and Gemma, other conversational models tend to produce verbose, assistant-like outputs, which reduces overlap with the ground truth summaries and lowers evaluation scores. In few-shot settings, GPT-3.5 Turbo achieves the highest F1 scores, reaching 0.9003 for bug reports combined with code and 0.8595 for bug reports only, largely due to its outputs

closely matching the ground truth. However, these high scores do not necessarily reflect qualitatively superior summaries, as the ground truth in Defects4J is often short and inconsistent; thus, closely matching it does not always equate to better or more informative summaries. In comparison, Mistral maintains a more balanced behavior, with few-shot F1 scores of 0.6928 with bug reports only and 0.6509 with bug reports and code, likely due to its instruction-tuned nature that encourages concise yet accurate summarization. CodeLlama also balances accuracy and instruction adherence, showing few-shot F1 scores of 0.6752 with bug reports and 0.6337 with bug reports and code, with only a moderate dip when code is added. Gemma, despite being a smaller-scale model with fewer parameters, exhibits strong performance, with few-shot F1 decreasing only marginally from 0.5894 with bug reports to 0.5768 with bug reports and code. Notably, Llama3's F1 score improves from 0.4795 to 0.5974 with few-shot prompting, likely because providing additional examples encourages more concise outputs, reducing verbosity. DeepSeek, with few-shot F1 scores of 0.5002 for bug reports only and 0.4789 for bug reports plus code, demonstrates that although it is optimized for code understanding, its conversational style and focus on generating explanations from code context do not align well with the short and abstract summaries in Defects4J.

To further analyze the role of code in summarization, we perform ablation studies on CodeLlama.

Buggy code only: As illustrated in Table 4.3, when provided solely with buggy code, CodeLlama achieves a few-shot F1 score of 0.4767. This is lower than using bug reports alone (0.6752) or in combination with code (0.6337), indicating that code snippets by themselves provide limited context for generating abstractive summaries. While some information is captured, the absence of descriptive bug report text reduces the model's ability to produce complete and accurate summaries.

Patch code: Patch code refers to only the corrected or modified portions of the buggy code, highlighting the specific changes made to fix the issue. As illustrated in Table 4.4, incorporating patch code along with bug reports slightly improves performance compared to buggy code alone, achieving a few-shot F1 score of 0.6368. This suggests that patch code provides additional context about the intended fix, helping CodeLlama better infer the nature of the bug and the summary content. Compared to the standard bug report and buggy code setting (F1 = 0.6337), the difference is modest, indicating that while patch code is informative, the descriptive bug report remains the primary source of abstractive summaries.

Input sequence: As illustrated in Table 4.5, we also evaluated the impact of the in-

put sequence, testing Code Bug Report versus Bug Report - > Code for CodeLlama. In few-shot settings, Code -> Bug Report achieves an F1 score of 0.6087, while Bug Report -> Code improves to 0.6337, suggesting that presenting the bug report first allows the model to prioritize descriptive context and better integrate code information. However, in zero-shot settings, reversing the sequence increases the F1 from 0.5109 to 0.6030, demonstrating that input order can substantially influence summarization performance even without examples.

Answer to RQ2: Integration of code snippets provides modest gains (highest F1 = 0.9003). Ablation studies with CodeLlama show that buggy code alone performs poorly (F1 = 0.4767), patch code with bug reports slightly improves performance (F1 = 0.6368), and input order further affects results. Overall, model behavior, influenced by pre-training, instruction tuning, scale, and sensitivity to code context, along with dataset characteristics, determines the effectiveness of code integration, though current evaluation metrics may not fully capture qualitative improvements in abstractive summaries.

4.3.3 Comparison of the Effects of Prompt Engineering and Fine Tuning (RQ3)

To investigate the effect of prompt engineering versus fine-tuning on abstractive bug report summarization, we evaluated multiple LLMs across four datasets. GPT-3.5 Turbo cannot be fine-tuned, so it was evaluated directly in zero-shot settings using prompt engineering. The other models were fine-tuned on bug reports and bug reports combined with code, as shown in Table 4.6.

Notably, on SDS [39] and ADS [40], which contain extractive ground truth summaries, fine-tuned models with both bug reports only and bug reports combined with code achieve moderate precision (45–60%) but low recall (35–45%). This means the generated summaries correctly include some relevant words from the reference but fail to reproduce many exact phrases, because the ground truth is highly detailed and extractive, while the models generate more abstractive summaries. In both datasets, the difference in performance between using only bug reports and adding code is marginal, indicating that code provides limited additional benefit for these extractive summaries.

In Fang's Corpus [24], which contains abstractive and relatively concise summaries, GPT-3.5 Turbo achieves the highest F1 in zero-shot prompt engineering (0.6394). Among the fine-tuned models, Mistral reaches 0.6202 with bug reports only and 0.6170 with

bug reports and code, CodeLlama achieves 0.5991 and 0.6010, and Gemma reaches 0.5779 and 0.5864. These results indicate that fine-tuned models on bug reports and code provide only marginal gains. Llama3, Qwen3, and DeepSeek exhibit lower precision despite moderate recall (55–58%), reflecting that they capture relevant content but generate more extraneous or less accurate words and phrases due to their conversational focus. Overall, models with a more conversational or verbose style tend to struggle with precision, whereas models that are less prone to verbosity better balance precision and recall on abstractive datasets.

On Defects4J [44], which presents short and abstract ground truth summaries, fine-tuned models tend to perform worse when incorporating bug reports and code compared to using only bug reports. CodeLlama achieves an F1 of 0.6767 with bug reports only but drops to 0.5308 when code is added, and similar patterns are observed for Llama3 (0.4816 to 0.4588), Phi3 (0.4849 to 0.4728), and DeepSeek (0.4958 to 0.4933). This indicates that adding code can sometimes introduce noise or irrelevant details that reduce overlap with the concise references. In contrast, as illustrated in Table 4.2, zero-shot prompt engineering shows a smaller performance drop or even slight improvements in some models: GPT-3.5 Turbo achieves the highest F1 (0.7640 with bug reports only, 0.6740 with code), while Mistral, Gemma, and Phi3 show more stable performance compared to their fine-tuned counterparts. This effect is likely exacerbated by the small size of the dataset (133 entries), so we prioritized generalizing across datasets rather than optimizing scores specifically on Defects4J. Overall, Defects4J demonstrates that fine-tuning on code-augmented inputs does not always guarantee better summarization and that zero-shot prompt engineering can be competitive, particularly for models already strong in understanding bug report context.

Answer to RQ3: Fine-tuning yields moderate performance on extractive datasets but provides only marginal gains when code is added. On abstractive datasets, prompt engineering often rivals or surpasses fine-tuned models. The relatively small size of Defects4J likely limits fine-tuning effectiveness, contributing to its weaker results compared to prompt engineering.

