
PROGRESSIVE CODE INTEGRATION FOR ABSTRACTIVE BUG REPORT SUMMARIZATION

Shaira Sadia Karim^{*†}
shairasadia@iut-dhaka.edu

Abrar Mahmud Rahim^{*†}
abrarmahmud@iut-dhaka.edu

Lamia Alam^{*†}
lamiaalam@iut-dhaka.edu

Ishmam Tashdeed[†]
ishmamtashdeed@iut-dhaka.edu

Lutfun Nahar Lota[†]
lota@iut-dhaka.edu

Dr. Md. Abu Raihan M. Kamal[†]
raihan.kamal@iut-dhaka.edu

Dr. Md. Azam Hossain[†]
azam@iut-dhaka.edu

[†]Islamic University of Technology, Gazipur, Bangladesh

^{*} Equal Contribution

December 2, 2025

ABSTRACT

Bug reports are often unstructured and verbose, making it challenging for developers to efficiently comprehend software issues. Existing summarization approaches typically rely on surface-level textual cues, resulting in incomplete or redundant summaries, and they frequently ignore associated code snippets, which are essential for accurate defect diagnosis. To address these limitations, we propose a progressive code-integration framework for LLM-based abstractive bug report summarization. Our approach incrementally incorporates long code snippets alongside textual content, overcoming standard LLM context window constraints and producing semantically rich summaries. Evaluated on four benchmark datasets using eight LLMs, our pipeline outperforms extractive baselines by 7.5%–58.2% and achieves performance comparable to state-of-the-art abstractive methods, highlighting the benefits of jointly leveraging textual and code information for enhanced bug comprehension.

1 Introduction

Bug reports are a critical component of software maintenance, providing developers with information about defects, failure conditions, and potential fixes. However, their unstructured nature, redundancy, and occasional ambiguity make them challenging to analyze efficiently [1, 2]. Existing summarization methods, ranging from supervised learning approaches to classical NLP-based techniques, predominantly focus on textual content and often overlook the semantic connections between bug descriptions and their corresponding source code [3]. While extractive approaches are widely used, they can produce redundant, fragmented, or incoherent summaries. Abstractive methods, which have the potential to generate more concise and semantically rich summaries, remain underexplored due to their technical complexity [4, 5].

The advent of Large Language Models (LLMs) has enabled more advanced summarization capabilities [6, 7]. However, prior LLM-based approaches for bug-report summarization largely ignore the inclusion of full code snippets [8], which are crucial for understanding the context of software defects. LLMs also face challenges related to missing background knowledge, domain-specific terminology, and structured code elements [9, 10], although prompt engineering has been shown to mitigate some of these limitations and improve generation quality [11, 12].

To address these gaps, we propose an LLM-based abstractive bug-report summarization framework that integrates textual bug reports with complete code snippets through carefully designed prompt engineering. Our method is

evaluated across multiple benchmark datasets, including SDS [13], ADS [14], Fang et al.’s corpus [15], and Defects4J [16]. The primary contributions of this work are threefold: (1) the first explicit use of full code snippets to enhance abstractive bug-report summarization, (2) a prompt-engineered framework for effectively combining text and code, and (3) demonstrably improved semantic summary quality as measured by BERTScore [17].

2 Related Work

2.1 Abstractive Bug Report Summarization

Abstractive summarization of bug reports presents significant challenges due to the technical complexity, domain-specific terminology, and heterogeneous structure of report content [18, 19]. Traditional methods often struggle to capture the semantic essence of both textual descriptions and associated code, leading to summaries that may omit critical fault-relevant information.

Recent advances in Large Language Models (LLMs) have substantially improved context-aware summarization capabilities [6]. Approaches such as RTA [15], KSCLP [20], and SumLLaMA [21] leverage pre-training on large corpora, knowledge-specific or contrastive learning techniques, and parameter-efficient fine-tuning to enhance performance on bug-report summarization tasks. While these methods achieve improved fluency and abstraction compared to extractive techniques, they generally focus on the textual content of bug reports and do not incorporate the underlying buggy code [22]. This omission limits the ability of these models to fully capture the context of software defects and constrains their applicability in code-aware summarization scenarios.

2.2 Prompt Engineering

Prompt engineering has emerged as a powerful technique for guiding LLMs in complex software engineering tasks [23]. Frameworks such as LIBRO [24], ChatBR [25], and LLIFT [26] employ structured prompts, few-shot examples, and progressive prompting strategies to enhance performance on tasks including test-case generation, bug detection, and bug-report generation. These methods leverage the inherent generative capabilities of LLMs while providing explicit cues to shape output content and maintain task relevance.

Similarly, ChatGPT and other general-purpose LLMs have demonstrated effectiveness in vulnerability management and abstractive bug-report summarization [27]. However, current prompt-based approaches predominantly focus on textual bug reports and do not incorporate the underlying code associated with defects. This limitation constrains their ability to capture the full semantic and structural context of software bugs, highlighting a gap that motivates the development of code-aware abstractive summarization techniques.

