applied sciences

MDPI

Article

CTQRS-Based Reinforcement Learning Framework for Reliable Bug Report Generation Using Open-Source Large Language Models

Geunseok Yang *

Department of Computer Applied Mathematics, Computer System Institute, Hankyong National University, Anseong 17579, Republic of Korea
* Correspondence: gsyang@hknu.ac.kr

Abstract

The advancement of Large Language Models (LLMs) has opened new possibilities for automating bug report generation in software engineering. However, a fundamental limitation remains: the generated reports often fail to maintain both consistent structure and reliable semantic quality. To address this issue, this study proposes a Reinforcement Learning (RL) framework that integrates the CTQRS (Completeness, Traceability, Quality, Reproducibility, Specificity) metric as a reward signal. The proposed method aims to enhance both the structural completeness and semantic coherence of generated reports, enabling the automatic creation of reliable bug reports based on open-source LLMs. The training process consists of three stages: Supervised Fine-Tuning (SFT), Reinforcement Learning (RL), and Refinement. In the SFT stage, the model learns the formal structure of bug reports, reducing the loss from 1.9 to 1.3 and achieving initial CTQRS and SBERT scores of 0.46 and 0.68, respectively. In the RL stage, a multi-reward function centered on CTQRS is combined with the Proximal Policy Optimization (PPO) algorithm, increasing the reward value from 0.42 to 0.63 with stable convergence confirmed through the Exponential Moving Average (EMA). During this process, the CTQRS and SBERT scores improved to 0.72 and 0.84, demonstrating that the model simultaneously enhanced structural completeness and semantic consistency. In the final Refinement stage, the outcomes of SFT and RL are integrated, and a critic-based fine-grained feedback adjustment strategy is applied to stabilize the final outputs. The refined reports maintained a reward value of approximately 0.65, achieving peak CTQRS and SBERT scores of 0.76 and 0.85, respectively. Throughout the entire training process, the stability of the reward gradients was preserved, and the adjustments to length rewards and repetition penalties effectively prevented excessive verbosity. Experimental results show that the proposed CTQRS-based reinforcement learning framework improves the structural completeness, contextual accuracy, and evaluation stability of bug reports, thereby quantitatively enhancing the reliability of LLM-based (v.Qwen2.5-7B-Instruct) software quality assurance documentation. Future work will focus on further improving formal precision and evaluation consistency by fine-tuning the number of critic iterations (critic_iters) and adjusting detailed reward weights.

Keywords: CTQRS-guided reinforcement learning; large language models; bug report generation; Proximal Policy Optimization; software quality assurance

check for updates

Academic Editors: Stefan Fischer and Douglas O'Shaughnessy

Received: 19 October 2025
Revised: 7 November 2025
Accepted: 24 November 2025
Published: 26 November 2025

Citation: Yang, G. CTQRS-Based Reinforcement Learning Framework for Reliable Bug Report Generation Using Open-Source Large Language Models. *Appl. Sci.* **2025**, *15*, 12545. https://doi.org/10.3390/app152312545

Copyright: © 2025 by the author. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Bug reports are essential artifacts for identifying and tracking defects throughout software development and maintenance processes [1]. However, in real-world projects, their quality often varies significantly depending on the reporter's expertise and writing style [2]. In particular, open-source projects and large-scale collaborative environments frequently involve non-expert contributors, leading to reports that are written in free formats or missing required fields. Such incomplete reports complicate the reproduction and resolution of defects, reducing maintenance efficiency and negatively impacting both software quality and productivity. Therefore, automating the structuring of unformatted reports and supplementing missing information has long been an important research topic in software engineering. To clarify, the goal of this study is to structurally organize and refine user-provided bug reports rather than to reconstruct or infer information that is not supplied by the user.

Recent advances in Large Language Models (LLMs) have opened new possibilities for automating bug report quality improvement. Models such as GPT [3], LLaMA [4], and Qwen [5] have shown strong capabilities in contextual understanding and natural language generation through large-scale text learning. With proper prompt design and fine-tuning techniques, these models can generate reports that adhere to given structural or stylistic constraints. In particular, few-shot learning [6] and the Chain-of-Thought (CoT) reasoning strategy [7] enable LLMs to learn complex narrative structures from only a few examples, establishing them as key techniques in automated report generation. However, most existing studies have relied on template-based generation or supervised fine-tuning (SFT), which fail to incorporate quality indicators such as CTQRS (Completeness, Traceability, Quality, Reproducibility, Specificity) into the learning process [8]. Moreover, evaluation has often depended on lexical metrics such as ROUGE-1 Recall, which insufficiently capture semantic precision and coherence [9].

To address these limitations, this study proposes a CTQRS-based Reinforcement Learning (RL) framework [10]. The proposed approach explicitly integrates report quality metrics as learning objectives, allowing the model to evaluate and improve its own generation outputs according to these metrics. The training process consists of three stages. First, during the Supervised Fine-Tuning (SFT) stage, the model learns the basic format and sentence structure of bug reports. In the subsequent RL stage, the Proximal Policy Optimization (PPO) algorithm [11] is used to optimize a composite reward function that combines CTQRS, ROUGE, and SBERT (Sentence-BERT) metrics [12], thereby improving both structural completeness and semantic coherence. Finally, in the refinement stage [13], the model further enhances the linguistic precision and document consistency of its outputs based on the RL results. This staged learning process enables the model to move beyond mere imitation of training data and acquire a quality-oriented generation policy [14].

This three-stage structure is designed not for procedural convenience but for stability and quality alignment. The supervised fine-tuning stage provides the model with structural and grammatical priors that stabilize the reward landscape during reinforcement learning. Direct RL from an uninitialized LLM was observed to produce incoherent or incomplete reports due to the absence of these priors. The refinement stage further harmonizes structural fidelity with linguistic precision, preventing reward bias and ensuring balanced improvement in both form and content.

Experiments conducted on 3966 bug reports collected from Bugzilla [15] show that the proposed model significantly outperforms the supervised learning baseline, improving the CTQRS score from 0.46 to 0.76 and the SBERT similarity from 0.68 to 0.85. The PPO training process exhibited stable convergence under the Exponential Moving Average (EMA) scheme, and the refinement stage maintained a reward value around 0.65, showing

improvements in both structural completeness and linguistic accuracy. These results indicate that the proposed framework advances beyond reproducing report templates, enabling autonomous generation that internalizes and reflects quality evaluation criteria.

The major contributions of this study are summarized as follows:

- A CTQRS-based RL framework was designed to enable the model to learn both structural completeness and semantic consistency of reports. This allows direct optimization of report quality during training.
- A multi-reward function integrating CTQRS, ROUGE, and SBERT metrics was proposed to balance structural completeness, lexical fidelity, and semantic coherence. Additional length rewards and repetition penalties were incorporated to quantitatively control expression quality.
- A staged learning structure combining supervised and reinforcement learning was implemented. After acquiring basic structure through SFT, the model optimizes quality metrics via RL and further refines precision and stability in the final stage.
- Experiments on the Bugzilla dataset verified consistent improvements across CTQRS, ROUGE, and SBERT metrics compared to the baseline, demonstrating the practical contribution of RL to bug report quality enhancement.

By incorporating quality metrics not merely as evaluation criteria but as core learning objectives, this study introduces a new direction for automated bug report generation. The proposed framework extends the language generation capabilities of LLMs into a quality-driven learning paradigm, enabling efficient and reproducible report generation that improves completeness and coherence without human feedback. This work provides a solid foundation for the automation of software quality management and the development of AI-assisted maintenance systems.

2. Background Knowledge

2.1. Bug Report Quality and the Need for Automation

In the process of software development and maintenance, bug reports serve as a critical medium that connects defect identification, reproduction, correction, and verification. However, in real-world practice, the quality of bug reports varies significantly depending on the author's expertise, technical understanding, and writing style. In open-source projects or collaborative development environments, non-expert users often submit reports, resulting in frequent cases where essential information is omitted or the reporting format is inconsistent. Such incomplete reports make it difficult to identify the root cause of problems and lead to unnecessary consumption of time and human resources during defect reproduction and resolution. Consequently, the efficiency of project quality management decreases, creating a vicious cycle that increases maintenance costs [1,2].

The degradation of bug report quality generally manifests in two dimensions. First, structural incompleteness refers to the absence of essential information or unclear formatting. For instance, when a report fails to clearly describe the Steps to Reproduce or does not properly distinguish between Expected Behavior and Actual Behavior, developers struggle to replicate the defect. Second, semantic inconsistency arises when ambiguity or lack of contextual clarity prevents the intended meaning of the report from being accurately conveyed. In such cases, even a formally complete report may fail to help developers fully understand the nature of the issue [1].

Traditional approaches to address these quality issues have relied on template-based reporting and checklist-driven validation rules. While these methods enforced a certain level of consistency by requiring adherence to predefined formats, they still depended heavily on the author's linguistic skills and technical comprehension, making it difficult to

guarantee fundamental quality. Moreover, manual validation is both time-consuming and costly, making it impractical as a scalable quality assurance method in large-scale projects.

Against this backdrop, recent research has increasingly focused on automating the structuring and quality refinement of bug reports. The rapid advancement of large language models (LLMs) has opened new possibilities for automatically analyzing and structuring reports written in natural language [3–5]. Through extensive text training, LLMs have acquired strong contextual understanding and narrative generation capabilities. By leveraging these capabilities, it becomes possible to transform unstructured inputs into structured formats and to supplement missing information. This technological progress has introduced a new direction for automating bug report quality management and underscored the need for quantitative approaches that simultaneously ensure completeness, reproducibility, and semantic coherence [8].

The quality of bug reports represents more than an issue of document formatting; it is a fundamental factor that directly influences the efficiency and reliability of software maintenance. As a result, developing an automated reporting framework that ensures both structural completeness and semantic consistency has emerged as a crucial area of research in modern software engineering [1,2].

2.2. LLM-Based Automated Bug Report Generation

The advancement of large language models (LLMs) has opened new possibilities for automatically structuring and refining unstructured bug reports. Models such as GPT, LLaMA, and Qwen [3–5] have been trained on vast amounts of text data, enabling them to understand context and generate coherent natural language. Through prompt design and supervised fine-tuning (SFT), these models can generate reports that conform to given constraints [14,16,17]. They are effectively utilized to extract key information from input reports or supplement missing content while maintaining sentence coherence and natural expression.