4.4 Challenges and Limitations

This study has several limitations that warrant careful consideration. First, fine-tuning on the relatively small subset of 133 Defects4J [44] entries necessitates a trade-off between benchmark performance and generalization. Consequently, the generated summaries may not fully reflect the diversity and complexity of real-world bug re-

ports. Second, the scarcity of datasets containing both bug reports and corresponding buggy code constrains the evaluation of our approach and limits its broader applicability. At the same time, generating such datasets is time-consuming, labor-intensive, and prone to bias, as summaries often reflect the subjective views of annotators, which can in turn affect evaluations due to variations in human judgment based on experience, fatigue, or interpretation. Third, aggregating code-level summaries into a single report-level summary can result in content loss, highlighting the potential of progressive prompting as a promising alternative. Additionally, prompt engineering remains iterative; although our prompts were refined over multiple iterations, there is no guarantee that they are fully optimal.

The evaluation of abstractive summaries introduces further challenges. Existing automated metrics, such as ROUGE and BERTScore, have limitations: ROUGE primarily measures lexical overlap, while BERTScore can penalize unmatched but semantically relevant content, meaning that both metrics may fail to fully capture the true semantic quality, adequacy, or usefulness of generated summaries [61], [77]. Evaluating long buggy code snippets is especially difficult due to model context limitations, and the resource-intensive nature of large language models demands significant computational resources.

4.5 Future Work

Several directions for future work emerge from this study. Human evaluation can complement automated metrics by providing deeper insights into the quality and practical usefulness of generated summaries. Developing a summarization framework or tool integrated into real-world debugging workflows could enhance practical impact and directly assist developers. There is also clear potential for designing evaluation metrics specifically tailored to bug report analysis, going beyond existing general-purpose measures [61]. Finally, constructing larger and more diverse bug report–code datasets would further support model training, evaluation, and generalization.

4.6 Implications and Significance of Findings

The findings of this study have several important implications for both research and practice in bug report summarization.

First, the results demonstrate that multi-stage abstractive summarization frameworks,

such as our chunk-and-fuse approach, can generate concise, coherent, and semantically meaningful summaries that outperform extractive baselines and remain competitive with fine-tuned abstractive models. This suggests that, in some cases, effective summarization can be achieved without dataset-specific fine-tuning, indicating potential for applying such frameworks to other software projects and datasets.

Second, while integrating code snippets can provide additional context, our analysis shows that the benefits are often modest and depend heavily on model behavior, dataset characteristics, and input presentation. Ablation studies with CodeLlama reveal that buggy code alone provides limited context, patch code offers incremental improvements, and input sequence affects performance. This insight underscores the importance of carefully considering how code context is incorporated into abstractive summarization pipelines, particularly when evaluation metrics may not fully capture qualitative improvements.

Third, the comparison between prompt engineering and fine-tuning highlights that zero-shot or few-shot prompt strategies can rival or even surpass fine-tuned models in certain settings, especially when datasets are small or summaries are concise and abstractive. This has practical significance for real-world deployment, as it reduces the reliance on resource-intensive fine-tuning and enables faster adaptation to new projects or bug report corpora.

Finally, the study emphasizes critical limitations and opportunities for the field: existing metrics may not fully capture semantic adequacy or usefulness; high-quality, diverse bug report–code datasets are scarce; and human evaluation remains necessary to validate model outputs. Addressing these challenges can lead to more robust evaluation frameworks, improved models, and practical tools that assist developers in debugging workflows.

Overall, our findings contribute to a deeper understanding of the interplay between bug report text, code context, and model capabilities, and they provide guidance for future research aiming to enhance abstractive bug report summarization in both academic and applied settings.

4.7 Conclusion of the Chapter

This chapter evaluated our chunk-and-fuse framework for abstractive bug report summarization, analyzing the contributions of bug reports, code context, prompt engineering, and fine-tuning. The bug-report-only component generates concise, coher-

ent, and semantically rich summaries, outperforming extractive baselines and remaining competitive with fine-tuned abstractive models. Code provides modest gains, with ablation studies on CodeLlama showing that buggy code alone is insufficient, patch code offers slight improvements, and input order can influence results.

Fine-tuning on small datasets like Defects4J [44] yields limited benefits, while prompt engineering often matches or surpasses fine-tuned models. Existing evaluation metrics provide useful insights but may not fully capture semantic quality or practical usefulness, highlighting the need for human evaluation and task-specific measures.

Overall, these findings suggest that effective abstractive summarization can be achieved with minimal dataset-specific adaptation, offering insights for integrating summarization frameworks into real-world debugging workflows and guiding future research on specialized metrics and larger, more diverse bug report–code datasets.

Table 4.6: BERTScore performance (Precision, Recall, F1) of LLMs on multiple datasets, comparing bug reports alone versus bug reports combined with corresponding buggy code. All LLMs were fine-tuned on the Defects4J dataset using a prompt with the bug report, and optionally the associated code summaries, and then evaluated across all datasets, except GPT-3.5 Turbo, which was only evaluated in a zero-shot prompt setting. This table highlights the effect of fine-tuning and the impact of including code context for abstractive bug report summarization.