3 Approach

3.1 Problem Statement

Let \mathcal{X}_T denote the space of textual bug-report utterances (e.g., natural-language descriptions, reproduction steps, error messages) and let \mathcal{X}_C denote the space of code snippets (e.g., stack traces, diffs, method bodies). A bug report is represented as a pair

$$x = (x_T, x_C) \in \mathcal{X}_T \times \mathcal{X}_C,$$

where x_T is a sequence of natural-language tokens and x_C is a sequence of code tokens. Formally,

$$\begin{aligned} x_T &= (t_1, t_2, \dots, t_{n_T}), & t_i &\in V_T, \\ x_C &= (c_1, c_2, \dots, c_{n_C}), & c_j &\in V_C, \end{aligned}$$

with V_T and V_C denoting the vocabularies for text and code respectively. A summary is an abstractive natural-language sequence

$$y = (y_1, y_2, \dots, y_m) \in \mathcal{Y},$$

where $y_k \in V_T$ and $m \ll n_T + n_C$.

We define the *abstractive bug-report summarization problem* as learning a conditional probability distribution

$$p_\theta(y | x_T, x_C),$$

parameterized by θ , such that generated sequences y satisfy two criteria:

- **Semantic Sufficiency:** The summary y preserves the core meaning of the bug report, capturing essential fault symptoms, reproduction conditions, and implicated code regions. More formally, we assume a semantic representation function

$$\phi : \mathcal{X}_T \times \mathcal{X}_C \rightarrow \mathcal{Z},$$

mapping inputs to a latent semantic space \mathcal{Z} , and a corresponding summary-induced representation

$$\psi : \mathcal{Y} \rightarrow \mathcal{Z}.$$

A summary is semantically sufficient if

$$d(\phi(x_T, x_C), \psi(y)) \leq \varepsilon,$$

for a task-appropriate similarity metric $d : \mathcal{Z} \times \mathcal{Z} \rightarrow \mathbb{R}_{\geq 0}$ (e.g., cosine distance in an embedding space) and a distortion threshold $\varepsilon > 0$.

- **Abstractive Novelty:** The summary may contain tokens not present in the original bug report or code snippet. Formally,

$$y \not\subseteq x_T \cup x_C \quad \text{is allowed,}$$

distinguishing the task from extractive summarization, which restricts summaries to subsequences or selections from the input.

Thus, abstractive summarization requires learning a generative transformation

$$f_\theta : \mathcal{X}_T \times \mathcal{X}_C \rightarrow \mathcal{Y},$$

capable of synthesizing novel linguistic expressions while preserving semantic fidelity.

To operationalize this transformation with a Large Language Model (LLM), we construct a structured prompt

$$\pi(x) = g(x_T, x_C),$$

where g is a deterministic formatting function (e.g., using delimiters, metadata fields, or role annotations) that exposes salient textual and code structure to the model. The LLM produces a summary

$$\hat{y} = \arg \max_{y \in \mathcal{Y}} p_\theta(y \mid \pi(x)),$$

or a sampled approximation thereof.

The objective of this work is to design the prompt representation g , specify an effective prompting or training strategy for the LLM, and evaluate the resulting summaries according to both linguistic quality and semantic fidelity with respect to the original bug report and its associated code.

3.2 Data Preprocessing

We construct our dataset by leveraging Defects4J [16], a curated benchmark suite containing real-world software defects from Java projects, and by augmenting it with bug-report information obtained via a custom web-scraping and data-normalization pipeline [28]. Formally, let \mathcal{B} denote the set of bug identifiers provided by Defects4J. For each bug $b \in \mathcal{B}$, Defects4J supplies (i) the buggy and fixed program versions and (ii) metadata linking each defect to an external issue tracker.

Our preprocessing pipeline defines a mapping

$$h : \mathcal{B} \rightarrow \mathcal{X}_T \times \mathcal{X}_C,$$

which retrieves and normalizes the relevant textual and code modalities for each bug. Concretely, for a given bug b :

- We scrape and extract the associated natural-language bug report from its corresponding issue tracker, yielding an unstructured text document r_b .
- We identify code snippets s_b related to the defect by mining commit diffs, stack traces, and referenced file paths from the Defects4J metadata. These snippets often exhibit heterogeneous formats, including inline code blocks, patch fragments, and trace logs.
- We apply a normalization operator

$$\nu : \mathcal{R} \times \mathcal{S} \rightarrow \mathcal{X}_T \times \mathcal{X}_C,$$

where \mathcal{R} and \mathcal{S} denote raw textual and code sources respectively, to produce structured sequences of tokens suitable for downstream LLM consumption. This step includes HTML cleaning, tokenization, code-block extraction, Unicode normalization, and removal of incidental noise (e.g., markup artifacts or irrelevant metadata).