For this reason, recent studies have explored various approaches that leverage LLMs to automatically generate or summarize bug reports. Representative methods include learning the structural patterns of reports through SFT or designing specific prompts to automatically generate elements such as reproduction steps or expected results. Some research has focused on automating the generation of summaries, defect classification, or textual refinement based on LLMs to improve the quality of bug reports [13]. These approaches have gained attention for overcoming the limitations of traditional template-based systems and producing more flexible and contextually rich reports.

However, LLM-based approaches still face fundamental challenges. Because these models primarily learn linguistic patterns, they often achieve formal completeness but fail to incorporate the technical details or logical relationships required for actual defect reproduction. Although the generated reports may appear grammatically sound, they frequently lack clear explanations of defect causes or symptoms, limiting their practical utility in real maintenance scenarios. Moreover, evaluation metrics are often biased toward lexical similarity, making it difficult to quantitatively assess the real usefulness and quality of reports. Most prior studies have relied on lexical-based metrics such as ROUGE [9] and BLEU [18], which are insufficient for simultaneously measuring both the structural completeness and semantic coherence of reports.

Research on automatic bug report generation using LLMs faces its greatest challenge in defining and incorporating quality into the learning process. Existing approaches have primarily emphasized reducing lexical differences from reference sentences, which limits the model's ability to learn quality attributes such as completeness and reproducibility [1,8].

As a result, the generated reports may appear linguistically well-structured, yet their reliability and practical usability in real maintenance environments remain insufficient.

Recently, there has been growing recognition of the need for quantitative metrics that can evaluate report quality from multiple perspectives. In particular, CTQRS has emerged as a quality assessment framework capable of considering both structural completeness and semantic coherence rather than simple lexical similarity [8]. This approach provides a crucial foundation for quantitatively controlling the quality of LLM-generated bug reports and enhancing them to a level that is practically applicable in real maintenance processes.

2.3. Bug Report Quality Metrics and the CTQRS Framework

To objectively assess the quality of bug reports, a multidimensional approach is required that goes beyond simple linguistic similarity or word frequency–based evaluation. Such an approach must simultaneously consider both the structural completeness and semantic coherence of a report. Existing evaluation metrics have failed to fully capture these composite aspects of quality, which limits their ability to precisely measure the performance of automated report generation models or provide meaningful directions for improvement.

Representative metrics commonly used in prior studies include lexical-based measures such as ROUGE [9] and BLEU [18]. ROUGE calculates the overlap of words and *n*-grams between the reference report and the generated report, making it useful for assessing lexical similarity. In particular, ROUGE-1 Recall measures how comprehensively the generated report covers the key words of the original report, thereby quantifying the extent to which essential information is preserved. However, the ROUGE family of metrics does not reflect contextual meaning and relies solely on the presence of identical words. As a result, it cannot accurately capture true quality differences when sentence structures differ or word choices vary.

To overcome this limitation, recent studies have introduced semantic-based evaluation methods using SBERT embeddings [12]. SBERT converts sentences into embedding vectors and calculates cosine similarity, allowing it to quantify whether two sentences convey the same meaning even when their lexical forms differ. A higher SBERT score indicates that the generated report semantically aligns with the reference report, effectively measuring logical structure and sentence-level expression quality. Nevertheless, SBERT still has limitations in evaluating structural relationships among sections or the overall formal consistency of a report.

To address these shortcomings, the CTQRS-based evaluation framework was proposed. CTQRS integrates five key attributes that collectively define bug report quality, offering a qualitative and quantitative metric that reflects both structural completeness and reproducibility [1,8].

- Completeness evaluates whether all essential components, such as the summary, reproduction steps, expected results, actual results, and environment information are included.
- Traceability measures whether execution contexts, configuration details, and version information are sufficiently specified to enable follow-up actions.
- Quality assesses clarity of description, syntactic consistency, and minimization of redundancy, relating to readability and precision of expression.
- Reproducibility evaluates whether the reported issue can be consistently reproduced based on the provided information, which is critical for effective debugging.
- Specificity measures whether the report contains concrete evidence such as symptoms, file paths, logs, and error codes rather than vague expressions.

CTQRS is significant in that it integrates both formal completeness and substantive usefulness, which are often missed by traditional lexical-based metrics. By independently scoring each component and aggregating them with weighted sums, CTQRS quantitatively

captures how omissions or imbalances in specific elements affect the overall report quality. This enables joint verification of both the structural completeness and practical applicability of automatically generated bug reports.

The CTQRS framework serves not only as an evaluation metric but also as a reference standard for managing the quality of bug reports. In this study, CTQRS is incorporated as a reward function within the training process, aligning the model's learning objective directly with the evaluation criteria. Through this integration, the model learns to optimize both structural completeness and semantic coherence rather than focusing solely on linguistic similarity. This design provides a foundation for quantitatively controlling the quality of automated bug report generation and enabling its practical use in real-world software maintenance environments.

2.4. Reinforcement Learning for Quality Optimization and Position of This Study

Research on automated bug report generation using large language models (LLMs) has advanced rapidly in recent years. Most existing approaches rely on supervised fine-tuning (SFT) or prompt-based generation [14]. These methods are effective in learning the typical document structure and linguistic patterns of reports. They do not, however, internalize the ability to evaluate or improve the quality of the generated reports. Conventional models reproduce statistical patterns from given data and remain limited in achieving the structural completeness and semantic consistency required in practical defect reports.

Reinforcement learning (RL) has emerged as a quality-oriented learning paradigm that can overcome these limitations [10]. Supervised learning passively trains models using labeled data, while reinforcement learning actively evaluates model outputs through a reward function and optimizes the policy based on the feedback. The model learns to distinguish "good reports" from "poor reports" through reward signals. The Proximal Policy Optimization (PPO) algorithm stabilizes this optimization by preventing abrupt policy changes and enables gradual performance improvement. PPO has become one of the most widely used algorithms in natural language generation tasks [11].

This study introduces the structural advantages of reinforcement learning to the task of bug report generation. Unlike conventional Reinforcement Learning with Human Feedback (RLHF) approaches [14], this work employs quantitative quality metrics directly as reward functions, enabling autonomous quality-driven learning. The reward function integrates CTQRS [8] with supplementary evaluation metrics such as ROUGE [9] and SBERT [12]. CTQRS captures the formal completeness and technical specificity of a report, while ROUGE and SBERT complement lexical fidelity and semantic coherence. This metric-based reward structure encourages the model to consider both structural integrity and informational utility rather than merely imitating surface patterns.

The reinforcement learning framework contributes to improving three dimensions of report quality simultaneously. The model internalizes structural completeness by repeatedly evaluating core report sections—summary, reproduction steps, expected results, and actual results—during training. The inclusion of semantic indicators in the reward signal enhances contextual coherence and clarity of technical descriptions. The interaction between reward and policy naturally suppresses redundancy and verbosity, resulting in higher readability and information efficiency. Through this process, the model evolves from a simple text generator into an autonomous system that learns quality evaluation criteria as part of its generation objective.

The proposed approach addresses the structural constraints and semantic instability that have limited prior LLM-based report generation studies. Supervised learning alone cannot achieve quality improvement beyond the patterns contained in data. By integrating reinforcement learning, the model iteratively evaluates and refines its outputs, achieving

a more quality-oriented learning process. The model first establishes linguistic stability through supervised training, strengthens quality criteria through reinforcement learning, and enhances accuracy and consistency of expressions in a final refinement phase [13,14]. This stepwise training structure enables quality-centered report generation that a single learning method cannot attain. The proposed framework serves as a practical solution that advances both structural completeness and semantic consistency in automated bug report generation.

3. Related Work

In real-world software development, bug reports serve as essential evidence for reproducing and resolving defects. Persistent issues have been raised regarding their inconsistent quality, largely due to differences in the author's writing ability and style. Even when describing the same defect, authors often use distinct narrative structures and terminology, leading to semantic inconsistencies across reports. Such irregularity and expressive diversity reduce comprehension and reproducibility, ultimately hindering the efficiency of maintenance activities. To address these problems, researchers have explored approaches that learn the contextual meaning of reports or extract and summarize key sentences to form structured representations. These efforts aim to enhance readability and reproducibility while minimizing information loss during maintenance.

Fang et al. [19] proposed a model that predicts bug-fixing priorities using a Weighted Graph Convolutional Network and showed that representational features of reports can improve maintenance efficiency. Fang et al. [20] later introduced RepresentThemAll, a universal representation learning framework that captures both contextual and semantic characteristics of bug reports, thereby expanding the general applicability of report embeddings. Liu et al. [21] presented BugSum, a deep learning-based summarization model for extracting essential context from lengthy reports, and Shao et al. [22] implemented a domain-specific summarization model optimized for software engineering through specialized representation learning. These studies advanced the understanding of structural and semantic features in bug reports but largely focused on static representation learning or extractive summarization. As a result, quality aspects such as completeness and reproducibility were not directly integrated into training, and the quality of generated reports remained dependent on post-hoc evaluation.

Automation in bug triage and management has also been explored to improve the utility and effectiveness of bug reports. Lamkanfi et al. [23] analyzed textual content to automatically predict defect severity, and Sarkar et al. [24] enhanced classification accuracy in industrial environments (e.g., Ericsson) by applying confidence-driven triage models. Medeiros et al. [25] employed crash report mining to locate defect positions automatically, providing empirical evidence that report quality directly influences maintenance efficiency. These studies improved automation in downstream processes but assumed a consistent input quality in bug reports. When the input quality is poor, subsequent triage or management tasks perform less accurately, leaving the fundamental issue unresolved.

Advances in large language models (LLMs) have enabled structured and coherent text generation. Chen et al. [26] evaluated code-focused models such as Codex and confirmed that large-scale pretraining significantly improves code generation performance. Ben Allal et al. [27] proposed StarCoder, trained on large open-source code datasets, and Rozière et al. [28] introduced CodeLLaMA, a model specialized for code comprehension and generation. Luo et al. [29] developed WizardCoder, which applied command-based fine-tuning to improve both fluency and reasoning in code generation. These findings illustrate that LLMs can internalize structural patterns and logical consistency through large-scale

learning. However, most models continue to emphasize linguistic fluency and grammatical correctness rather than structural completeness or explicit quality objectives.