Dataset	**Model**	**Bug Reports**			**Bug Reports + Buggy Code**		
		Precision	**Recall**	**F1 Score**	**Precision**	**Recall**	**F1 Score**
Defects4j [44]	CodeLlama	0.6608	0.7002	0.6767	0.5110	0.5650	0.5308
	LLama-3.1	0.4025	0.6064	0.4816	0.3812	0.5886	0.4588
	Mistral	**0.6764**	**0.7072**	**0.6877**	**0.5944**	**0.6220**	**0.6051**
	Phi-3	0.4056	0.6140	0.4849	0.4109	0.5665	0.4728
	Gemma	0.5726	0.6756	0.6173	0.5932	0.6463	0.6160
	Qwen3	0.4006	0.5868	0.4744	0.3955	0.5707	0.4658
	DeepSeek Coder	0.4075	0.6416	0.4958	0.4109	0.6277	0.4933
	GPT-3.5 Turbo	0.7478	0.7869	0.7640	-	-	-
SDS [39]	CodeLlama	0.6072	0.3753	0.4620	0.6049	0.3582	0.4476
	LLama-3.1	0.4969	0.4337	0.4624	0.4947	0.4148	0.4503
	Mistral	**0.5983**	0.3539	0.4435	0.6133	0.3662	**0.4575**
	Phi-3	0.5279	0.4001	0.4520	0.5131	0.4150	0.4563
	Gemma	0.5821	0.3968	0.4702	0.5871	0.3893	0.4661
	Qwen3	**0.5532**	**0.4685**	**0.5070**	0.5468	0.4673	0.5036
	DeepSeek Coder	0.5124	0.4394	0.4721	0.5187	0.4442	0.4778
	GPT-3.5 Turbo	0.6354	0.3745	0.4703	-	-	-
ADS [40]	CodeLlama	0.6111	0.4054	0.4856	0.6141	0.3954	0.4795
	LLama-3.1	0.4842	0.4405	0.4599	0.4856	0.4406	0.4607
	Mistral	0.6176	0.3977	0.4827	**0.6294**	0.4116	**0.4967**
	Phi-3	0.5165	0.4510	0.4797	0.5075	0.4477	0.4732
	Gemma	0.5913	0.4408	0.5041	0.5971	0.4326	0.5004
	Qwen3	**0.5327**	**0.4884**	**0.5088**	0.5351	**0.4908**	0.5113
	DeepSeek Coder	0.5221	0.4853	0.5016	0.5081	0.4778	0.4910
	GPT-3.5 Turbo	0.6536	0.4202	0.5104	-	-	-
Fang's Corpus [24]	CodeLlama	0.6148	0.5912	0.5991	0.6185	0.5913	0.6010
	LLama-3.1	0.3846	0.5452	0.4476	0.3911	0.5507	0.4542
	Mistral	**0.6297**	**0.6167**	**0.6202**	0.6210	**0.6195**	0.6170
	Phi-3	0.4249	0.5645	0.4816	0.4170	0.5673	0.4773
	Gemma	0.5581	0.6063	0.5779	0.5746	0.6067	0.5864
	Qwen3	0.3946	0.5484	0.4571	0.3949	0.5485	0.4576
	DeepSeek Coder	0.3959	0.5801	0.4677	0.3898	0.5749	0.4614
	GPT-3.5 Turbo	0.6519	0.6330	0.6394	-	-	-

Chapter 5

Conclusion

5.1 Restating Research Objectives and Questions

To conclude, it is useful to briefly revisit the objectives and research questions that guided this thesis. Our work set out to design a summarization approach that combines bug report text with relevant code snippets, applies abstractive techniques through large language models, and leverages prompt engineering to produce concise and contextually meaningful summaries.

The study was driven by three main research questions:

- How effective is our approach compared to existing bug report summarization methods?
- What impact does integrating full code snippets have on the quality of summaries?
- How does prompt engineering compare to fine-tuning across different LLMs in terms of summarization performance?

By restating these aims and questions here, we bring the focus back to the foundation of this work, which structured both our methodological choices and evaluation strategy.

5.2 Summary of Key Findings

Our study produced the following key findings, organized around the research questions:

- **RQ1:** The proposed chunk-and-fuse approach outperformed extractive baselines and achieved strong semantic performance, though fine-tuned abstractive baselines reported higher ROUGE scores due to dataset-specific fine-tuning.
- **RQ2:** Incorporating code provided only modest gains, with benefits depending on whether buggy or patch code was used and on the order of integration with the bug report.
- **RQ3:** Prompt engineering proved highly effective, often rivaling or exceeding fine-tuned models, particularly on smaller datasets where fine-tuning offered limited advantage.

Overall, the results demonstrate that combining bug reports with selective code context and using prompt-based LLMs can yield meaningful abstractive summaries, while also underscoring the importance of better datasets and evaluation metrics for this task.

5.3 Discussion of Implications

The findings of this study carry several important implications for both research and practice.

First, they demonstrate that large language models, guided through prompt engineering, can produce abstractive summaries of bug reports that are coherent, semantically rich, and useful for developers, even without dataset-specific fine-tuning. This highlights the potential of prompt-based methods as a lightweight and flexible alternative to traditional fine-tuned approaches, particularly when annotated data is scarce.

Second, the limited but measurable impact of integrating buggy or patch code suggests that while textual bug reports remain the primary source of summarization value, code context can add nuance when incorporated selectively. This finding underscores the need for future research to explore more sophisticated ways of aligning natural language bug reports with source code, rather than treating them as parallel but independent inputs.

Third, the study contributes methodologically by systematically evaluating abstractive summarization in the bug report domain using both lexical (ROUGE) [63] and semantic (BERTScore) [99] metrics. The divergence between these metrics emphasizes the importance of moving beyond surface-level overlap toward semantic adequacy in evaluation, and points to the need for domain-specific evaluation frameworks tailored to software engineering tasks.

Finally, by situating the results within the broader discourse on software maintenance and developer productivity, this research highlights the practical potential of automated summarization to reduce cognitive load and accelerate debugging workflows. At the same time, the limitations identified particularly around dataset availability and evaluation call for further community efforts to build richer corpora and more robust benchmarks for this emerging area.

5.4 Contributions of the Research

This research has made several important contributions to the field of bug report summarization:

- It demonstrates the value of integrating code context, specifically buggy code, with textual bug reports to improve the semantic quality of generated summaries.
- It highlights the effectiveness of prompt engineering in guiding large language models to produce concise and contextually rich abstractive summaries.
- It provides empirical evidence using BERTScore, demonstrating that the proposed approach achieves strong semantic similarity and generates coherent, contextually accurate summaries, highlighting its effectiveness for developer-relevant abstractive summarization.

Overall, these contributions advance both the methodological and practical aspects of automated bug report summarization, offering insights for future research and development in developer-assistive tools.

5.5 Acknowledging Limitations

This study has several limitations that should be considered when interpreting the results. First, the small size of the Defects4J [44] subset (133 entries) constrains generalization, and the generated summaries may not fully capture the diversity of real-world bug reports. Second, the scarcity of high-quality datasets containing both bug reports and corresponding buggy code limits evaluation and broader applicability. Creating such datasets is also time-consuming, labor-intensive, and prone to annotation bias, which can affect human evaluations. Third, aggregating code-level summaries into a single report-level summary may result in information loss, while prompt engineering remains iterative and may not be fully optimized.

Evaluation of abstractive summaries introduces additional challenges. Automated

metrics, including BERTScore, provide valuable semantic insights but may penalize valid content that does not directly match the reference [61]. Long code snippets remain difficult to summarize due to model context limitations, and resource requirements for large language models can be substantial. Acknowledging these limitations provides context for the findings and highlights areas for future research and methodological improvement.

5.6 Suggestions for Future Research

Building on the findings and limitations of this study, several directions for future research are apparent. Human evaluation could complement automated metrics to provide deeper insights into the practical usefulness of generated summaries. Developing integrated summarization tools for real-world debugging workflows, designing evaluation metrics tailored to bug report analysis, and creating larger, more diverse bug report–code datasets would further enhance model performance, generalization, and practical applicability.