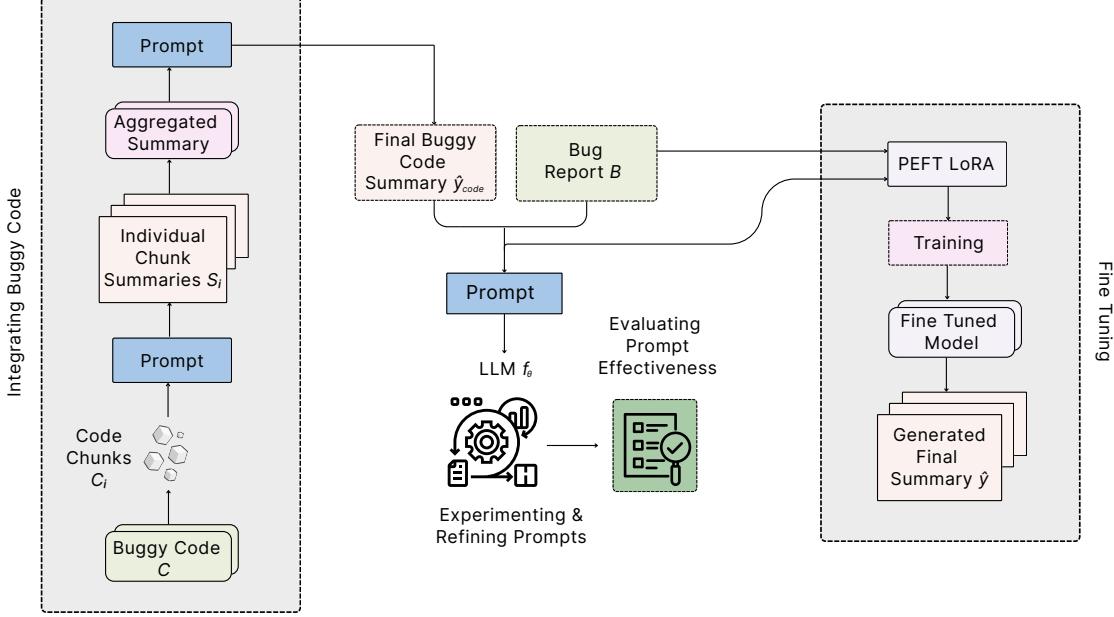


Figure 1: End-to-end pipeline for abstractive bug report summarization. Here long buggy code is chunked and summarized, combined with bug-report text through prompting and refined using LoRA-based fine-tuning to produce summaries.

Thus, each bug b is ultimately mapped to a pair

$$(x_T^{(b)}, x_C^{(b)}) = h(b),$$

where $x_T^{(b)} \in \mathcal{X}_T$ is a cleaned and tokenized bug-report text sequence, and $x_C^{(b)} \in \mathcal{X}_C$ is a set of structurally standardized code snippets. This preprocessing pipeline transforms heterogeneous, unstructured data into a consistently formatted representation that is directly usable by LLM-based summarization models.

3.3 Prompt Engineering

Our approach employs multi-modal structured prompts that explicitly integrate both textual bug-report content and associated code snippets in order to guide the LLM toward producing coherent and semantically faithful abstractive summaries. Let $\pi(x)$ denote the prompt constructed from an input pair $x = (x_T, x_C)$, where π formats the content using templated delimiters, role specifications, and modality markers. This structured formulation provides the LLM with clear boundaries between natural-language descriptions and code-related information, reducing hallucination and improving semantic grounding.

We explored several prompting regimes, including zero-shot, one-shot, and few-shot strategies. In the zero-shot setting, the model receives only the structured prompt and is expected to infer the desired summarization behavior. One-shot and few-shot variants extend the prompt with one or more demonstration pairs $(x^{(i)}, y^{(i)})$, enabling the model to condition on exemplar input–output relationships. These exemplars were selected to minimize domain-specific bias and to expose the model to representative patterns of bug-report language and code structures. Formally, a few-shot prompt can be viewed as the concatenation

$$\pi_{fs}(x) = \langle \pi(x^{(1)}), y^{(1)}, \dots, \pi(x^{(k)}), y^{(k)}, \pi(x) \rangle,$$

where k determines the number of demonstrations provided.

Long code snippets, which frequently exceed the context-length constraints of contemporary LLMs, required an additional hierarchical processing step. Given a code sequence $x_C = (c_1, \dots, c_{n_C})$ and a maximum permissible context length L , we partition the code into contiguous segments

$$x_C^{(1)}, x_C^{(2)}, \dots, x_C^{(M)}, \quad M = \left\lceil \frac{n_C}{L} \right\rceil,$$

each satisfying the length limit. For each segment $x_C^{(m)}$, we generate an intermediate summary $s^{(m)}$ by conditioning the LLM on a prompt $\pi(x_T, x_C^{(m)})$. These intermediate representations are subsequently aggregated into a consolidated

code-level summary \tilde{s} through a second-stage abstraction pass, effectively compressing multi-segment code information into a unified semantic description. The final prompt used for generating the complete abstractive summary is then formed by combining the bug-report text x_T with the aggregated code summary \tilde{s} , enabling the LLM to operate on a contextually rich yet size-constrained representation.

This hierarchical prompting strategy ensures that both textual and code modalities contribute meaningfully to the final abstractive summary, while also preserving compatibility with LLM context limitations and maintaining high coherence across different stages of generation.

3.4 Fine-Tuning

To adapt pre-trained LLMs to the domain of software bug-report summarization, we conducted parameter-efficient fine-tuning using supervised examples derived from the processed dataset. Let each training instance consist of an input pair $x = (x_T, x_C)$ and a corresponding target summary y . We constructed two dataset variants to evaluate the role of code information in the learning process. The first variant conditions the model solely on the natural-language bug-report text x_T , while the second augments this input with the aggregated code summary \tilde{s} , producing an enriched representation (x_T, \tilde{s}) . This design enables a controlled examination of whether incorporating code-derived semantic content materially improves model performance.