Research on self-refinement and feedback-based improvement has gained increasing attention. Chen et al. [30] proposed the Self-Debug approach, which allows models to identify and correct their own errors during code generation. Gou et al. [31] introduced CRITIC, integrating external tools with critical feedback to iteratively refine model outputs. These approaches enhanced self-evaluation and self-improvement capabilities, yet they primarily relied on heuristic feedback or unstructured corrections. Methods that incorporate quantitative quality metrics as direct learning signals remain insufficiently developed.

Evaluation methodologies have shifted from lexical similarity toward semantic and functional perspectives. Zhang et al. [32] proposed BERTScore, which uses contextual embeddings to quantify semantic similarity and evaluate meaning alignment in text. Eghbali et al. [33] introduced CrystalBLEU for fine-grained assessment of code generation, and Zhou et al. [34] developed CodeBERTScore to capture semantic correspondence between code and natural language. Wang et al. [35] proposed evaluating functional correctness based on execution results, while Lu et al. [36] constructed CodeXGLUE, a benchmark that standardized such evaluation frameworks. These studies advanced the assessment of semantic alignment in text and code generation but did not sufficiently address quality dimensions such as formal completeness or informational specificity. Evaluation metrics have become more refined, yet systematic mechanisms that utilize them as learning objectives remain undeveloped.

Research across representation learning, triage automation, LLM-based generation, self-refinement, and evaluation design has collectively enhanced the efficiency of software maintenance. Despite these achievements, most studies have not integrated report quality as a direct learning goal. Prior work generally treated report quality as a post-training evaluation measure rather than a feature that the model could recognize and optimize during training.

This study introduces a quality-oriented reinforcement learning framework that incorporates CTQRS metrics into the reward function. The framework enables models to evaluate and refine structural completeness and semantic coherence beyond grammatical correctness through self-improving quality learning. The approach moves beyond imitation learning and presents a new direction for enhancing reliability and reproducibility in automated bug report generation.

4. Methodology

The CTQRS-based reinforcement learning framework proposed in this study is designed to automatically generate reliable and structurally complete bug reports by leveraging the representational capabilities of open-source large language models. The overall architecture of the proposed system is summarized in Figure 1.

Figure 1 illustrates the end-to-end workflow of the proposed framework in four layers. The top layer shows the data preprocessing process, where approximately 15,000 raw Bugzilla reports are filtered through rule-based verification and CTQRS scoring to construct a structured dataset of 3966 high-quality entries. The middle layer represents the three-stage training pipeline composed of supervised fine-tuning (SFT), reinforcement learning (RL), and self-critic refinement. Each stage progressively enhances structural completeness and semantic coherence. The lower-left section describes the CTQRS scoring and reward integration logic, where five subcomponents (Completeness, Traceability, Quality, Reproducibility, and Specificity) are aggregated into a composite score and combined with ROUGE and SBERT metrics to form the final reward. The lower-center section depicts the PPO optimization flow, in which the Policy Model, RewardAgent, and Value Model

interact to update the policy through advantage estimation and EMA-based stabilization. Finally, the lower-right section presents the comparative effects of the three training stages, highlighting incremental improvements in CTQRS (0.46 → 0.72 → 0.76), SBERT (0.68 → 0.84 → 0.85), and ROUGE (0.52 → 0.61 → 0.63) as well as qualitative refinements in report clarity and completeness.

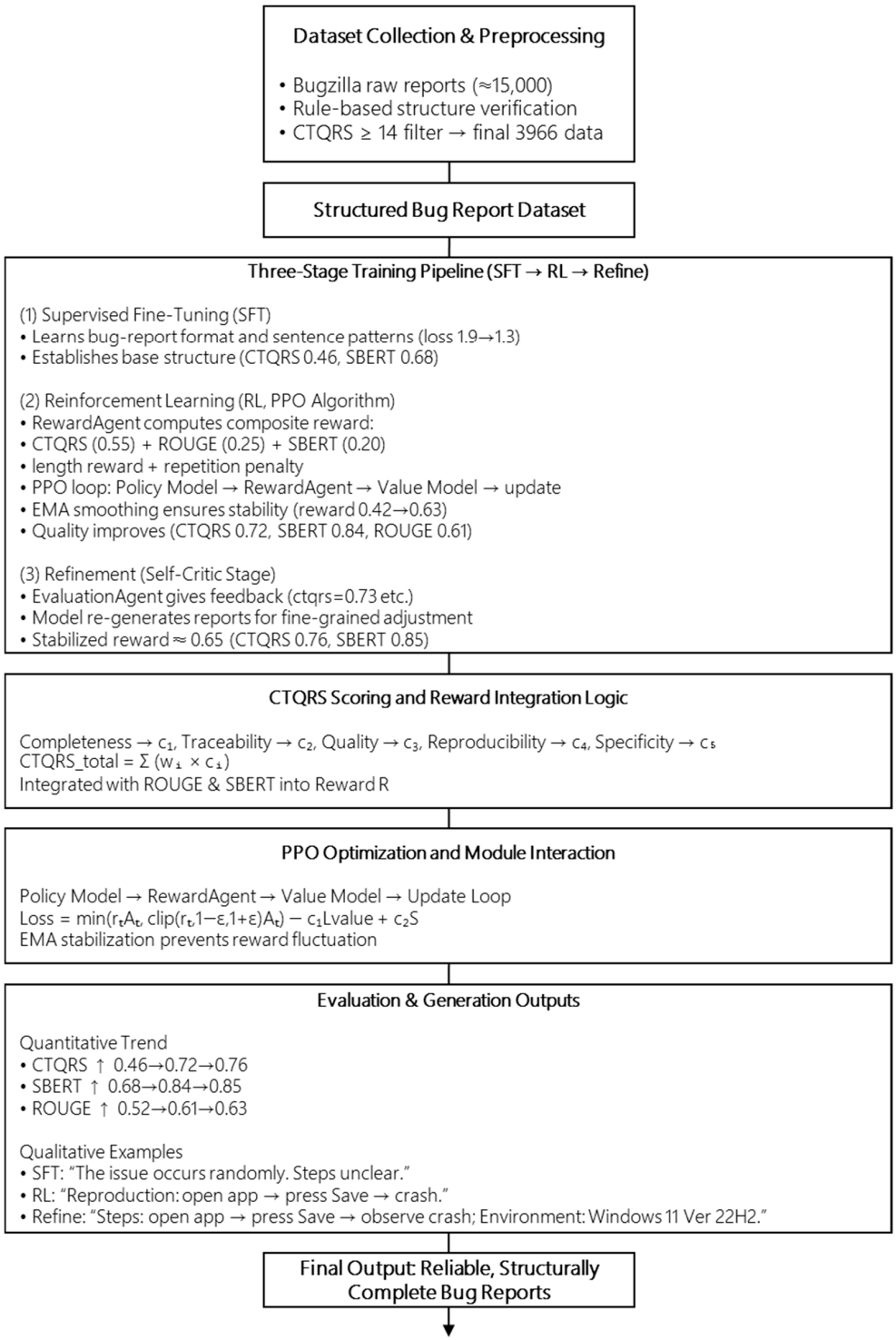

Figure 1. Integrated overview of the proposed CTQRS-guided reinforcement learning framework, including the data preprocessing pipeline, CTQRS scoring logic, PPO optimization flow, and the three-stage training (SFT → RL → Refinement) with comparative performance trends.

The framework defines the quality of bug reports in quantitative terms and incorporates this evaluation directly into the training process. This approach addresses the structural instability and semantic inconsistency that often occur in text generation-based models. The system consists of three stages: supervised fine-tuning (SFT), reinforcement learning (RL), and refinement.

In the initial SFT stage, the model learns the formal conventions and narrative structure of bug reports, internalizing the fundamental format of the documents. In the reinforcement learning stage, the CTQRS metric serves as a reward signal, enabling the model to improve the completeness, reproducibility, specificity, and consistency of its generated reports. Mechanisms such as reward normalization, repetition suppression, and length adjustment are employed to ensure stable policy convergence during training.

After reinforcement learning, a critic-based refinement stage adjusts the logical coherence and linguistic fluency of the generated reports. This process minimizes overfitting and contextual distortion that may arise during reward-driven learning, enhancing both readability and stylistic consistency.

As a result, the proposed framework functions not merely as a generative model but as an autonomous learning system capable of evaluating and improving its own outputs based on quality indicators. The reports produced through this approach exhibit a level of structural clarity and technical reliability comparable to human-written documents, extending the practical potential of automating software quality assurance (QA) documentation.

4.1. Data Preprocessing

This study constructed a bug report dataset suitable for training large language models, based on actual defect reports from open-source software projects. The source data were collected from the public bug-tracking system Bugzilla, yielding approximately 15,000 reports. During data collection, only entries marked as "fixed" or "closed," rather than "open" or "in progress", were selected to ensure reliability and completeness. This criterion was established to include only cases with clearly documented reproduction procedures and resolution outcomes.

The collected data exhibited substantial variation in quality and were unsuitable for direct use. To address this issue, two sequential stages were performed: rule-based structural verification and quality score–based refinement. In the first stage, regular expressions were used to retain only reports containing all essential components, including Summary, Steps to Reproduce, Expected Behavior, Actual Behavior, and Additional Information. After structural filtering, the CTQRS automatic evaluation tool [8] was applied, and reports scoring 14 or higher out of 17 points were classified as high quality. This threshold followed the criterion defined in prior research for identifying "sufficiently complete reports" and was adopted to minimize learning bias caused by incomplete entries.

After refinement, approximately 3966 reports were finalized as the dataset. A portion of the data was manually reviewed to confirm quality suitability, and the dataset maintained the same structure as the version registered in a public repository [15], ensuring independent verification. The data were divided into training, validation, and test sets in an 8:1:1 ratio to support generalization performance evaluation. Random seed fixation was applied to preserve consistency across experiments.

All finalized reports were normalized into the model input format, consisting of an Input Summary and a Reference Report. Through this preprocessing procedure, the dataset achieved both structural consistency and quality compliance across documents. It provided a reliable foundation for reinforcement learning, ensuring stable policy optimization and reward function design based on the CTQRS framework.