5.7 Final Reflections

This study underscores the significance and broader impact of automated abstractive bug report summarization, highlighting its potential to improve developer productivity and reduce cognitive load in debugging workflows. The research journey has been both intellectually enriching and practically challenging. A significant constraint was the scarcity of computational resources, which necessitated renting external GPU services at team expense to conduct experiments. Managing these limitations required careful planning, resourcefulness, and persistence. These experiences not only strengthened our understanding of large-scale abstractive summarization and the integration of bug reports with code but also highlighted the practical challenges of conducting research under resource constraints. Overall, the study underscores both the potential of automated bug report summarization and the problem-solving skills needed to advance such work in real-world conditions.

5.8 Concluding Remarks

This thesis presents a hierarchical, multi-stage pipeline that leverages LLMs to generate abstractive bug report summaries by progressively integrating buggy code with

textual reports. Our experiments show that this approach achieves strong semantic performance, as measured by BERTScore, and outperforms extractive baselines, while prompt-engineered LLMs can, in some cases, rival fine-tuned models. Including full buggy code can provide additional context that enhances summary quality, demonstrating the potential of code-aware abstractive summarization.

Despite the promising results, limitations such as small code-aware datasets, variability in ground-truth summaries, and evaluation metric constraints suggest caution in interpreting quantitative gains. Future research should explore larger, more diverse bug report–code datasets, tailored evaluation methods, and human-centered studies to assess practical impact. Overall, this work lays a foundation for bridging natural-language bug reports and source code, aiming to improve software maintenance and developer productivity.

References

[1] M. Abdin et al., "Phi-4 technical report," *arXiv preprint arXiv:2412.08905*, 2024.

[2] J. Acharya and G. Ginde, *Can we enhance bug report quality using llms?: An empirical study of llm-based bug report generation*, 2025. arXiv: 2504.18804 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2504.18804.

[3] J. Achiam et al., "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[4] A. Afzal, J. Vladika, D. Braun, and F. Matthes, "Challenges in domain-specific abstractive summarization and how to overcome them," in *Proceedings of the 15th International Conference on Agents and Artificial Intelligence*, SCITEPRESS - Science and Technology Publications, 2023, pp. 682–689. DOI: 10.5220/0011744500003393. [Online]. Available: http://dx.doi.org/10.5220/0011744500003393.

[5] T. Ahmed, K. S. Pai, P. Devanbu, and E. T. Barr, *Automatic semantic augmentation of language model prompts (for code summarization)*, 2024. arXiv: 2304.06815 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2304.06815.

[6] A. Al Hasan, S. Saha, M. M. Imran, and T. S. Zaman, "Llput: Investigating large language models for bug report-based input generation," in *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, ser. FSE Companion '25, Clarion Hotel Trondheim, Trondheim, Norway: Association for Computing Machinery, 2025, pp. 1652–1659, ISBN: 9798400712760. DOI: 10.1145/3696630.3728701. [Online]. Available: https://doi.org/10.1145/3696630.3728701.

[7] P. Anbalagan and M. Vouk, "On mining data across software repositories," in *2009 6th IEEE International Working Conference on Mining Software Repositories*, 2009, pp. 171–174. DOI: 10.1109/MSR.2009.5069498.

[8] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" In *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06, Shanghai, China: Association for Computing Machinery, 2006, pp. 361–370, ISBN: 1595933751. DOI: 10.1145/1134285.1134336. [Online]. Available: https://doi.org/10.1145/1134285.1134336.

[9] D. Arya, W. Wang, J. L. C. Guo, and J. Cheng, "Analysis and detection of information types of open source software issue discussions," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19, Montreal, Quebec, Canada: IEEE Press, 2019, pp. 454–464. DOI: 10.1109/ICSE.2019.00058. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00058.

[10] A. Auriemma Citarella, M. Barbella, M. G. Ciobanu, F. De Marco, L. Di Biasi, and G. Tortora, "Assessing the effectiveness of rouge as unbiased metric in extractive vs. abstractive summarization techniques," *Journal of Computational Science*, vol. 87, p. 102 571, 2025, ISSN: 1877-7503. DOI: https://doi.org/10.1016/j.jocs.2025.102571. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877750325000481.

[11] S. Banerjee, Z. Syed, J. Helmick, M. Culp, K. Ryan, and B. Cukic, "Automated triaging of very large bug repositories," *Information and Software Technology*, vol. 89, pp. 1–13, 2017, ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2016.09.006. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584916301653.

[12] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann, "What makes a good bug report?" In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. SIGSOFT '08/FSE-16, Atlanta, Georgia: Association for Computing Machinery, 2008, pp. 308–318, ISBN: 9781595939951. DOI: 10.1145/1453101.1453146. [Online]. Available: https://doi.org/10.1145/1453101.1453146.

[13] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, "Extracting structural information from bug reports," in *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, ser. MSR '08, Leipzig, Germany: Association for Computing Machinery, 2008, pp. 27–30, ISBN: 9781605580241. DOI: 10.1145/1370750.1370757. [Online]. Available: https://doi.org/10.1145/1370750.1370757.

[14] T. F. Bissyandé, D. Lo, L. Jiang, L. Réveillere, J. Klein, and Y. Le Traon, "Got issues? who cares about it? a large scale investigation of issue trackers from github," in *2013 IEEE 24th international symposium on software reliability engineering (ISSRE)*, IEEE, 2013, pp. 188–197.

[15] L. Bo, W. Ji, X. Sun, T. Zhang, X. Wu, and Y. Wei, "Chatbr: Automated assessment and improvement of bug report quality using chatgpt," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '24, Sacramento, CA, USA: Association for Computing Machinery, 2024, pp. 1472–1483, ISBN: 9798400712487. DOI: 10.1145/3691620.3695518. [Online]. Available: https://doi.org/10.1145/3691620.3695518.

[16] T. B. Brown et al., "Language models are few-shot learners," in *Proceedings of the 34th International Conference on Neural Information Processing Systems*, ser. NIPS '20, Vancouver, BC, Canada: Curran Associates Inc., 2020, ISBN: 9781713829546.

[17] D. Cotroneo, R. Pietrantuono, S. Russo, and K. Trivedi, "How do bugs surface? a comprehensive study on the characteristics of software bugs manifestation," *Journal of Systems and Software*, vol. 113, pp. 27–43, 2016, ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2015.11.021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121215002460.

[18] D. Cubranic and G. Murphy, "Hipikat: Recommending pertinent software development artifacts," in *25th International Conference on Software Engineering, 2003. Proceedings.*, 2003, pp. 408–418. DOI: 10.1109/ICSE.2003.1201219.