Fine-tuning was performed using Low-Rank Adaptation (LoRA) [29], a method that inserts trainable low-rank matrices into selected weight components of the transformer architecture. Formally, a weight matrix $W_0 \in \mathbb{R}^{d \times d}$ is reparameterized during fine-tuning as

$$W = W_0 + BA,$$

where $A \in \mathbb{R}^{r \times d}$ and $B \in \mathbb{R}^{d \times r}$ are the learned low-rank factors with $r \ll d$. This formulation substantially reduces the number of trainable parameters, allowing efficient specialization of the base model without modifying the full parameter set. LoRA thus offers a computationally tractable way to fine-tune large-scale models while preserving their general linguistic and reasoning capabilities.

Two fine-tuned model variants were produced: one trained on bug-report-only inputs and one trained on inputs enriched with code summaries. These variants allow a direct comparison of the conditional distributions

$$p_{\theta_{\text{BR}}}(y | x_T) \quad \text{and} \quad p_{\theta_{\text{BR+CS}}}(y | x_T, \tilde{s}),$$

thereby quantifying the contribution of code-level context to the abstractive summarization task. The resulting models provide empirical insight into how multimodal conditioning influences the fidelity, conciseness, and technical accuracy of generated summaries.

3.5 Research Questions

This work is guided by three central research questions that structure our investigation into LLM-based abstractive bug-report summarization. The first question (**RQ1**) concerns the comparative effectiveness of our proposed methodology relative to state-of-the-art approaches. We aim to determine whether structured prompting, multimodal integration, and parameter-efficient fine-tuning yield measurable improvements in summary quality when benchmarked against existing extractive and abstractive baselines.

The second question (**RQ2**) examines the role of code information in shaping summary fidelity and technical accuracy. By contrasting models trained or prompted with bug-report text alone against those additionally conditioned on full or hierarchically summarized code snippets, we seek to evaluate how code-level context influences semantic completeness, specificity, and alignment with underlying defect characteristics.

The third question (**RQ3**) focuses on the effect of methodological choices—specifically prompt engineering strategies and fine-tuning configurations—on summarization performance across different LLMs and dataset variants. This question addresses the broader issue of how model behavior varies under different adaptation regimes and aims to identify principled strategies for constructing, prompting, and customizing LLMs for software-maintenance tasks.

3.6 Experimental Setup

3.6.1 Dataset

To evaluate the effectiveness and generalizability of our approach, we conducted experiments on four benchmark datasets that vary in report length, authorship patterns, and overall scale. The Defects4J dataset [16] serves as our primary benchmark for code-aware summarization, providing paired bug reports and corresponding code snippets,

which allows for multimodal modeling and hierarchical summarization strategies. The SDS [13] and ADS [14] datasets are employed as extractive summarization baselines, offering manually annotated summaries or reference selections that enable a comparative assessment of abstractive methods against traditional extractive approaches. Finally, we include the dataset curated by Fang et al. [15] to test generalization across different projects and reporting conventions. This dataset provides sampled bug-report subsets with high-quality human annotations, facilitating evaluation of model performance in scenarios that differ from those in the Defects4J, SDS, and ADS benchmarks. Collectively, these datasets allow a comprehensive assessment of our models in both multimodal and text-only summarization contexts.

3.6.2 Models

We evaluated eight state-of-the-art LLMs spanning a range of sizes, architectures, and levels of code specialization to assess their effectiveness in abstractive bug-report summarization. The selected models include CodeLlama [30], which is explicitly trained for code understanding and generation; Llama-3.1 [31] and Mistral [32], representing large general-purpose LLMs; Phi-3 [33] and Gemma [34], designed for mixed-language reasoning and knowledge-intensive tasks; Qwen3 [35], optimized for multilingual and technical text processing; DeepSeek Coder [36], focused on code-related tasks; and GPT-3.5 Turbo [37], a widely used general-purpose model. By comparing these models, we aim to analyze the relative benefits of general-purpose versus code-specialized capabilities, as well as the influence of model scale and architecture on the quality, coherence, and technical fidelity of generated summaries.

3.6.3 Evaluation Metrics

To rigorously quantify the quality of generated abstractive summaries, let $Y = \{y^{(i)}\}_{i=1}^N$ denote the set of reference summaries and $\hat{Y} = \{\hat{y}^{(i)}\}_{i=1}^N$ the corresponding set of model-generated summaries for N evaluation instances. Each summary $y^{(i)}$ or $\hat{y}^{(i)}$ is a sequence of tokens in V_T , the textual vocabulary.