4.2. CTQRS-Guided Reward Design

In this study, the reward function used during the reinforcement learning phase was designed to quantitatively evaluate both the structural completeness and semantic consistency of bug reports. The key idea of the reward design is to integrate multiple dimensions of report quality centered on the CTQRS metric. This allows the model to learn to generate reports that satisfy both structural format and semantic coherence, rather than producing simple sentences.

Each of the five CTQRS dimensions was scored on a normalized scale from 0 to 1 using weighted sub-formulas derived from the presence and completeness of specific report sections. The scoring functions were defined as follows:

Each sub-component of the CTQRS metric is mathematically defined in Equations (1)–(5).

$$\text{Completeness } (C_1) = (N_{\text{present}}/N_{\text{required}}) \tag{1}$$

$$\text{Traceability } (C_2) = f(\text{version, environment, configuration}) \tag{2}$$

$$\text{Quality } (C_3) = 1 - (\text{redundant}_{\text{phrases}}/\text{total}_{\text{sentences}}) \tag{3}$$

$$\text{Reproducibility } (C_4) = \exp(-\Delta\text{steps}/\sigma_{\text{steps}}) \tag{4}$$

$$\text{Specificity } (C_5) = (N_{\text{concrete\_evidence}}/N_{\text{total\_evidence}}) \tag{5}$$

Here, N_{present} denotes the number of required sections present (out of five: Summary, Steps, Expected, Actual, Environment); Δsteps is the deviation from the optimal number of reproduction steps, and σ_{steps} controls the penalty smoothness. Each score was normalized by min–max scaling before aggregation, and the final CTQRS score was computed as

$$\text{CTQRS} = (C_1 + C_2 + C_3 + C_4 + C_5)/5 \tag{6}$$

The overall CTQRS score is then computed by averaging the five sub-scores as shown in Equation (6).

The reward function consists of three major components.

First, the fundamental quality evaluation metrics include the CTQRS score [8], ROUGE-1 Recall and F1 scores [9], and SBERT similarity [12]. CTQRS measures the overall structural fidelity and completeness of a report. The ROUGE metrics assess lexical accuracy, and SBERT evaluates semantic consistency between sentences. All metrics are normalized within the range of 0 to 1 and then integrated using a weighted sum. The default weights are set as $w_{\text{ctqrs}} = 0.55$, $w_{\text{rouge1\_r}} = 0.25$, $w_{\text{sbert}} = 0.20$. These values place greater emphasis on structural completeness while maintaining a balance that reflects semantic naturalness and lexical reproducibility.

Each reward component was computed through a standardized normalization process before integration. The CTQRS, ROUGE, and SBERT scores were scaled to the [0, 1] range using min–max normalization to ensure equal contribution of each metric. The length reward was calculated using a Gaussian function defined as:

$$L_{\text{bonus}} = \exp(-((L - L_0)^2/(2\sigma^2))) \tag{7}$$

The Gaussian-based length reward is formulated in Equation (7) to maintain appropriate report length.

where L denotes the token length of the generated report, $L_0 = 280$ is the target mean length, and $\sigma = 140$ controls the acceptable range of deviation. This function yields its maximum

value of 1 when $L \approx L_0$ and decreases smoothly as the length diverges. The repetition penalty was computed from the 4-g duplication rate:

$$P_{repeat} = n_{dup}/n_{total} \quad (8)$$

where n_{dup} is the count of duplicated 4-g and n_{total} the total number of 4-g. A negative reward of $-0.05 \times P_{repeat}$ was applied proportionally to discourage redundancy. These intermediate calculations produced consistent scaling across reward terms before the final weighted-sum integration.

The weight configuration of (0.55, 0.25, 0.20) was determined through preliminary sensitivity experiments that assessed the relative influence of each metric on convergence stability and output diversity. CTQRS received the highest weight because structural completeness and reproducibility represent the primary objectives of bug report generation, while ROUGE-1 and SBERT were assigned lower but complementary weights to preserve lexical fidelity and semantic coherence. When the CTQRS weight exceeded 0.6, the model produced structurally rigid and less diverse reports, whereas increasing the SBERT weight above 0.3 resulted in longer and more redundant sentences. Conversely, reducing any component below 0.15 degraded convergence stability and quality balance. The selected ratio therefore provided the most stable trade-off between structure-oriented and meaning-oriented optimization. To mitigate potential overfitting to specific reward components, the reward signals were smoothed through an Exponential Moving Average (EMA) mechanism during PPO updates, which prevented the policy from reacting excessively to transient metric fluctuations. Small variations (± 0.05 per component) produced similar qualitative outcomes, confirming that the model's behavior is robust to minor weighting changes.

Second, the length-based reward component ensures that the generated report remains within an appropriate range of descriptive length. Reports that are too short tend to omit information, while excessively long ones often contain redundant or inefficient expressions. To prevent this, the target average length was set to $L_0 = 280$ with a standard deviation $\sigma = 140$. The length reward was defined in a Gaussian form, where the reward approaches its maximum value (1) when the report length falls within the target range and decreases as it diverges from it. This component contributes to the total reward with a weight factor of 0.1.

Third, the repetition penalty component was introduced to suppress excessive repetition of identical phrases or sentence patterns. The 4-g frequency of the generated document is calculated, and a negative reward of -0.05 is applied whenever identical *n*-grams are repeated. The repetition penalty is defined in Equation (8) to suppress redundant *n*-gram patterns. This mechanism helps ensure readability and diversity in expression.

The final reward function is defined as follows:

$$R = w_{ctqrs}\, C + w_{rouge1\_f1}\, F1 + w_{rouge1\_r}\, R1 + w_{sbert}\, S + \alpha L_{bonus} + \beta P_{repeat} \quad (9)$$

- Here, C, F1, R1, and S denote CTQRS, ROUGE-1 F1, ROUGE-1 Recall, and SBERT similarity, respectively. L_{bonus} represents the length reward, and P_{repeat} represents the repetition penalty. The constants $\alpha = 0.10$ and $\beta < 0$ control the relative influence of the length and repetition components.

The complete reward function integrating CTQRS, ROUGE, SBERT, length reward, and repetition penalty is summarized in Equation (9).

The RewardAgent computes the reward values by comparing the generated reports with their reference counterparts based on this reward function and passes them to the policy optimization algorithm (PPO) [10,11]. Each metric is computed through the Evalu-

ationAgent, which also provides fallback mechanisms to approximate the reward using ROUGE and SBERT when CTQRS evaluation is unavailable.

This reward design enables the reinforcement learning model to go beyond simple imitation of textual similarity. It allows comprehensive learning of report structure, content reproducibility, and linguistic naturalness. The CTQRS metric captures structural completeness, while ROUGE and SBERT ensure lexical and semantic consistency. The length reward and repetition penalty further stabilize overall document quality. As a result, this reward framework serves as a central component of reinforcement learning that enhances bug report quality across multiple dimensions.

4.3. Reinforcement Learning Framework

The reinforcement learning procedure in this study was designed as a policy optimization framework in which the bug report generation model learns to maximize a CTQRS-based reward function. The overall process consists of four components: the Policy Model, the Reference Model, the Value Model, and the Reward Model. Training is conducted using the Proximal Policy Optimization (PPO) algorithm [11]. This structure ensures both stability and efficiency in reward-driven learning while preventing the text generation model from overfitting to unstable reward signals.

The Policy Model is initialized from a supervised fine-tuned large language model, Qwen2.5-7B [5], and is trained to generate reports that reflect CTQRS metrics. The Reference Model shares the same architecture but remains fixed during training to serve as a stable baseline for measuring policy updates.

The Reward Model is implemented through the RewardAgent. It compares the generated reports with reference reports and computes evaluation metrics such as CTQRS, ROUGE-1, and SBERT. These metrics are combined into a single scalar reward according to the reward function. The computed reward provides a quantitative signal of the generated report's quality and guides policy updates during the PPO advantage estimation process.

The Value Model is a neural network designed to approximate the expected reward of the policy. It reduces the variance of the advantage function and improves training stability. In this study, the ValueModelWrapper receives the hidden states from the base language model and outputs a scalar value. This allows the model to estimate the relative value of a generated report for a given input and mitigates excessive reward fluctuations during policy updates.

Training applies PPO's clipping mechanism to prevent the policy from changing excessively within a single update. In each episode, the model receives a bug report summary as input, generates a structured report, obtains a reward signal from the RewardAgent, and updates the policy network by minimizing the loss function. The loss consists of policy loss, value loss, and an entropy term, expressed as:

$$L = E_t\,[\min(r_t(\theta)A_t,\ \mathrm{clip}(r_t(\theta),1-\varepsilon,1+\varepsilon)A_t)] - c_1 L_{value} + c_2 S \tag{10}$$

The overall PPO objective function is defined in Equation (10), balancing policy improvement and reward stability.

- Where $r_t(\theta)$ is the probability ratio between the new and reference policies, A_t is the advantage function, L_{value} denotes the value loss, S represents policy entropy, and c_1, c_2 are weighting coefficients. This formulation maintains stable convergence and balances exploration and exploitation during training.

The total loss was further decomposed into three sub-components:

$$\text{Policy Loss: } L_{policy} = -E_t\,[\min\,(r_t(\theta)\,A_t,\ \mathrm{clip}\,(r_t(\theta),\,1-\varepsilon,\,1+\varepsilon)\,A_t)] \tag{11}$$

$$\text{Value Loss: } L_{value} = E_t\,[(V\_\theta(s_t) - R_t)^2] \tag{12}$$

$$\text{Entropy Loss: } L_{entropy} = -E_t\,[\,\Sigma\,\pi\_\theta(a \mid s_t)\,\log\,\pi\_\theta(a \mid s_t)] \tag{13}$$

Here, R_t denotes the discounted reward-to-go, and $V\_\theta(s_t)$ is the predicted state value. The policy loss encourages reward-aligned updates, the value loss reduces variance in advantage estimation, and the entropy term maintains exploration diversity. The combined objective balances exploitation and exploration for stable convergence.