[19] S. Davies and M. Roper, "What's in a bug report?" In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2014, pp. 1–10.

[20] Y. Deng, C. S. Xia, C. Yang, S. D. Zhang, S. Yang, and L. Zhang, "Large language models are edge-case generators: Crafting unusual programs for fuzzing deep learning libraries," in *Proceedings of the 46th IEEE/ACM international conference on software engineering*, 2024, pp. 1–13.

[21] 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://arxiv.org/abs/1810.04805.

[22] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior: A general approach to inferring errors in systems code," *ACM SIGOPS Operating Systems Review*, vol. 35, no. 5, pp. 57–72, 2001.

[23] Ç. Eren, K. Şahin, and E. Tüzün, "Analyzing bug life cycles to derive practical insights," in *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '23, Oulu, Finland: Association for Computing Machinery, 2023, pp. 162–171, ISBN: 9798400700446. DOI: 10.1145/3593434.3593504. [Online]. Available: https://doi.org/10.1145/3593434.3593504.

[24] S. Fang, T. Zhang, Y. Tan, H. Jiang, X. Xia, and X. Sun, "Representthemall: A universal learning representation of bug reports," in *Proceedings of the 45th International Conference on Software Engineering*, ser. ICSE '23, Melbourne, Victoria, Australia: IEEE Press, 2023, pp. 602–614, ISBN: 9781665457019. DOI: 10.1109/ICSE48619.2023.00060. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00060.

[25] M. Gambhir and V. Gupta, "Recent automatic text summarization techniques: A survey," *Artif. Intell. Rev.*, vol. 47, no. 1, pp. 1–66, Jan. 2017, ISSN: 0269-2821. DOI: 10.1007/s10462-016-9475-9. [Online]. Available: https://doi.org/10.1007/s10462-016-9475-9.

[26] N. Giarelis, C. Mastrokostas, and N. Karacapilidis, "Abstractive vs. extractive summarization: An experimental review," *Applied Sciences*, vol. 13, no. 13, 2023, ISSN: 2076-3417. DOI: 10.3390/app13137620. [Online]. Available: https://www.mdpi.com/2076-3417/13/13/7620.

[27] S. Goyal and A. Kaur, "Summarization of software bug report based on sentence semantic similarity (ssbrsss) technique," *Procedia Computer Science*, vol. 235, pp. 1761–1771, 2024, International Conference on Machine Learning and Data Engineering (ICMLDE 2023), ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2024.04.167. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050924008433.

[28] A. Grattafiori et al., "The llama 3 herd of models," *arXiv preprint arXiv:2407.21783*, 2024.

[29] D. Guo et al., *Deepseek-coder: When the large language model meets programming – the rise of code intelligence*, 2024. arXiv: 2401.14196 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2401.14196.

[30] S. Gupta and S. K. Gupta, "Abstractive summarization: An overview of the state of the art," *Expert Systems with Applications*, vol. 121, pp. 49–65, 2019, ISSN: 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2018.12.011. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0957417418307735.

[31] S. Gupta and G. S.K, "Comparative study of bug report summarization techniques," *Indian Journal of Computer Science and Engineering*, vol. 10, pp. 158–167, Dec. 2019. DOI: 10.21817/indjcse/2019/v10i6/191006041.

[32] J. He, N. Nazar, J. Zhang, T. Zhang, and Z. Ren, "Prst: A pagerank-based summarization technique for summarizing bug reports with duplicates," *International Journal of Software Engineering and Knowledge Engineering*, vol. 27, pp. 869–896, Jun. 2017. DOI: 10.1142/S0218194017500322.

[33] A. Hindle and C. Onuczko, "Preventing duplicate bug reports by continuously querying bug reports," *Empirical Softw. Engg.*, vol. 24, no. 2, pp. 902–936, Apr. 2019, ISSN: 1382-3256. DOI: 10.1007/s10664-018-9643-4. [Online]. Available: https://doi.org/10.1007/s10664-018-9643-4.

[34] E. J. Hu et al., *Lora: Low-rank adaptation of large language models*, 2021. arXiv: 2106.09685 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2106.09685.

[35] L. Huang et al., "A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions," *ACM Transactions on Information Systems*, vol. 43, no. 2, pp. 1–55, Jan. 2025, ISSN: 1558-2868. DOI: 10.1145/3703155. [Online]. Available: http://dx.doi.org/10.1145/3703155.

[36] "Ieee standard glossary of software engineering terminology," *IEEE Std 610.12-1990*, pp. 1–84, 1990. DOI: 10.1109/IEEESTD.1990.101064.

[37] A. Q. Jiang et al., *Mistral 7b*, 2023. arXiv: 2310.06825 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2310.06825.

[38] H. Jiang, X. Li, Z. Ren, J. Xuan, and Z. Jin, "Toward better summarizing bug reports with crowdsourcing elicited attributes," *IEEE Transactions on Reliability*, vol. 68, no. 1, pp. 2–22, 2018.

[39] H. Jiang, X. Li, Z. Ren, J. Xuan, and Z. Jin, "Towards better summarizing bug reports with crowdsourcing elicited attributes," *arXiv preprint arXiv:1810.00125*, 2018, Accessed: 2025-08-31. [Online]. Available: https://arxiv.org/abs/1810.00125.

[40] H. Jiang, J. Zhang, H. Ma, N. Nazar, and Z. Ren, "Mining authorship characteristics in bug repositories," *Science China Information Sciences*, vol. 60, no. 1, p. 012 107, 2017, Accessed: 2025-08-31. DOI: 10.1007/s11432-014-0372-y. [Online]. Available: https://link.springer.com/article/10.1007/s11432-014-0372-y.

[41] Z. Jiang, F. F. Xu, J. Araki, and G. Neubig, "How can we know what language models know?" *Transactions of the Association for Computational Linguistics*, vol. 8, M. Johnson, B. Roark, and A. Nenkova, Eds., pp. 423–438, 2020. DOI: 10.1162/tacl_a_00324. [Online]. Available: https://aclanthology.org/2020.tacl-1.28/.

[42] S. G. Jindal and A. Kaur, "Automatic keyword and sentence-based text summarization for software bug reports," *IEEE Access*, vol. 8, pp. 65 352–65 370, 2020. DOI: 10.1109/ACCESS.2020.2985222.

[43] E. G. S. Jr et al., *Which prompting technique should i use? an empirical investigation of prompting techniques for software engineering tasks*, 2025. arXiv: 2506.05614 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2506.05614.

[44] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the*

2014 International Symposium on Software Testing and Analysis (ISSTA 2014), ACM, 2014, pp. 437–440. DOI: 10.1145/2610384.2628055. [Online]. Available: https://doi.org/10.1145/2610384.2628055.

