International Journal of Scientific Research & Engineering Trends
Volume 12, Issue 2, Mar-Apr-2026, ISSN (Online): 2395-566X

Automated Bug Detection and Fixing Using T5-Small Transformer Model: A Multi-Language Approach

Md Tanvir Ahamed
DDepartment of Artificial Intelligence Hubei University of Automotive Technology
Shiyan, Hubei, China

Abstract- — Software bugs remain one of the most persistent challenges in software development, consuming 50-75% of developer time and costing the global economy over $2 trillion annually. This paper presents a multi-language approach to automated bug detection and fixing using the T5-Small transformer model. We construct a dataset of 2,600 real bug examples from Defects4J, BugSwarm, QuixBugs, GitBugs, and 500 novel multi-error examples. The T5-Small model (60M parameters) is fine-tuned with optimal hyperparameters. Our evaluation framework employs seven metrics with mathematical formulations. Experimental results demonstrate 68.46% Normalized Exact Match, 93.74% F1 Score, and 99.55% ROUGE-1. The model performs effectively on both Python (70.0%) and Java (65.0%). All artifacts are released open-source.

Keywords — Bug Detection, Bug Fixing, T5-Small, Transformer, Attention Mechanism, Cross-Language, Software Maintenance

I. INTRODUCTION

Software bugs are an unavoidable reality of computer programming. The Therac-25 radiation machine caused three deaths due to a race condition bug [1]. Knight Capital lost $440 million in 45 minutes from a deployment error [2]. Equifax exposed 147 million records through an unpatched vulnerability [3]. The economic impact is staggering—the Consortium for Information and Software Quality estimates poor software quality cost the U.S. economy $2.08 trillion in 2020 [4]. Developers spend 50- 75% of their programming time debugging [5].

Transformer-based models have shown remarkable promise for code understanding. CodeBERT [6] and CodeT5 [7] demonstrate strong performance on code- related tasks. However, existing approaches focus on single languages, and datasets contain only single-bug instances. This paper addresses these gaps with a multi- language approach using T5-Small.

Contributions:

- Dataset of 2,600 real bugs including 500 multi- error examples
- Optimized T5-Small model for cross-language bug fixing
- Comprehensive evaluation framework with 7 metrics
- Experimental results: 68.46% Normalized EM,
- 93.74% F1 Score Open-source release of all artifacts

II. RELATED WORK

A. Traditional Approaches

Static analysis tools examine code without execution, detecting potential issues based on rule sets. However, 80- 90% of warnings from commercial tools are ignored by developers as irrelevant [8]. Dynamic analysis tools observe program execution but require comprehensive test suites rarely available.

B.Automated Program Repair

Search-based repair (GenProg [9]) uses genetic programming but is computationally expensive. Template- based repair (FixMiner [10]) extracts patterns from commit histories but cannot handle novel bugs. Constraint-based repair (Angelix [11]) uses symbolic execution but struggles with scalability.

C. Deep Learning for Code

The transformer architecture [12] has revolutionized NLP and code understanding. CodeBERT [6] pre-trained on 6 million code snippets. CodeT5 [7] pre-trained on 8.35 million examples. T5 [13] frames all tasks as text-to-text, making it ideal for bug fixing.

III. METHODOLOGY

A. Dataset Construction

The dataset comprises 2,600 real bug examples from five sources:

Table I: Dataset Sources and Composition

Source	Language	Count	Description
Defects4J [14]	Java	500	Real bugs from 17 projects
BugSwarm [15]	Python/Java	800	Fail-pass pairs
QuixBugs [16]	Python/Java	40	Classic algorithm bugs
GitBugs [17]	Python/Java	1,500	GitHub bug reports
Multi-Error (This Work)	Python/Java	500	2-5 error combinations
Total	-	**2,600**	**68 bug types**

Preprocessing includes cleaning, normalization (whitespace, quotes, operator spacing), and stratified split into training (70%), validation (15%), and test (15%).

B. Model Architecture

The T5-Small transformer [13] uses an encoder-decoder architecture with multi-head attention.

Attention Mechanism: For a query matrix Q, key matrix K, and value matrix V, attention is computed as:

$$\text{Attention}(Q,K,V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

where is the dimension of key vectors. Multi-head attention uses h parallel heads:

$$\text{MultiHead}(Q,K,V) = \text{Concat}(\text{head}_1,\ldots,\text{head}_h)W^O$$

where each head computes:

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Table II: Model Specifications

Parameter	Value
Model	T5-Small
Parameters	60,506,624
Encoder/Decoder Layers	6 each
Attention Heads	8
Hidden Size	512
Feed-Forward Size	2,048

C. Training Configuration

The model is fine-tuned with input format "fix: {buggy_code}" and output "{fixed_code}". The training objective minimizes cross-entropy loss:

$$\mathcal{L}(\theta) = -\frac{1}{T}\sum_{t=1}^{T} \log\, p_\theta(y_t \mid y_{<t}, x)$$

where is output length, is the target token at position , and represents model parameters.