The three sub-loss components of PPO (policy, value, and entropy) are expressed in Equations (11)–(13). These sub-loss functions were derived directly from the standard PPO objective [11], ensuring theoretical consistency with the baseline formulation while allowing explicit control over exploration and variance reduction.

All training was conducted using the PyTorch-based PPOv2Trainer module. The mini-batch size was set to 1, and the learning rate was 2×10^{-6}. Rewards computed by the RewardAgent were used with the Value Model for advantage estimation, and exponential moving average (EMA) validation was applied to confirm convergence stability. After training, a critic-based refinement process was applied to enhance the logical consistency of the generated reports [13].

As a result, the proposed reinforcement learning framework directly regulates policy quality using structured reward signals derived from CTQRS, ROUGE, and SBERT metrics. Through PPO stabilization mechanisms, it ensures both the consistency and convergence of report quality. This structured training procedure enables the model to move beyond simple text imitation and acquire an autonomous capability to generate reports that inherently meet quality standards.

In this study, the PPO algorithm followed the standard implementation of Schulman et al. [11]. The clipping range for policy updates was set to [0.8, 1.2], constraining the probability ratio between new and reference policies to $\pm 20\%$ per update step. This range was selected empirically to balance reward stability and policy exploration. Discounted rewards were used in advantage computation with a discount factor of $\gamma = 0.99$, ensuring that long-term quality signals were preserved without propagating excessive variance. The entropy weight coefficient was set to $c_2 = 0.01$ to encourage diversity and prevent premature convergence toward high-frequency patterns. All hyperparameters were validated through preliminary stability experiments that confirmed consistent reward convergence under EMA-based tracking.

4.4. Stabilization and Refinement Mechanism

In reinforcement learning–based bug report generation, instability in reward signals can slow convergence and reduce the consistency of training outcomes. To address this issue, this study designed a learning stabilization mechanism that combines Reward Stabilization and Output Refinement.

During the reward stabilization phase, an exponential moving average (EMA)–based reward estimation was applied to mitigate reward fluctuations observed during training. The reward calculated at each step was tracked cumulatively through EMA, and when the deviation from the previous mean reward exceeded a threshold, the learning rate was automatically adjusted. This prevented the policy from reacting excessively to short-term reward increases or overfitting to specific episodes. Length rewards and repetition penalties within the reward function were also scaled by EMA, allowing stable reward signals as training progressed. These procedures suppressed abrupt distortions in the reward distribution during the convergence of the PPO-based policy and ensured stable learning [11].

The exponential moving average (EMA) reward smoothing was computed as:

$$EMA_t = \alpha \times Reward_t + (1 - \alpha) \times EMA\_\{t - 1\} \quad (14)$$

The EMA-based reward smoothing process is formulated in Equation (14) to stabilize short-term reward fluctuations during training.

Where α is the smoothing coefficient that controls the balance between responsiveness and stability. In this study, $\alpha = 0.1$ was empirically selected to provide stable yet sufficiently responsive tracking of reward fluctuations. This mechanism suppressed short-term reward oscillation while retaining gradual improvement signals.

This formulation represents a recursive smoothing process where recent rewards contribute more strongly to the moving average, effectively filtering noise from transient spikes and providing a stable feedback signal for PPO updates.

The output refinement phase followed the completion of reinforcement learning and aimed to review and improve the quality of the generated bug reports. This study adopted a self-critic refinement approach [13]. Instead of training a separate critic network, the same language model was used in a critic role to perform self-evaluation and self-revision. Specifically, a draft produced during reinforcement learning was evaluated by the EvaluationAgent using CTQRS, ROUGE-1 (F1 and Recall), and SBERT metrics. The results were converted into concise feedback sentences such as CTQRS = 0.73, rouge1_r = 0.61, and sbert = 0.83. This feedback was inserted into the [Feedback] section of the refinement prompt, guiding the model to regenerate the report with the same input and context but in a direction that improved quality metrics.

The refinement prompt was structured into three parts: [Input Summary], [Draft Report], and [Feedback]. The [Feedback] section contained metric-based guidance generated by the EvaluationAgent in the format "CTQRS = 0.73, ROUGE1_R = 0.61, SBERT = 0.83." This prompt explicitly instructed the model to improve completeness, clarity, and semantic alignment without modifying factual content from the draft. Refinement iterations were controlled by the hyperparameter critic_iters (set to 1 in this study), and automatic termination occurred when metric gains fell below ΔCTQRS < 0.01 and ΔSBERT < 0.005. This procedure ensured stable convergence and prevented over-editing that could distort the original context.

Decoding during refinement used deterministic settings with do_sample = False and max_new_tokens = 512 to ensure reproducibility across runs. Each refinement call consisted of a single self-critic regeneration, and the number of iterations could be adjusted through the critic_iters hyperparameter. The process terminated when quality improvement fell below a threshold (ΔCTQRS < 0.01, ΔSBERT < 0.005) or reached the maximum iteration count. These conditions prevented excessive modifications that might compromise document consistency.

The core of this refinement stage lies in an autonomous self-improvement process, where the model interprets quantitative feedback on its own output and integrates it into subsequent revisions. Feedback centered on CTQRS enhances logical coherence and structural balance across report sections, while ROUGE and SBERT improve linguistic fluency and semantic alignment. This self-critic refinement structure minimizes reward overfitting and contextual distortion that can arise during reinforcement learning, ultimately improving both readability and consistency of the final report.

The proposed stabilization and refinement mechanism ensures consistent reward signals and convergence stability throughout reinforcement learning, followed by fine-grained quality improvement through self-critic refinement. Reward stabilization reinforces stability during training, and self-critic refinement corrects the completeness of the final output. As a result, the framework goes beyond simple reward optimization, establishing

an autonomous quality control system capable of maintaining and enhancing the structural completeness and semantic coherence of generated reports.

4.5. Bug Report Generation Process

After reinforcement learning is completed, the model generates actual bug reports based on the policy it has internalized through training. The purpose of this process is to automatically produce structured and reproducible bug reports from the given summaries. The overall procedure naturally proceeds through three stages: prompt construction, report generation, and result evaluation.

Before generation, the input data are preprocessed according to a predefined CTQRS-based structured template. The input summary includes the problem situation or key symptoms reported by users, which the model converts into a format suitable for the prompt. The prompt consists of seven sections: summary, reproduction steps, expected result, actual result, environment information, evidence, and additional information. Constraints are applied to ensure that each section is included. This structured template guides the model to produce systematic and consistent reports rather than free-form text.

In the report generation stage, the trained policy model creates documents aligned with the CTQRS standard. A deterministic decoding strategy is used to ensure that identical inputs always produce identical outputs. Probabilistic sampling is excluded, and the temperature parameter is fixed at zero to eliminate randomness. These settings minimize stochastic variation, securing reproducibility and comparability of results. The model is trained to maintain logical coherence across sections, allowing the symptoms described in the summary to connect naturally with the actual results and environmental context.

The generated reports are automatically validated using CTQRS, ROUGE, and SBERT metrics to evaluate both structural completeness and semantic consistency. CTQRS measures the presence of required elements and reproducibility, ROUGE assesses lexical similarity with reference reports, and SBERT evaluates semantic alignment. These evaluation results serve as quantitative evidence of output quality and can be used as feedback for further refinement if necessary.

This generation process goes beyond simple text creation. It implements an automated documentation procedure capable of self-assessment according to predefined quality standards. The model transforms the input summary into a structured report based on the internalized policy and verifies whether the generated content meets the quality metrics, achieving autonomous document generation. Consequently, the proposed bug report generation process functions as a systematic procedure that consistently produces reports with reproducibility, structural coherence, and semantic reliability.

5. Experiments

5.1. Experimental Setup

The experiments were conducted to verify the effectiveness and stability of the proposed CTQRS-based reinforcement learning framework. All experiments were performed using the Qwen2.5-7B-Instruct model [5]. Training and inference were carried out under identical environments. The model performance was evaluated in terms of structural completeness, lexical fidelity, and semantic consistency. Three representative quality metrics were used: CTQRS, ROUGE-1, and SBERT.

The training procedure consisted of two stages: supervised fine-tuning (SFT) and reinforcement learning (RL). In the SFT stage, the model learned the fundamental structure and expression rules of bug reports [14]. In the RL stage, the PPO algorithm [11] was applied to optimize a reward function that integrated CTQRS, ROUGE, and SBERT. During this phase, the model generated reports based on input summaries and updated its policy

iteratively by receiving the evaluation scores as reward signals, leading to progressive quality improvement.

After the reinforcement learning phase, an additional self-critic refinement stage was introduced [13]. In this stage, the model re-input its own generated reports and revised them by incorporating feedback from the EvaluationAgent, which computed the CTQRS, ROUGE, and SBERT metrics. The number of refinement iterations was controlled by the hyperparameter critic_iters, which was set to 1 in this study. Additional refinement automatically stopped when the improvement in quality metrics became negligible (ΔCTQRS < 0.01, ΔSBERT < 0.005). All refinement processes used deterministic decoding with do_sample = False and max_new_tokens = 512 to ensure reproducible outputs across runs.

The training environment was based on Ubuntu 24.04 (WSL2). The hardware configuration included an Intel Xeon Silver 4310 CPU, 128 GB of RAM, and dual NVIDIA RTX 4090 GPUs (24 GB each). CUDA 12.6, PyTorch 2.5.1, Transformers 4.55.3 [37], and Unsloth 2025.9.1 [38] were used. The learning rate was set to 2×10^{-6}, the batch size to 1, the gradient accumulation steps to 8, and the number of epochs to 3. All experiments used a fixed random seed of 42 to ensure consistent and reproducible results.

The dataset consisted of 3966 high-quality reports collected from Bugzilla, selected from a total of 15,000 entries that achieved CTQRS scores of at least 14 [15]. The data were divided into training, validation, and test sets at an 8:1:1 ratio. Each report contained an input summary and a reference report. The dataset was normalized to reflect the CTQRS-based template structure. Incomplete or duplicate reports were excluded.

The report generation process used a deterministic decoding strategy that excluded probabilistic sampling. The temperature parameter was fixed at 0 so that identical inputs always produced identical outputs. This configuration eliminated randomness and allowed clear analysis of the effects of reinforcement learning and refinement.