[45] S. Kang, J. Yoon, and S. Yoo, *Large language models are few-shot testers: Exploring llm-based general bug reproduction*, 2023. arXiv: 2209.11515 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2209.11515.

[46] M. Karim, A. Ihara, X. Yang, H. Iida, and K. Matsumoto, "Understanding key features of high-impact bug reports," Mar. 2017, pp. 53–58. DOI: 10.1109/IWESEP.2017.17.

[47] W. S. El-Kassas, C. R. Salama, A. A. Rafea, and H. K. Mohamed, "Automatic text summarization: A comprehensive survey," *Expert Systems with Applications*, vol. 165, p. 113 679, 2021, ISSN: 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2020.113679. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0957417420305030.

[48] S. Kim, K. Pan, and E. E. J. Whitehead, "Memories of bug fixes," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. SIGSOFT '06/FSE-14, Portland, Oregon, USA: Association for Computing Machinery, 2006, pp. 35–45, ISBN: 1595934685. DOI: 10.1145/1181775.1181781. [Online]. Available: https://doi.org/10.1145/1181775.1181781.

[49] M. Kirmani, G. Kaur, and M. mohd, "Analysis of abstractive and extractive summarization methods," *International Journal of Emerging Technologies in Learning (iJET)*, vol. 19, pp. 86–96, Jan. 2024. DOI: 10.3991/ijet.v19i01.46079.

[50] Y. Koh, S. Kang, and S. Lee, "Deep learning-based bug report summarization using sentence significance factors," *Applied Sciences*, vol. 12, no. 12, 2022, ISSN: 2076-3417. DOI: 10.3390/app12125854. [Online]. Available: https://www.mdpi.com/2076-3417/12/12/5854.

[51] W. Kryscinski, B. McCann, C. Xiong, and R. Socher, "Evaluating the factual consistency of abstractive text summarization," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, B. Webber, T. Cohn, Y. He, and Y. Liu, Eds., Online: Association for Computational Linguistics, Nov. 2020, pp. 9332–9346. DOI: 10.18653/v1/2020.emnlp-main.750. [Online]. Available: https://aclanthology.org/2020.emnlp-main.750/.

[52] A. Kumar, S. Haiduc, P. P. Das, and P. P. Chakrabarti, *Llms as evaluators: A novel approach to evaluate bug report summarization*, 2024. arXiv: 2409.00630 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2409.00630.

[53] S. Kumar and A. Solanki, "An abstractive text summarization technique using transformer model with self-attention mechanism," *Neural Comput. Appl.*, vol. 35, no. 25, pp. 18 603–18 622, Jun. 2023, ISSN: 0941-0643. DOI: 10.1007/s00521-023-08687-7. [Online]. Available: https://doi.org/10.1007/s00521-023-08687-7.

[54] Q. L. Le, A. Raad, J. Villard, J. Berdine, D. Dreyer, and P. W. O'Hearn, "Finding real bugs in big programs with incorrectness logic," *Proceedings of the ACM on Programming Languages*, vol. 6, no. OOPSLA1, pp. 1–27, 2022.

[55] H. Li, Y. Hao, Y. Zhai, and Z. Qian, "Enhancing static analysis for practical bug detection: An llm-integrated approach," *Proc. ACM Program. Lang.*, vol. 8, no. OOPSLA1, Apr. 2024. DOI: 10.1145/3649828. [Online]. Available: https://doi.org/10.1145/3649828.

[56] X. Li, H. Jiang, D. Liu, Z. Ren, and G. Li, "Unsupervised deep bug report summarization," in *Proceedings of the 26th Conference on Program Comprehension*, ser. ICPC '18, Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 144–155, ISBN: 9781450357142. DOI: 10.1145/3196321.3196326. [Online]. Available: https://doi.org/10.1145/3196321.3196326.

[57] Y. Li et al., "A knowledge enhanced large language model for bug localization," *Proc. ACM Softw. Eng.*, vol. 2, no. FSE, Jun. 2025. DOI: 10.1145/3729356. [Online]. Available: https://doi.org/10.1145/3729356.

[58] C.-Y. Lin, "ROUGE: A package for automatic evaluation of summaries," in *Text Summarization Branches Out*, Barcelona, Spain: Association for Computational Linguistics, Jul. 2004, pp. 74–81. [Online]. Available: https://aclanthology.org/W04-1013/.

[59] H. Liu, Y. Yu, S. Li, Y. Guo, D. Wang, and X. Mao, "Bugsum: Deep context understanding for bug report summarization," in *2020 IEEE/ACM 28th International Conference on Program Comprehension (ICPC)*, 2020, pp. 94–105. DOI: 10.1145/3387904.3389272.

[60] P. Liu et al., "Exploring chatgpt's capabilities on vulnerability management," in *Proceedings of the 33rd USENIX Conference on Security Symposium*, ser. SEC '24, Philadelphia, PA, USA: USENIX Association, 2024, ISBN: 978-1-939133-44-1.

[61] G. Long, "Learning software bug reports: A systematic literature review," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 34, no. 3, Article 34, 2025, Accessed: 2025-08-31. DOI: 10.1145/3750040. [Online]. Available: https://dl.acm.org/doi/10.1145/3750040.

[62] R. Lotufo, Z. Malik, and K. Czarnecki, "Modelling the 'hurried' bug report reading process to summarize bug reports," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, 2012, pp. 430–439. DOI: 10.1109/ICSM.2012.6405303.

[63] S. Mani, R. Catherine, V. S. Sinha, and A. Dubey, "Ausum: Approach for unsupervised bug report summarization," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12, Cary, North Carolina: Association for Computing Machinery, 2012, ISBN: 9781450316149. DOI: 10.1145/2393596.2393607. [Online]. Available: https://doi.org/10.1145/2393596.2393607.

[64] S. K. Mishra, K. Harshavardhan, S. Mitra, S. Saha, and P. Bhattacharyya, "Bug report summarization using multi-view multi-objective optimization framework," in *Proceedings of the Genetic and Evolutionary Computation Conference*, ser. GECCO '22, Boston, Massachusetts: Association for Computing Machinery, 2022, pp. 1245–1253, ISBN: 9781450392372. DOI: 10.1145/3512290.3528843. [Online]. Available: https://doi.org/10.1145/3512290.3528843.

[65] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker, "Automatic generation of natural language summaries for java classes," in *2013 21st International Conference on Program Comprehension (ICPC)*, 2013, pp. 23–32. DOI: 10.1109/ICPC.2013.6613830.