Our primary metric is **BERTScore** [17], which measures semantic similarity between generated and reference summaries. Let $E(y)$ denote the sequence of contextual embeddings obtained from a pre-trained transformer for a summary y . BERTScore computes a precision, recall, and F1 measure based on pairwise cosine similarity between token embeddings:

$$\begin{aligned}\text{Precision}(\hat{y}^{(i)}, y^{(i)}) &= \frac{1}{|\hat{y}^{(i)}|} \sum_{\hat{e} \in E(\hat{y}^{(i)})} \max_{e \in E(y^{(i)})} \cos(\hat{e}, e), \\ \text{Recall}(\hat{y}^{(i)}, y^{(i)}) &= \frac{1}{|y^{(i)}|} \sum_{e \in E(y^{(i)})} \max_{\hat{e} \in E(\hat{y}^{(i)})} \cos(e, \hat{e}), \\ \text{F1} &= 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}.\end{aligned}$$

For comparison with prior work, we also report **ROUGE-N** and **ROUGE-L** [38]. ROUGE-N measures n -gram overlap between the generated and reference summaries:

$$\text{ROUGE-N} = \frac{\sum_{i=1}^N \sum_{g \in \text{ngrams}_n(y^{(i)})} \min(\text{Count}(g, \hat{y}^{(i)}), \text{Count}(g, y^{(i)}))}{\sum_{i=1}^N \sum_{g \in \text{ngrams}_n(y^{(i)})} \text{Count}(g, y^{(i)})}.$$

ROUGE-L captures the longest common subsequence (LCS) between the generated and reference sequences:

$$\text{ROUGE-L} = \frac{\sum_{i=1}^N \text{LCS}(\hat{y}^{(i)}, y^{(i)})}{\sum_{i=1}^N |y^{(i)}|}.$$

Collectively, these metrics provide a rigorous evaluation of both semantic fidelity (via BERTScore) and surface-level textual alignment (via ROUGE), allowing a comprehensive assessment of abstractive summarization performance.

3.6.4 Experimental Configurations

All experiments were performed on a single NVIDIA RTX A6000 GPU within a controlled Docker-based environment to ensure reproducibility. Each input pair $x = (x_T, x_C)$ was tokenized into sequences of tokens from the respective vocabularies V_T and V_C , applying truncation or padding as necessary to satisfy the context length limitations of each LLM. We evaluated zero-shot, one-shot, and few-shot prompting strategies, where few-shot prompts incorporated k exemplar input-output pairs concatenated with the target input, formally represented as

$$\pi_{\text{fs}}(x) = \langle (\pi(x^{(1)}), y^{(1)}), \dots, (\pi(x^{(k)}), y^{(k)}), \pi(x) \rangle.$$

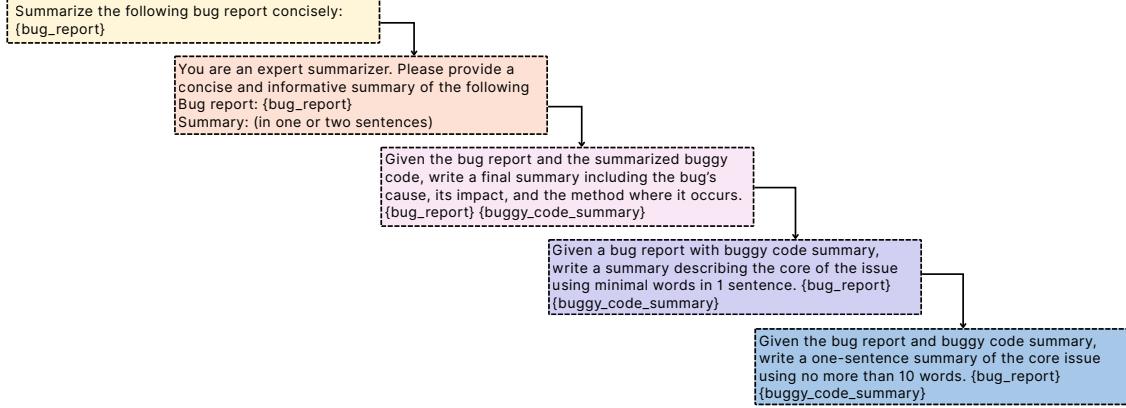


Figure 2: Overview of the iterative prompt refinement process. Prompts were progressively adjusted based on the quality of model-generated summaries to improve accuracy, completeness, and readability.

For code-containing inputs, sequences exceeding a maximum token length $L_{\max} = 1024$ were partitioned into $M = \lceil |x_C|/L_{\max} \rceil$ contiguous chunks $x_C^{(1)}, \dots, x_C^{(M)}$. Each chunk was summarized individually, producing intermediate summaries $\{s^{(m)}\}_{m=1}^M$ that were then aggregated into a consolidated code-level summary \tilde{s} , forming the final input (x_T, \tilde{s}) for abstractive summarization.

Text generation from the LLM employed standard decoding strategies including beam search, top- k sampling, and repetition constraints to improve diversity and reduce hallucinations. Fine-tuning was conducted on the Defects4J dataset, using a 90/10 train/validation split. The training configuration included a batch size of 1, gradient accumulation over 4 steps to emulate a larger effective batch size, and a learning rate of $\eta = 5 \times 10^{-5}$. Let θ denote the model parameters, then the fine-tuning update at step t followed

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta \mathcal{L}(\hat{y}^{(t)}, y^{(t)}),$$

where \mathcal{L} is the cross-entropy loss between the generated summary $\hat{y}^{(t)}$ and the reference $y^{(t)}$.

Table 1: ROUGE scores of baseline models and LLMs on Fang et al. corpus using bug reports only. Fine-tuned models evaluated on Defects4J; GPT-3.5 Turbo in zero-shot setting.