The entire experiment consisted of four modular stages: model training, report generation, automatic evaluation, and result analysis. Each stage was implemented independently to allow separate verification. This modular design ensured both training stability and experimental reproducibility. It also enabled a systematic evaluation of the proposed reinforcement learning framework's ability to enhance the structural completeness and semantic consistency of bug reports.

5.2. Dataset

This study constructed a training dataset based on real bug reports to perform reinforcement learning experiments for automatic bug report generation in open-source software environments. The data were collected from the open-source bug tracking system Bugzilla, where approximately 15,000 raw reports were gathered. After a multi-stage refinement process, a total of 3966 high-quality reports were retained [15].

During data collection, only reports marked as "fixed" or "closed" were selected, excluding those labeled "open" or "in progress." Resolved reports typically include clear descriptions of the issue's cause, execution environment, and reproduction steps. Such characteristics make them suitable for training models to learn the structural components of bug reports. The collected data were then filtered sequentially using regular expression–based screening followed by CTQRS automatic evaluation to reduce quality variation.

In the first stage, only reports containing all five essential sections—Summary, Steps to Reproduce, Expected Behavior, Actual Behavior, and Additional Information—were retained. Reports consisting solely of log outputs or code fragments were removed, along with incomplete or duplicate entries. In the second stage, the CTQRS automatic evaluation tool was applied, and only reports that achieved a score of 14 points or higher were selected. This threshold was established based on previous research defining the characteristics of a

"good-quality bug report" and serves as a quantitative criterion to ensure both structural completeness and reproducibility.

To confirm data quality, 200 samples were manually reviewed by researchers. The final dataset consisted of 3966 reports, divided into training, validation, and test sets in an 8:1:1 ratio. A fixed random seed was applied to maintain identical splits across repetitions, ensuring reproducibility and comparability of experimental results.

The decision to exclude low-quality reports was made to ensure stable and consistent training signals during reinforcement learning. Because the CTQRS-based reward function quantitatively evaluates report completeness and reproducibility, including extremely incomplete or noisy reports would lead to unstable or undefined reward values, hindering policy convergence. High-quality reports were therefore used to establish reliable structural and semantic standards that the model could internalize as quality priors. Future work will extend this framework to include low-quality or unstructured reports as inputs, enabling evaluation of the model's ability to autonomously refine them into high-quality outputs under CTQRS-guided learning.

To clarify, this study intentionally focused on training and validating the model using high-quality reports (CTQRS $\geq$ 14) to ensure stable reward computation and convergence during reinforcement learning. Consequently, the model's direct capability to enhance low-quality or incomplete reports was not quantitatively evaluated in this version. Nevertheless, the proposed CTQRS-guided reward and self-critic refinement mechanisms are inherently designed to generalize to such cases, and future work will explicitly test this capability using low-quality inputs to validate the model's refinement performance.

Each data instance was formatted for model training in a standardized structure. The input consisted of user-provided unstructured content (from the NEW_llama_output column) reformatted as an Input Summary, and the output represented the fully structured Reference Report (from the text column). All entries were preprocessed to conform to the CTQRS template, enabling the model to learn consistent section structures within prompts.

This dataset design allows the model to learn from diverse real-world bug report patterns and acquire the ability to transform unstructured inputs into well-organized reports. Consequently, the dataset functions as a quantitative foundation for evaluating not only sentence generation performance but also quality-oriented report generation capability.

5.3. Evaluation Metrics

This study applied a multidimensional evaluation framework to objectively verify the quality of generated bug reports. The framework encompasses structural completeness, lexical fidelity, and semantic consistency. Four metrics were used: CTQRS, ROUGE-1 Recall, ROUGE-1 F1, and SBERT similarity. Each metric quantitatively measures a distinct aspect of report quality.

CTQRS [8] served as the primary indicator for assessing the structural integrity and reproducibility of the reports. It assigns a score based on how thoroughly the essential components are included. Seven elements are evaluated: Summary, Steps to Reproduce, Expected Result, Actual Result, Environment Information, Evidence, and Additional Information. The maximum score of 17 points was normalized to a range between 0 and 1. A higher value indicates that the report is more systematically organized and contains sufficient information for defect reproduction.

ROUGE-1 Recall [9] measures how well the generated report includes the key vocabulary found in the reference report. This metric evaluates lexical coverage, and higher values indicate a stronger reflection of the main content in the reference report. However, since Recall considers only the proportion of overlapping words, it can yield high scores

even when unnecessary words are included. To address this limitation, ROUGE-1 F1 was also measured.

ROUGE-1 F1 [9] represents the harmonic mean of Recall and Precision, reflecting both lexical coverage and expression accuracy. It evaluates how precisely the generated text includes the essential words while minimizing redundant or irrelevant content. A higher F1 score indicates well-balanced expression and minimal verbosity or repetition.

SBERT similarity [12] was used to assess semantic consistency between reports. Both the generated and reference reports were converted into sentence embeddings, and cosine similarity was computed to determine whether they convey equivalent meanings despite differences in wording. A higher score indicates that the generated report maintains the semantic context of the reference report consistently.

These four metrics complement one another and collectively capture multiple dimensions of report quality that a single measure cannot represent. CTQRS assesses structural completeness, ROUGE metrics evaluate lexical fidelity and precision, and SBERT measures semantic consistency. By combining these metrics, this study comprehensively verified whether the generated reports exhibit structural completeness and semantic reliability beyond simple text generation.

5.4. Baseline and Research Questions

The experimental design in this study compared three training configurations to verify the progressive improvement of bug report generation models. The first configuration was supervised fine-tuning (SFT), where the model learned the structure and sentence composition of bug reports using structured training data. The second configuration was reinforcement learning (RL), which used the SFT-trained model as the initial policy and applied a CTQRS-based reward function to directly learn structural completeness and semantic consistency. The third configuration, combined training (SFT + RL), integrated both approaches to reflect linguistic stability from supervised learning and quality awareness acquired through reinforcement learning. These three configurations formed the foundation of the experimental design for systematically comparing the influence of each learning method on bug report quality.

The baseline model was a LoRA-based SFT model trained to imitate the format of bug reports without incorporating quality metrics such as CTQRS or SBERT during training. This approach effectively reproduced document structure but did not quantitatively improve the completeness or semantic coherence of reports. In contrast, the proposed RL model optimized a multi-reward function that included CTQRS, ROUGE, and SBERT metrics using the PPO algorithm. This allowed the model to directly target report quality as its learning objective. The integrated model (SFT + RL) combined the quality optimization capability of reinforcement learning with the stability of supervised training, ensuring both structural consistency and quality-oriented learning effects.

It should be noted that the RL configuration was trained directly from the base LLM without prior supervised fine-tuning. This setting was intentionally designed to evaluate whether reinforcement learning alone can achieve structural and semantic improvements without any SFT initialization. In contrast, the integrated SFT + RL configuration represents the sequential pipeline combining both stages.

To compare these three learning configurations, the following research questions were defined:

- RQ1. Can a reinforcement learning–based model improve structural completeness and semantic consistency compared with a model trained only with supervised fine-tuning? This question aims to verify whether reinforcement learning can achieve absolute improvements in quality metrics such as CTQRS, ROUGE, and SBERT. It evaluates

whether reward-driven learning contributes more effectively to bug report quality enhancement than simple supervised learning.

- RQ2. Can the integrated model (SFT + RL) achieve a higher quality level than each single training method?
  This question examines whether the combined approach can utilize the advantages of both individual methods, achieving both structural stability and quality consistency.

This comparative experiment provided a systematic analysis of the relationships among the three training configurations and established empirical evidence that reinforcement learning serves as a key technique for improving the quality of automated bug report generation.

5.5. Experimental Results

The experimental results of this study are presented in Table 1. The proposed reinforcement learning–based bug report generation model effectively improves structural completeness and semantic consistency compared to the supervised learning baseline. The entire training process proceeded sequentially, beginning with the supervised fine-tuning (SFT) stage, followed by the reinforcement learning (RL) stage, and concluding with the integrated refinement stage that combined both learning methods.

Table 1. Experimental Results of Each Training Phase.

Training Phase	Loss	Reward	CTQRS Score	SBERT Similarity	ROUGE-1 Recall	Description
SFT (Supervised Fine-Tuning)	1.9 → 1.3	-	0.46	0.68	0.52	The model learned the formal structure of bug reports and converged stably, although some entries remained missing and minor semantic inconsistencies were observed.
RL (Reinforcement Learning)	-	0.42 → 0.63	0.72	0.84	0.61	Applying the CTQRS-based reward function improved both structural completeness and semantic consistency, and the PPO algorithm showed stable convergence.
Refine (Integrated SFT + RL)	-	~0.65 (Stable)	0.76	0.85	0.63	Integrating SFT and RL enhanced structural integrity and expressive precision simultaneously, achieving the highest performance in reward and quality metrics.

In this context, "RL (Reinforcement Learning)" in Table 1 refers to direct RL training starting from the base LLM without any prior SFT. This configuration was included to examine whether quality-driven optimization alone could replace structural learning through supervised fine-tuning.

In the initial SFT stage, the model learned the formal structure of bug reports and achieved stable convergence. The training loss decreased from 1.9 to 1.3, and the model successfully internalized the core section composition and sentence patterns of reports. The CTQRS score averaged 0.46, and the SBERT similarity reached 0.68, indicating that the basic structural format and contextual coherence were established. However, some reports exhibited missing sections or incomplete semantic connections, showing that supervised learning alone was insufficient to meet quality standards.

During the RL stage, the CTQRS-based reward function guided the model to directly optimize structural completeness and semantic consistency. The reward value gradually increased from 0.42 to 0.63, confirming that the PPO algorithm operated stably with exponential moving average (EMA) convergence. In this stage, the CTQRS score rose to 0.72, the SBERT similarity reached 0.84, and the ROUGE-1 recall also improved overall.

Through reinforcement learning, the model advanced beyond reproducing report formats and learned policies that improved quality based on evaluation metrics.

In the final integrated refinement stage, the outcomes of SFT and RL were combined to enhance both stability and fine-grained expression. The reward value remained stable at approximately 0.65. The CTQRS and SBERT scores achieved peak performances of 0.76 and 0.85, respectively. Logical connections between sections and conciseness of expression improved compared to earlier stages, resulting in a balanced and coherent report structure. The length reward and repetition penalty functioned properly, preventing excessive verbosity or redundancy in generated texts.