[66] S. Mukhtar, S. Lee, and J. Heo, "A multidocument summarization technique for informative bug summaries," *IEEE Access*, vol. 12, pp. 158 908–158 926, 2024. DOI: 10.1109/ACCESS.2024.3487443.

[67] G. Murray and G. Carenini, "Summarizing spoken and written conversations," in *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, ser. EMNLP '08, Honolulu, Hawaii: Association for Computational Linguistics, 2008, pp. 773–782.

[68] N. Nazar, Y. Hu, and J. He, "Summarizing software artifacts: A literature review," *Journal of Computer Science and Technology*, vol. 31, pp. 883–909, Sep. 2016. DOI: 10.1007/s11390-016-1671-1.

[69] OpenAI, *Gpt-3.5 turbo*, https://platform.openai.com/docs/models/gpt-3.5-turbo, Accessed: 2025-09-11, 2023.

[70] A. Patil, *Gitbugs: Bug reports for duplicate detection, retrieval augmented generation, triage, and more*, 2025. arXiv: 2504.09651 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2504.09651.

[71] M. Raatikainen et al., "Improved management of issue dependencies in issue trackers of large collaborative projects," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 2128–2148, 2023. DOI: 10.1109/TSE.2022.3212166.

[72] O. Rambow, L. Shrestha, J. Chen, and C. Lauridsen, "Summarizing email threads," Jan. 2004. DOI: 10.3115/1613984.1614011.

[73] E. Rashid and M. D. Ansari, "Fixing the bugs in software projects from software repositories for improvisation of quality," *Recent Advances in Electrical Electronic Engineering (Formerly Recent Patents on Electrical Electronic Engineering)*, vol. 12, Feb. 2019. DOI: 10.2174/1872121136661902151504 58.

[74] S. Rastkar, G. C. Murphy, and G. Murray, "Summarizing software artifacts: A case study of bug reports," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10, Cape Town, South Africa: Association for Computing Machinery, 2010, pp. 505–514, ISBN: 9781605587196. DOI: 10.1145/1806799.1806872. [Online]. Available: https://doi.org/10.1145/1806799.1806872.

[75] S. Rastkar, G. C. Murphy, and G. Murray, "Automatic summarization of bug reports," *IEEE Trans. Softw. Eng.*, vol. 40, no. 4, pp. 366–380, Apr. 2014, ISSN: 0098-5589. DOI: 10.1109/TSE.2013.2297712. [Online]. Available: https://doi.org/10.1109/TSE.2013.2297712.

[76] M. Ravaut, H. Chen, R. Zhao, C. Qin, S. Joty, and N. Chen, *Promptsum: Parameter-efficient controllable abstractive summarization*, 2023. arXiv: 2308.03117 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2308.03117.

[77] D. Roy, S. Fakhoury, and V. Arnaoudova, "Reassessing automatic evaluation metrics for code summarization tasks," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021, Athens, Greece: Association for Computing Machinery, 2021, pp. 1105–1116, ISBN: 9781450385626. DOI: 10.1145/3468264.3468588. [Online]. Available: https://doi.org/10.1145/3468264.3468588.

[78] B. Rozière et al., *Code llama: Open foundation models for code*, 2024. arXiv: 2308.12950 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2308.12950.

[79] P. Schugerl, J. Rilling, and P. Charland, "Mining bug repositories–a quality assessment," in *2008 International Conference on Computational Intelligence for Modelling Control Automation*, 2008, pp. 1105–1110. DOI: 10.1109/CIMCA.2008.63.

[80] H. Shakil, A. Farooq, and J. Kalita, "Abstractive text summarization: State of the art, challenges, and improvements," *Neurocomputing*, vol. 603, p. 128 255, Oct. 2024, ISSN: 0925-2312. DOI: 10.1016/j.neucom.2024.128255. [Online]. Available: http://dx.doi.org/10.1016/j.neucom.2024.128255.

[81] Y. Shao, T. Zhang, Y. Tan, H. Jiang, X. Xia, and X. Sun, "Towards effective bug report summarization by domain-specific representation learning," *arXiv preprint arXiv:2409.00630*, 2024, Accessed: 2025-08-31. [Online]. Available: https://arxiv.org/abs/2409.00630.

[82] V. Sobreira, T. Durieux, F. Madeiral, M. Monperrus, and M. A. Maia, "Dissection of a Bug Dataset: Anatomy of 395 Patches from Defects4J," in *Proceedings of SANER*, 2018. DOI: 10.1109/SANER.2018.8330203.

[83] Supriyono, A. P. Wibawa, Suyono, and F. Kurniawan, "A survey of text summarization: Techniques, evaluation and challenges," *Natural Language Processing Journal*, vol. 7, p. 100 070, 2024, ISSN: 2949-7191. DOI: https://doi.org/10.1016/j.nlp.2024.100070. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2949719124000189.

[84] G. Team et al., "Gemini: A family of highly capable multimodal models," *arXiv preprint arXiv:2312.11805*, 2023.

[85] G. Team et al., "Gemma: Open models based on gemini research and technology," *arXiv preprint arXiv:2403.08295*, 2024.

[86] Y. Tian, C. Sun, and D. Lo, "Improved duplicate bug report identification," in *2012 16th European Conference on Software Maintenance and Reengineering*, 2012, pp. 385–390. DOI: 10.1109/CSMR.2012.48.

[87] A. Vahabzadeh, A. Milani Fard, and A. Mesbah, "An empirical study of bugs in test code," Sep. 2015, pp. 101–110. DOI: 10.1109/ICSM.2015.7332456.

[88] S. Wan and K. McKeown, "Generating overview summaries of ongoing email thread discussions," in *Proceedings of the 20th International Conference on Computational Linguistics*, ser. COLING '04, Geneva, Switzerland: Association for Computational Linguistics, 2004, 549–es. DOI: 10.3115/1220355.1220434. [Online]. Available: https://doi.org/10.3115/1220355.1220434.

[89] J. Wei et al., "Chain-of-thought prompting elicits reasoning in large language models," in *Advances in Neural Information Processing Systems*, vol. 35, Curran Associates, Inc., 2022, pp. 24 824–24 837. DOI: 10.5555/3600270.3602070. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2022/file/9d5609613524ecf4f15af0f7b31abca4-Paper-Conference.pdf.

[90] X. Xia, D. Lo, E. Shihab, and X. Wang, "Automated bug report field reassignment and refinement prediction," *IEEE Transactions on Reliability*, vol. 65, no. 3, pp. 1094–1113, 2016. DOI: 10.1109/TR.2015.2484074.

[91] X. Xia, D. Lo, X. Wang, and B. Zhou, "Accurate developer recommendation for bug resolution," in *2013 20th Working Conference on Reverse Engineering (WCRE)*, 2013, pp. 72–81. DOI: 10.1109/WCRE.2013.6671282.