Model	ROUGE-1	ROUGE-2	ROUGE-L
SUMLLAMA [21]	0.4426	0.2515	0.4101
KSCLP [20]	0.4133	0.2202	0.3789
RTA [15]	0.3919	0.2057	0.3597
BugSum [39]	0.2591	0.1166	0.2469
DeepSum [3]	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

3.7 Baselines

To evaluate the performance of our proposed approach, we compare against five established bug-report summarization methods. SUMLLAMA [21] leverages transformer-based models to generate abstractive summaries using large-scale pretraining. KSCLP [20] applies a knowledge-aware sequence-to-sequence framework, incorporating structural information from both bug reports and software artifacts. RTA [15] uses a retrieval-augmented summarization approach

Table 2: BERTScore of LLMs on multiple datasets comparing bug reports alone vs bug reports + code; highlights effects of fine-tuning and code integration.

Dataset	Model	Bug Reports			Bug Reports + Buggy Code		
		Precision	Recall	F1 Score	Precision	Recall	F1 Score
Defects4J [16]	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 [13]	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 [14]	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 [15]	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	-	-	-

to enhance coverage of relevant report content. BugSum [39] focuses on extractive summarization with heuristics designed to identify salient sentences in bug reports. DeepSum [3] integrates neural sequence modeling with software-specific features for both extractive and abstractive summarization. These baselines provide a diverse set of comparison points, spanning both extractive and abstractive strategies, and enable rigorous evaluation of the advantages offered by multimodal LLM-based summarization.

4 Experimental Results

4.1 Effectiveness Compared to Baselines (RQ1)

We evaluated the bug-report-only component of our framework against both extractive and abstractive baselines. The extractive baselines include BugSum [39] and DeepSum [3], while the abstractive baselines comprise SUMLLAMA [21], KSCLP [20], and RTA [15]. Performance was assessed on the Fang et al. corpus [15] using ROUGE and BERTScore metrics.

Table 3: BERTScore (Precision, Recall, F1) on Defects4J for bug reports versus bug reports + buggy code across prompting strategies.

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

Table 4: BERTScore for CodeLlama using only buggy code to assess code-alone contribution.

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

As reported in Table 1, extractive models underperform on this dataset due to their inability to paraphrase and synthesize novel expressions. BugSum achieved ROUGE-1 = 0.2591, while DeepSum scored 0.1760. In contrast, our models produced more concise and coherent summaries, with Mistral achieving ROUGE-1 = 0.2786, CodeLlama = 0.2641, and Gemma = 0.2016. Although abstractive baselines fine-tuned on the Fang corpus attained higher ROUGE scores—SUMLLAMA (ROUGE-1 = 0.4426, ROUGE-L = 0.4101), KSCLP (ROUGE-1 = 0.4133, ROUGE-L = 0.3789), and RTA (ROUGE-1 = 0.3919, ROUGE-L = 0.3597)—our models maintained strong semantic fidelity, as reflected by BERTScore results in Table 2.

Quantitatively, the average BERTScore of our models over N evaluation instances consistently exceeded that of extractive baselines demonstrating that even when ROUGE scores are surpassed by fine-tuned abstractive methods, our approach preserves semantic coherence and conciseness.

Table 5: BERTScore for CodeLlama using bug reports + patch code, highlighting the effect of corrected code snippets.

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

Table 6: BERTScore comparing input sequences (Code → Bug Report vs Bug Report → Code) using CodeLlama.

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

Answer to RQ1: *Our approach generates concise, coherent summaries that outperform extractive baselines and achieve strong semantic performance, remaining competitive with fine-tuned abstractive models.*

4.2 Impact of Code Context (RQ2)

We evaluated eight LLMs on the Defects4J dataset [16] under two input conditions: using bug reports alone, and using bug reports combined with associated code summaries. The inclusion of code information generally led to modest improvements in few-shot F1 scores. For example, GPT-3.5 Turbo increased from 0.8595 to 0.9003 when code was included, while Mistral showed a decrease from 0.6928 to 0.6509 in certain configurations, as reported in Tables 3–5.

Ablation studies further highlight the influence of code type and input ordering. Using only buggy code yields low F1 performance (0.4767, Table 4), whereas incorporating patch code provides a modest increase (0.6368, Table 5). The sequence in which inputs are presented to the model also affects results: presenting the bug report first followed by code (Bug Report → Code) achieves F1 = 0.6337, while the reverse order (Code → Bug Report) decreases performance to 0.6087, as shown in Table 6. These findings indicate that while code can enhance summary quality, bug-report text remains the dominant source of information, and careful consideration of code type and prompt structure is necessary to maximize benefits.

Answer to RQ2: *Code integration yields modest gains (F1 up to 0.9003). Bug reports remain primary input; patch code and input order can further influence results. Current metrics may not fully capture qualitative improvements.*

4.3 Effect of Prompt Engineering vs Fine-Tuning (RQ3)

We analyzed the relative impact of prompt engineering and fine-tuning across multiple datasets. On extractive datasets such as SDS [13] and ADS [14], fine-tuning resulted in moderate precision values between 0.5 and 0.6, but recall remained relatively low (0.35–0.45). Incorporating code information in these settings provided minimal additional benefit, indicating that extractive benchmarks are largely driven by textual content in the bug reports.