These results confirm the first research question (RQ1): reinforcement learning substantially improves the structural quality and semantic coherence of reports compared to pure supervised learning. Through reinforcement learning, the model internalized quality metrics and autonomously adjusted report completeness and consistency using reward signals. The findings also address the second research question (RQ2): the integrated model that combines supervised and reinforcement learning achieves higher quality than either single learning method. The linguistic stability of SFT and the quality-oriented optimization of RL complemented each other, enhancing both structural completeness and semantic consistency.

Across all training stages, the reward gradients converged stably, and no model collapse or reward explosion occurred. The CTQRS score increased from 0.46 to 0.76, and the SBERT similarity improved from 0.68 to 0.85, demonstrating that the model enhanced both structural quality and semantic reliability. These outcomes empirically verify that the proposed CTQRS-based reinforcement learning framework effectively improves the quality of automated bug report generation while ensuring structural stability and semantic coherence. As summarized visually in Figure 1, the three-stage training (SFT, RL, and Refinement) demonstrates progressive improvements in both structural and semantic metrics. The CTQRS score rises from 0.46 to 0.72 and finally 0.76, while SBERT similarity increases from 0.68 to 0.85, confirming that each training phase contributes cumulatively to report quality.

6. Discussion

6.1. Analysis of Experimental Results

The analysis of experimental results focused on the performance differences across model training structures and the impact of each stage on report quality. The core objective was to compare how three models, namely Supervised Fine-Tuning (SFT), Reinforcement Learning (RL), and Integrated Training (SFT + RL), affected the quality of bug report generation.

The SFT stage was effective in learning the basic structure of reports. The model reproduced the structural format of the CTQRS-based template, and the loss value decreased steadily from 1.9 to 1.3. This indicates that the model successfully learned the overall framework of the document and the inclusion of essential elements. However, while structural consistency was achieved, the logical connection between items and contextual meaning remained limited. In particular, comparisons between the "Expected Behavior" and "Actual Behavior" sections were often unclear, and the "Environment" section lacked sufficient specificity. Quantitatively, the CTQRS score was 0.46 and the SBERT similarity was 0.68, confirming that the model learned the basic structure but had limitations in semantic coherence.

The RL stage addressed these limitations more effectively. By applying a reward function that combined CTQRS, ROUGE, and SBERT metrics under the PPO algorithm, the model advanced beyond simple format reproduction and began quality-oriented learning aligned with evaluation criteria. The reward value increased steadily from 0.42 to 0.63,

showing stable convergence through exponential moving average (EMA) smoothing. The CTQRS score rose to 0.72, and the SBERT similarity reached 0.84. These results show that RL improved both structural completeness and semantic consistency. Reports generated after RL displayed clearer procedural descriptions in the "Steps to Reproduce" section and more direct, explicit comparisons in "Expected–Actual" expressions. The balance between length reward and repetition penalty reduced verbosity, resulting in concise and information-dense outputs.

The integrated SFT + RL model combined the stability of supervised learning with the quality improvements of reinforcement learning. It achieved the highest performance among the three approaches, with CTQRS of 0.76 and SBERT of 0.85. The grammatical stability and structural reproduction acquired during SFT synergized with the reward-driven optimization of RL, strengthening both the precision of expressions and the semantic connections between sections. The "Environment" section contained more specific details such as operating system, network, and version information, while the "Evidence" section more naturally included examples like logs, file paths, and screenshots. This indicates that the model internalized contextual patterns of real-world defect reports beyond mere structural learning.

Quantitative results further supported this trend. The CTQRS score improved by approximately 65% compared with SFT, and SBERT increased by about 25%. ROUGE-1 recall also rose, indicating broader lexical coverage, and the stable F1 score of around 0.61 showed a balanced relationship between precision and recall. These findings suggest that the model did not simply generate longer outputs but learned to retain essential information efficiently while minimizing redundant expressions.

Throughout training, the reward gradients converged stably, and neither reward explosion nor policy collapse was observed. PPO's policy clipping prevented excessive updates, and EMA-based reward tracking reduced reward amplitude fluctuations. As a result, the reward distribution remained stable in later training stages, indicating convergence toward consistent improvement rather than unstable exploration.

Reinforcement learning internalized quality awareness within the model, which supervised learning alone could not provide. The integrated SFT + RL framework combined this quality awareness with structural stability, resulting in the generation of the highest-quality bug reports. The findings indicate an evolution toward an autonomous generation model capable of self-assessing and refining report structure and semantics based on quantitative quality metrics rather than simple text generation.

The proposed CTQRS-based reinforcement learning framework enhanced both structural completeness and semantic coherence by directly learning and internalizing quality evaluation criteria. The results empirically show that reinforcement learning can serve as a key method for quality-driven software documentation, extending beyond conventional language model improvements.

The RL-only configuration improved report quality compared with the SFT-only baseline but exhibited instability in structural alignment. This confirms that reinforcement learning can enhance quality metrics but still requires structural priors from supervised training to achieve fully coherent outputs.

In addition, a preliminary configuration combining only SFT and refinement without RL produced results nearly identical to the SFT baseline, indicating that reward-driven optimization is indispensable for further quality improvement.

6.2. Threats to Validity

This study showed that the CTQRS-based reinforcement learning framework can enhance the quality of bug report generation. However, several potential threats to the

reliability and generalizability of the results may exist in the design and interpretation of the experiments. These threats relate to dataset composition, reward function design, evaluation metrics, training stability, and applicability scope. A multifaceted examination of these factors is essential for assessing the validity of the findings.

From the perspective of dataset composition, this study used 3966 reports that achieved a CTQRS score of 14 or higher out of approximately 15,000 reports collected from Bugzilla. This filtering step was intended to ensure high-quality training data, but it inevitably reduced dataset diversity. Bug report quality varies widely depending on the author's expertise, project domain, and linguistic expression. The dataset used in this study contained reports that were relatively complete and well-structured. As a result, the model's ability to generalize to incomplete or unstructured reports may not have been sufficiently verified. The train, validation, and test sets were divided with a fixed random seed to ensure reproducibility, yet a single-seed configuration cannot fully eliminate accidental bias in data distribution.

The design of the reward function may also influence the construct validity of the results. The reward signal was calculated as a weighted sum of CTQRS, ROUGE, and SBERT scores, and the weights were determined empirically. Such settings directly affect the direction of policy learning. If one metric receives excessive weight, the model may overfit to that aspect at the expense of others. For instance, a high CTQRS weight can improve structural completeness but may reduce semantic diversity and flexibility of expression. Conversely, emphasizing SBERT may enhance semantic similarity while weakening structural consistency. Because reward sensitivity critically affects reinforcement learning outcomes, future work should systematically explore different weight configurations to identify a balanced reward structure.

Nevertheless, the chosen configuration (0.55, 0.25, 0.20) was empirically validated through sensitivity analysis, which showed that moderate variations in each weight (± 0.05) did not significantly affect convergence trends or qualitative performance. This indicates that the reward function is not overfitted to a specific parameter setting and remains robust under small perturbations.

The limitations of the evaluation metrics further affect the reliability of interpretation. The CTQRS, ROUGE, and SBERT metrics used in this study offer automated assessments of structural completeness, lexical fidelity, and semantic coherence. Although these measures are useful for quantifying report quality, they do not fully capture practical usefulness. CTQRS evaluates the presence of required elements but does not assess logical coherence or technical correctness among them. ROUGE relies on token-level overlap and does not reflect contextual appropriateness, while SBERT's semantic similarity accuracy depends on the underlying embedding model. To address these shortcomings, human evaluation based on actual developer usage or task-oriented indicators such as defect reproduction time and fixing efficiency should be incorporated in future work.

Training stability in the reinforcement learning process is another potential source of threat. PPO generally provides stable learning, yet when the reward distribution is unbalanced or the learning rate is too low, policy updates may become insufficient. This study adopted an exponential moving average (EMA) to smooth abrupt fluctuations in rewards, but this approach may also dilute subtle quality differences. As a result, the model may fail to capture minor stylistic or semantic refinements even when generating high-quality reports. The balance between training stability and expressive precision remains a key issue for further optimization.

The external validity of the results is also limited. The experiments were conducted primarily on English-language open-source bug reports, so equivalent performance cannot be assumed for other languages or commercial software environments. Enterprise issue

tracking systems such as Jira and GitHub Issues follow distinct reporting structures and conventions, and quality standards differ across projects. These environmental variations can significantly affect model behavior, and the CTQRS-based reward mechanism may not function uniformly across all domains.

This study showed the feasibility of reinforcement learning for generating bug reports. Factors such as dataset diversity, reward weight configuration, reliance on automated evaluation, detailed tuning of reinforcement learning, and domain generalization introduce uncertainty in interpreting the results. These factors represent opportunities for refinement and expansion rather than limitations. Building more diverse datasets, incorporating human-centered evaluation methods, optimizing multi-objective reward structures, and improving algorithms for learning stability would strengthen both the practical applicability and general validity of the proposed framework.

7. Conclusions

This study proposes a reinforcement learning (RL) framework based on CTQRS to automatically generate reliable and structurally complete bug reports using open-source large language models (LLMs). Previous models for bug report generation failed to address both structural incompleteness and semantic inconsistency simultaneously. To overcome this limitation, the proposed framework directly integrates CTQRS metrics into the reward system, enabling the model to learn both structural completeness and semantic coherence of reports. The framework follows a multi-stage learning process combining supervised fine-tuning (SFT), reinforcement learning, and critic-based refinement. In the SFT stage, the model learns the basic format and sentence structure of reports. During the reinforcement learning stage, the PPO algorithm optimizes the CTQRS-centered reward function. Stability is further enhanced through an exponential moving average (EMA) of rewards and a self-critic refinement process, ensuring convergence stability and consistency in outcomes. This integrated approach improves both structural completeness and semantic accuracy of generated reports, achieving significant performance gains in CTQRS and SBERT metrics. Experimental results show that the proposed RL-based model substantially enhances adherence to structural rules and contextual coherence compared with conventional supervised models, improving the reliability and practicality of quality assurance (QA) document automation. The findings show the potential of evolving from a simple natural language generation model to a self-improving generation system that evaluates and adjusts its outputs using quantitative quality metrics. Future work will focus on refining the sensitivity of individual CTQRS components through critic iteration tuning and parameter decomposition, as well as applying the proposed framework to domain-specific datasets such as security vulnerability reports and embedded system log analysis. Moreover, the study plans to extend the reward mechanism by combining automated evaluation with expert quality feedback, advancing toward a human-guided reinforcement learning paradigm that strengthens reliability and interpretability.