[92] B. Xiang and Y. Shao, "Sumllama: Efficient contrastive representations and fine-tuned adapters for bug report summarization," *IEEE Access*, vol. 12, pp. 78 562–78 571, 2024. DOI: 10.1109/ACCESS.2024.3397326.

[93] J. Xuan, Y. Hu, and H. Jiang, "Debt-prone bugs: Technical debt in software maintenance," *arXiv preprint arXiv:1704.04766*, 2017.

[94] A. Yang et al., "Qwen3 technical report," *arXiv preprint arXiv:2505.09388*, 2025.

[95] C.-Z. Yang, C.-M. Ao, and Y.-H. CHUNG, "Towards an improvement of bug report summarization using two-layer semantic information," *IEICE Transactions on Information and Systems*, vol. E101.D, pp. 1743–1750, Jul. 2018. DOI: 10.1587/transinf.2017KBP0016.

[96] K. Zechner, "Automatic summarization of open-domain multiparty dialogues in diverse genres," *Computational Linguistics*, vol. 28, no. 4, pp. 447–485, 2002. DOI: 10.1162/089120102762671945. [Online]. Available: https://aclanthology.org/J02-4003/.

[97] H. Zhang, P. S. Yu, and J. Zhang, *A systematic survey of text summarization: From statistical methods to large language models*, 2024. arXiv: 2406.11289 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2406.11289.

[98] J. Zhang, X. Wang, D. Hao, B. Xie, L. Zhang, and H. Mei, "A survey on bug-report analysis," *Science China Information Sciences*, vol. 58, no. 2, pp. 1–24, Feb. 2015, ISSN: 1869-1919. DOI: 10.1007/s11432-014-5241-2. [Online]. Available: https://doi.org/10.1007/s11432-014-5241-2.

[99] T. Zhang, V. Kishore, F. Wu, K. Q. Weinberger, and Y. Artzi, *Bertscore: Evaluating text generation with bert*, 2020. arXiv: 1904.09675 [cs.CL]. [Online]. Available: https://arxiv.org/abs/1904.09675.

[100] X. Zhang, Z. Zhu, and Y. Li, "Enhancing bug localization through bug report summarization," in *2023 IEEE International Conference on Data Mining (ICDM)*, 2023, pp. 1541–1546. DOI: 10.1109/ICDM58522.2023.00205.

[101] Z. Zhao, E. Wallace, S. Feng, D. Klein, and S. Singh, "Calibrate before use: Improving few-shot performance of language models," in *Proceedings of the 38th*

International Conference on Machine Learning, M. Meila and T. Zhang, Eds., ser. Proceedings of Machine Learning Research, vol. 139, PMLR, Jul. 2021, pp. 12 697–12 706. [Online]. Available: https://proceedings.mlr.press/v139/zhao21c.html.

[102] L. Zhou and E. Hovy, "A web-trained extraction summarization system.," Jan. 2003. DOI: 10.3115/1073445.1073482.

[103] X. Zhu and G. Penn, "Summarization of spontaneous conversations," Sep. 2006. DOI: 10.21437/Interspeech.2006-430.

[104] W. Zou, D. Lo, Z. Chen, X. Xia, Y. Feng, and B. Xu, "How practitioners perceive automated bug report management techniques," *IEEE Transactions on Software Engineering*, vol. 46, no. 8, pp. 836–862, 2020. DOI: 10.1109/TSE.2018.2870414.

Appendices

Appendix A

Formulas and Important Definitions

A.1 Evaluation Metric Definitions

- **ROUGE (Recall-Oriented Understudy for Gisting Evaluation) [58]**: Measures the overlap between the predicted and reference summaries at the n-gram level. It includes ROUGE-N (unigrams, bigrams, trigrams), ROUGE-L (longest common subsequence), and other variants. The higher the value, the better the model performance.
- **BERTScore [99]**: Measures the similarity between predicted and reference summaries by comparing BERT embeddings. It uses cosine similarity between token embeddings and includes Precision, Recall, and F1 scores. The higher the value, the better the model performance.

A.2 Mathematical Formulas

A.2.1 ROUGE-N Formula

The ROUGE-N score measures the overlap of n-grams (where N can be 1, 2, or 3) between the system-generated summary and the reference summary. The formula is:

$$\text{ROUGE-N} = \frac{\sum_{s \in \text{system}} \text{Count of N-grams in system summary}}{\sum_{r \in \text{reference}} \text{Count of N-grams in reference summary}} \tag{A.1}$$

Where: - The numerator is the count of overlapping N-grams in the system-generated summary. - The denominator is the total number of N-grams in the reference sum-

mary.

A.2.2 ROUGE-L Formula

The ROUGE-L score evaluates the similarity between a system-generated summary and a reference summary based on the Longest Common Subsequence (LCS). It considers both precision and recall, defined as:

$$P_{LCS} = \frac{\text{LCS}(X, Y)}{\text{length of system summary } X} \tag{A.2}$$

$$R_{LCS} = \frac{\text{LCS}(X, Y)}{\text{length of reference summary } Y} \tag{A.3}$$

$$F_{LCS} = \frac{(1 + \beta^2) \cdot P_{LCS} \cdot R_{LCS}}{R_{LCS} + \beta^2 \cdot P_{LCS}} \tag{A.4}$$

Where: - $\text{LCS}(X, Y)$ is the length of the longest common subsequence between the system summary X and the reference summary Y. - β is a parameter (commonly set as the ratio of recall to precision, often 1) that balances precision and recall in the F-measure. - P_{LCS}, R_{LCS}, and F_{LCS} represent precision, recall, and F-score based on the LCS.

A.2.3 BERTScore Formula

BERTScore measures the similarity between the system summary and the reference summary by comparing the embeddings of individual tokens using BERT. It includes three components: Precision, Recall, and F1 score:

$$\text{BERTScore Precision} = \frac{1}{|S|} \sum_{s \in \text{system}} \max_{r \in \text{reference}} \text{cosine\_similarity}(BERT(s), BERT(r)) \tag{A.5}$$

$$\text{BERTScore Recall} = \frac{1}{|R|} \sum_{r \in \text{reference}} \max_{s \in \text{system}} \text{cosine\_similarity}(BERT(s), BERT(r)) \tag{A.6}$$

$$\text{BERTScore F1} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{A.7}$$

Where: - cosine_similarity($BERT(s), BERT(r)$) measures the similarity between the embeddings of tokens s and r in the system and reference summaries, respectively. - $|S|$ and $|R|$ represent the total number of tokens in the system and reference summaries, respectively.