On abstractive datasets, including Defects4J and Fang et al.’s corpus, fine-tuned models exhibited modest gains. For example, Mistral achieved F1 = 0.6202 without code and 0.6170 with code, while CodeLlama scored 0.5991 without code and 0.6010 with code. Interestingly, zero-shot prompt engineering often matched or outperformed fine-tuned models: GPT-3.5 Turbo achieved F1 = 0.7640 on Defects4J and 0.6394 on the Fang dataset, as summarized in Table 2. These results suggest that for abstractive summarization, carefully designed prompts can leverage pre-trained knowledge effectively, reducing the need for extensive fine-tuning, particularly when training data is limited.

Answer to RQ3: *Fine-tuning yields moderate improvements on extractive datasets but marginal gains on abstractive datasets; prompt engineering remains competitive, particularly for small training datasets.*

5 Limitations & Future Work

This study has several limitations. First, the primary dataset, Defects4J, is relatively small, which constrains the diversity of bug-report and code patterns available for model training and evaluation. Second, there is a scarcity of publicly available datasets that provide both bug reports and corresponding code, limiting opportunities for extensive multimodal experimentation. Third, our hierarchical aggregation of long code snippets may result in partial information loss, potentially affecting the fidelity of generated summaries. Fourth, although our prompt engineering strategies were designed to be structured and multi-modal, they are not necessarily optimal, and alternative prompt formulations may yield improved performance.

Future work will address these limitations in several directions. Human evaluation of generated summaries is needed to capture qualitative aspects such as clarity, technical accuracy, and usefulness to developers. Integration of our approach into real-world software maintenance tools could provide practical feedback and validate model utility in situ. Finally, the development of evaluation metrics specifically tailored to bug-report summarization [40] would enable more precise and domain-relevant assessment of model performance, particularly for capturing semantic fidelity and actionable content beyond standard text-similarity metrics.

6 Conclusion

We present a multi-stage LLM-based pipeline for abstractive bug-report summarization that effectively integrates both textual descriptions and associated code snippets. Our approach outperforms extractive baselines, achieving ROUGE improvements ranging from +7.5% to +58.2%, while maintaining strong semantic fidelity as measured by BERTScore. Experiments demonstrate that carefully designed prompt engineering can, in some cases, rival or even exceed the benefits of fine-tuning, particularly on limited training data. Despite these promising results, the relatively small size of available datasets and limitations of current evaluation metrics suggest that performance should be interpreted cautiously. Future work targeting human evaluation, larger multimodal datasets, and task-specific metrics will further clarify the practical utility and robustness of LLM-based bug-report summarization.

References

- [1] Weiqin Zou, David Lo, Zhenyu Chen, Xin Xia, Yang Feng, and Baowen Xu. How practitioners perceive automated bug report management techniques. *IEEE Transactions on Software Engineering*, 46(8):836–862, 2020.
- [2] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. What makes a good bug report? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, SIGSOFT ’08/FSE-16*, page 308–318, New York, NY, USA, 2008. Association for Computing Machinery.
- [3] Xiaochen Li, He Jiang, Dong Liu, Zhilei Ren, and Ge Li. Unsupervised deep bug report summarization. In *Proceedings of the 26th Conference on Program Comprehension, ICPC ’18*, page 144–155, New York, NY, USA, 2018. Association for Computing Machinery.
- [4] Wafaa S. El-Kassas, Cherif R. Salama, Ahmed A. Rafea, and Hoda K. Mohamed. Automatic text summarization: A comprehensive survey. *Expert Systems with Applications*, 165:113679, 2021.
- [5] Wojciech Kryscinski, Bryan McCann, Caiming Xiong, and Richard Socher. Evaluating the factual consistency of abstractive text summarization. In Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu, editors, *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 9332–9346, Online, November 2020. Association for Computational Linguistics.
- [6] Haopeng Zhang, Philip S. Yu, and Jiawei Zhang. A systematic survey of text summarization: From statistical methods to large language models, 2024.
- [7] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