Funding: This research received no funding from any source.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data used in this study are publicly available from the Bugzilla open-source repository, as cited in the manuscript [15]. The dataset was utilized solely for research and preprocessing purposes to ensure data quality. No new or proprietary data were created or analyzed in this study.

Conflicts of Interest: The author declares no conflict of interest.

References

1. Bettenburg, N.; Just, S.; Schröter, A.; Weiss, C.; Premraj, R.; Zimmermann, T. What makes a good bug report? In Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE-16), Atlanta, GA, USA, 9–14 November 2008; pp. 308–318.
2. Fan, Y.; Xia, X.; Lo, D.; Hassan, A.E. Chaff from the wheat: Characterizing and determining valid bug reports. *IEEE Trans. Softw. Eng.* **2018**, *46*, 495–525. [CrossRef]
3. OpenAI. GPT-4 technical report. *arXiv* **2023**, arXiv:2303.08774. [CrossRef]
4. Touvron, H.; Lavril, T.; Izacard, G.; Martinet, X.; Lachaux, M.A.; Lacroix, T.; Rozière, B.; Goyal, N.; Hambro, E.; Azhar, F.; et al. LLaMA: Open and Efficient Foundation Language Models. *arXiv* **2023**, arXiv:2302.13971. [CrossRef]
5. Qwen Team. Qwen2.5 technical report. *arXiv* **2024**, arXiv:2412.15115. [CrossRef]
6. Brown, T.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J.D.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; et al. Language models are few-shot learners. *Adv. Neural Inf. Process. Syst.* **2020**, *33*, 1877–1901.
7. Wei, J.; Wang, X.; Schuurmans, D.; Bosma, M.; Ichter, B.; Xia, F.; Chi, E.; Le, Q.V.; Zhou, D. Chain-of-thought prompting elicits reasoning in large language models. *Adv. Neural Inf. Process. Syst.* **2022**, *35*, 24824–24837.
8. Zhang, H.; Zhao, Y.; Yu, S.; Chen, Z. Automated quality assessment for crowdsourced test reports based on dependency parsing. In Proceedings of the 2022 9th International Conference on Dependable Systems and Their Applications (DSA), Wulumuqi, China, 4–5 August 2022; pp. 34–41.
9. Lin, C.-Y. ROUGE: A package for automatic evaluation of summaries. In Proceedings of the Workshop on Text Summarization Branches Out (ACL), Barcelona, Spain, 25 July 2004; pp. 74–81.
10. Sutton, R.S.; Barto, A.G. *Reinforcement Learning: An Introduction*, 2nd ed.; MIT Press: Cambridge, MA, USA, 2018.
11. Schulman, J.; Wolski, F.; Dhariwal, P.; Radford, A.; Klimov, O. Proximal Policy Optimization Algorithms. *arXiv* **2017**, arXiv:1707.06347. [CrossRef]
12. Reimers, N.; Gurevych, I. Sentence-BERT: Sentence embeddings using Siamese BERT-networks. In Proceedings of the EMNLP, Hong Kong, China, 3–7 November 2019; pp. 3980–3990.
13. Madaan, A.; Tandon, N.; Gupta, P.; Hallinan, S.; Gao, L.; Wiegreffe, S.; Alon, U.; Dziri, N.; Prabhumoye, S.; Yang, Y.; et al. Self-Refine: Iterative refinement with feedback. *arXiv* **2023**, arXiv:2303.17651.
14. Ouyang, L.; Wu, J.; Jiang, X.; Almeida, D.; Wainwright, C.; Mishkin, P.; Zhang, C.; Agarwal, S.; Slama, K.; Ray, A.; et al. Training language models to follow instructions with human feedback. *Adv. Neural Inf. Process. Syst.* **2022**, *35*, 27730–27744.
15. GitHub. Bug Report Summarization Benchmark dataset. GindeLab/Ease_2025_AI_model. Available online: https://github.com/GindeLab/Ease_2025_AI_model (accessed on 18 September 2025).
16. Hu, E.J.; Shen, Y.; Wallis, P.; Allen-Zhu, Z.; Li, Y.; Wang, S.; Wang, L.; Chen, W. LoRA: Low-rank adaptation of large language models. *arXiv* **2021**, arXiv:2106.09685.
17. Dettmers, T.; Pagnoni, A.; Holtzman, A.; Zettlemoyer, L. QLoRA: Efficient finetuning of quantized LLMs. *arXiv* **2023**, arXiv:2305.14314. [CrossRef]
18. Papineni, K.; Roukos, S.; Ward, T.; Zhu, W.-J. BLEU: A method for automatic evaluation of machine translation. In Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL), Philadelphia, PA, USA, 6–12 July 2002; pp. 311–318.
19. Fang, S.; Tan, Y.-S.; Zhang, T.; Xu, Z.; Liu, H. Effective prediction of bug-fixing priority via weighted graph convolutional networks. *IEEE Trans. Reliab.* **2021**, *70*, 563–574. [CrossRef]
20. Fang, S.; Zhang, T.; Tan, Y.; Jiang, H.; Xia, X.; Sun, X. RepresentThemAll: A universal learning representation of bug reports. In Proceedings of the IEEE/ACM 45th International Conference on Software Engineering (ICSE), Melbourne, Australia, 14–20 May 2023; pp. 602–614.
21. Liu, H.; Yu, Y.; Li, S.; Guo, Y.; Wang, D.; Mao, X. BugSum: Deep context understanding for bug report summarization. In Proceedings of the IEEE/ACM International Conference on Program Comprehension (ICPC), Seoul, Republic of Korea, 13–15 July 2020; pp. 94–105.
22. Shao, Y.; Xiang, B. Towards effective bug report summarization by domain-specific representation learning. *IEEE Access* **2024**, *12*, 37653–37662. [CrossRef]
23. Lamkanfi, K.; Beggas, A.; Demeyer, B.; Demeyer, S. Predicting the severity of a reported bug. In Proceedings of the IEEE International Conference on Mining Software Repositories (MSR), Cape Town, South Africa, 2–3 May 2011; pp. 274–277.
24. Sarkar, A.; Rigby, P.C.; Bartalos, B. Improving Bug Triaging with High Confidence Predictions at Ericsson. In Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME), Cleveland, OH, USA, 29 September–4 October 2019; pp. 81–91.
25. Medeiros, M.; Kulesza, U.; Coelho, R.; Bonifácio, R.; Treude, C.; Barbosa, E.A. The impact of bug localization based on crash report mining: A developers' perspective. *arXiv* **2024**, arXiv:2403.10753. [CrossRef]
26. Chen, M. Evaluating large language models trained on code. *arXiv* **2021**, arXiv:2107.03374. [CrossRef]

27. Allal, L.B.; Muennighoff, N.; Umapathi, L.K.; Lipkin, B.; von Werra, L. StarCoder: Open source code LLMs. *arXiv* **2023**, arXiv:2305.06161.
28. Rozière, B.; Gehring, J.; Gloeckle, F.; Sootla, S.; Gat, I.; Tan, X.E.; Adi, Y.; Liu, J.; Sauvestre, R.; Remez, T.; et al. Code LLaMA: Open foundation models for code. *arXiv* **2023**, arXiv:2308.12950.
29. Luo, Z.; Xu, C.; Zhao, P.; Sun, Q.; Geng, X.; Hu, W.; Tao, C.; Ma, J.; Lin, Q.; Jiang, D. WizardCoder: Empowering code LLMs to speak code fluently. *arXiv* **2023**, arXiv:2306.08568.
30. Chen, X.; Lin, M.; Schärli, N.; Zhou, D. Teaching LLMs to self-debug. *arXiv* **2023**, arXiv:2304.05125.
31. Gou, Z.; Shao, Z.; Gong, Y.; Shen, Y.; Yang, Y.; Duan, N.; Chen, W. CRITIC: LLMs can self-correct with tool-interactive critiquing. *arXiv* **2023**, arXiv:2305.11738.
32. Zhang, T.; Kishore, V.; Wu, F.; Weinberger, K.Q.; Artzi, Y. BERTScore: Evaluating text generation with BERT. In Proceedings of the ICLR, Addis Ababa, Ethiopia, 26–30 April 2020.
33. Eghbali, A.; Pradel, M. CrystalBLEU: Precisely and efficiently measuring code generation quality. In Proceedings of the ASE, Rochester, MI, USA, 10–14 October 2022; pp. 1–12.
34. Zhou, S.; Alon, U.; Agarwal, S.; Neubig, G. CodeBERTScore: Evaluating code generation with BERT-based similarity. *arXiv* **2023**, arXiv:2302.05527.
35. Wang, Z.; Zhou, S.; Fried, D.; Neubig, G. Execution-based evaluation for open-domain code generation. *arXiv* **2022**, arXiv:2212.10481.
36. Lu, S.; Guo, D.; Ren, S.; Huang, J.; Svyatkovskiy, A.; Blanco, A.; Clement, C.; Drain, D.; Jiang, D.; Tang, D.; et al. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. *arXiv* **2021**, arXiv:2102.04664. [CrossRef]
37. Wolf, T.; Debut, L.; Sanh, V.; Chaumond, J.; Delangue, C.; Moi, A.; Cistac, P.; Rault, T.; Louf, R.; Funtowicz, M.; et al. Transformers: State-of-the-Art Natural Language Processing. In Proceedings of the EMNLP 2020: System Demonstrations, Punta Cana, Dominican Republic, 16–20 November 2020; pp. 38–45.
38. Unsloth Team. Unsloth: Efficient Fine-Tuning Framework for LLMs. GitHub Repository. 2025. Available online: https://github.com/unslothai/unsloth (accessed on 18 October 2025).

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.