- [8] Preetha Chatterjee, Manziba Akanda Nishi, Kostadin Damevski, Vinay Augustine, Lori Pollock, and Nicholas A. Kraft. What information about code snippets is available in different software-related documents? an exploratory study. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 382–386, 2017.
- [9] Silvia Breu, Rahul Premraj, Jonathan Sillito, and Thomas Zimmermann. Information needs in bug reports: improving cooperation between developers and users. In *Proceedings of the 2010 ACM Conference on Computer Supported Cooperative Work, CSCW '10*, page 301–310, New York, NY, USA, 2010. Association for Computing Machinery.
- [10] Senthil Mani, Anush Sankaran, and Rahul Aralikatte. Deeptriage: Exploring the effectiveness of deep learning for bug triaging. In *Proceedings of the 40th ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 602–614. IEEE, 2018. Accessed: 2025-08-31.
- [11] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems, NIPS '20*, Red Hook, NY, USA, 2020. Curran Associates Inc.
- [12] Zhengbao Jiang, Frank F. Xu, Jun Araki, and Graham Neubig. How can we know what language models know? *Transactions of the Association for Computational Linguistics*, 8:423–438, 2020.
- [13] Sarah Rastkar, Gail C. Murphy, and Gabriel Murray. Summarizing software artifacts: a case study of bug reports. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE '10*, page 505–514, New York, NY, USA, 2010. Association for Computing Machinery.
- [14] He Jiang, Jingxuan Zhang, Hongjing Ma, Najam Nazar, and Zhilei Ren. Mining authorship characteristics in bug repositories. *Science China Information Sciences*, 60(1):012107, 2017. Accessed: 2025-08-31.
- [15] Sen Fang, Tao Zhang, Youshuai Tan, He Jiang, Xin Xia, and Xiaobing Sun. Representthemall: A universal learning representation of bug reports. In *Proceedings of the 45th International Conference on Software Engineering, ICSE '23*, page 602–614. IEEE Press, 2023.
- [16] 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)*, pages 437–440. ACM, 2014.
- [17] Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q. Weinberger, and Yoav Artzi. Bertscore: Evaluating text generation with bert, 2020.
- [18] Mahak Gambhir and Vishal Gupta. Recent automatic text summarization techniques: a survey. *Artif. Intell. Rev.*, 47(1):1–66, January 2017.
- [19] Hassan Shakil, Ahmad Farooq, and Jugal Kalita. Abstractive text summarization: State of the art, challenges, and improvements. *Neurocomputing*, 603:128255, October 2024.
- [20] 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.
- [21] B. Xiang and Y. Shao. Sumllama: Efficient contrastive representations and fine-tuned adapters for bug report summarization. *IEEE Access*, 12:78562–78571, 2024.
- [22] Najam Nazar, Yan Hu, and He Jiang. Summarizing software artifacts: A literature review. *Journal of Computer Science and Technology*, 31(5):883–909, September 2016.
- [23] E. G. Santana Jr, Gabriel Benjamin, Melissa Araujo, Harrison Santos, David Freitas, Eduardo Almeida, Paulo Anselmo da M. S. Neto, Jiawei Li, Jina Chun, and Iftekhar Ahmed. Which prompting technique should i use? an empirical investigation of prompting techniques for software engineering tasks, 2025.
- [24] S. Kang, J. Yoon, and S. Yoo. Large language models are few-shot testers: Exploring llm-based general bug reproduction. In *Proceedings of IEEE/ACM International Conference on Software Engineering (ICSE)*, 2023.
- [25] Lili Bo, Wangjie Ji, Xiaobing Sun, Ting Zhang, Xiaoxue Wu, and Ying Wei. Chatbr: Automated assessment and improvement of bug report quality using chatgpt. In *2024 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1472–1483, 2024.
- [26] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. Enhancing static analysis for practical bug detection: An llm-integrated approach. *Proc. ACM Program. Lang.*, 8(OOPSLA1), April 2024.

- [27] Peiyu Liu, Junming Liu, Lirong Fu, Kangjie Lu, Yifan Xia, Xuhong Zhang, Wenzhi Chen, Haiqin Weng, Shouling Ji, and Wenhui Wang. Exploring chatgpt’s capabilities on vulnerability management. In *Proceedings of the 33rd USENIX Conference on Security Symposium, SEC ’24*, USA, 2024. USENIX Association.
- [28] Victor Sobreira, Thomas Durieux, Fernanda Madeiral, Martin Monperrus, and Marcelo A. Maia. Dissection of a Bug Dataset: Anatomy of 395 Patches from Defects4J. In *Proceedings of SANER*, 2018.
- [29] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models, 2021.
- [30] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémie Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code, 2024.
- [31] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.
- [32] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mistral 7b, 2023.
- [33] Marah Abdin, Jyoti Aneja, Harkirat Behl, Sébastien Bubeck, Ronen Eldan, Suriya Gunasekar, Michael Harrison, Russell J Hewett, Mojgan Javaheripi, Piero Kauffmann, et al. Phi-4 technical report. *arXiv preprint arXiv:2412.08905*, 2024.
- [34] Gemma Team, Thomas Mesnard, Cassidy Hardin, Robert Dadashi, Surya Bhupatiraju, Shreya Pathak, Laurent Sifre, Morgane Rivière, Mihir Sanjay Kale, Juliette Love, et al. Gemma: Open models based on gemini research and technology. *arXiv preprint arXiv:2403.08295*, 2024.
- [35] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengan Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.
- [36] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. Deepseek-coder: When the large language model meets programming – the rise of code intelligence, 2024.
- [37] OpenAI. Gpt-3.5 turbo. <https://platform.openai.com/docs/models/gpt-3.5-turbo>, 2023. Accessed: 2025-09-11.
- [38] Chin-Yew Lin. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pages 74–81, Barcelona, Spain, July 2004. Association for Computational Linguistics.
- [39] X. Li, J. Zhang, H. Ma, N. Nazar, and Z. Ren. Deep context understanding for bug report summarization. In *Proceedings of the 22nd International Conference on Program Comprehension (ICPC)*, pages 1–10. IEEE, 2018. Accessed: 2025-08-31.
- [40] Guoming Long, Jingzhi Gong, Hui Fang, and Tao Chen. Learning software bug reports: A systematic literature review, 2025.