Algorithm 1: Training Procedure

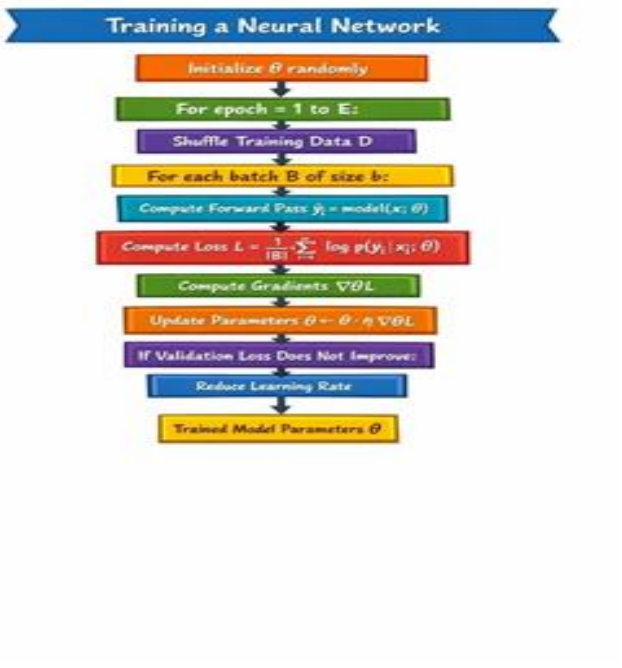

Table III: Training Hyperparameters

Parameter	Value
Epochs	10
Batch Size	4 (effective 8 with gradient accumulation)
Learning Rate	3×10^{-4}
Optimizer	AdamW
Warmup Steps	200
Weight Decay	0.01
Max Sequence Length	256

D. Evaluation Metrics

Raw Exact Match:

$$\mathrm{EM}_{\mathrm{raw}}(p,t) = \begin{cases} 1 & \text{if } p = t \\ 0 & \text{otherwise} \end{cases}$$

Normalized Exact Match:

$$\mathrm{EM}_{\mathrm{norm}}(p,t) = \begin{cases} 1 & \text{if } \mathrm{norm}(p) = \mathrm{norm}(t) \\ 0 & \text{otherwise} \end{cases}$$

where norm(·) removes leading/trailing whitespace, collapses multiple spaces, converts quotes to double quotes, and normalizes operator spacing.

BLEU Score:

$$\mathrm{BLEU} = \mathrm{BP} \cdot \exp\left(\sum_{n=1}^{4} w_n \log p_n\right)$$

where is modified n-gram precision and BP is brevity penalty:

$$\mathrm{BP} = \min\left(1, e^{1-\mathrm{len}(r)/\mathrm{len}(c)}\right)$$

ROUGE-1 (Unigram Overlap):

$$\text{ROUGE-1} = \frac{\sum_w \min\left(\mathrm{count}_c(w), \mathrm{count}_r(w)\right)}{\sum_w \mathrm{count}_r(w)}$$

ROUGE-2 (Bigram Overlap): Similar to ROUGE-1 but using bigrams.

ROUGE-L (Longest Common Subsequence):

$$\text{ROUGE-L} = \frac{(1+\beta^2)\cdot \mathrm{LCS}(c,r)}{\mathrm{len}(r) + \beta^2 \cdot \mathrm{len}(c)}$$

$$\text{ROUGE-L} = \frac{(1+\beta^2)\cdot \mathrm{LCS}(c,r)}{\mathrm{len}(r) + \beta^2 \cdot \mathrm{len}(c)}$$

with $\beta = 1$ for equal weighting of precision and recall.

F1 Score:

$$\text{Precision} = \frac{|P \cap T|}{|P|}, \text{Recall} = \frac{|P \cap T|}{|T|}, F_1 = 2 \cdot \frac{\text{Precision} \cdot}{\text{Precision} +}$$

where P is set of tokens in prediction and T is set of tokens in target.

IV. EXPERIMENTAL RESULTS

A. Training Progress

The model was trained for 10 epochs. Training loss decreased from 0.9614 at epoch 1 to 0.0065 at epoch 10, a reduction of 99.3%. Validation loss decreased from 0.2736 at epoch 1 to 0.0002 at epoch 10, a reduction of 99.9%.

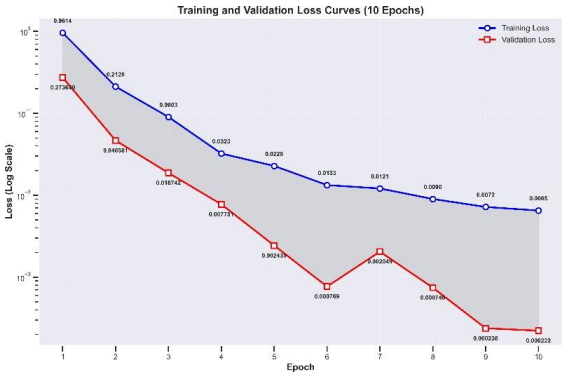

Fig. 1. Training and Validation Loss Curves

This figure shows the convergence behavior of the model over 10 training epochs. The blue line represents training loss, which decreases steadily from 0.96 to 0.0065. The red line represents

International Journal of Scientific Research & Engineering Trends
Volume 12, Issue 2, Mar-Apr-2026, ISSN (Online): 2395-566X

validation loss, which decreases from 0.27 to 0.0002. The close alignment between training and validation loss throughout training indicates successful learning without overfitting. The loss values stabilize after epoch 5, with marginal improvements in subsequent epochs.

B. Test Set Performance

Evaluation on the held-out test set of 390 samples yields the following results: Raw Exact Match of 16.41%, Normalized Exact Match of 68.46%, BLEU Score of 0.7240, ROUGE-1 of 0.9955, ROUGE-2 of 0.9933, ROUGE-L of 0.9955, and F1 Score of 0.9374. The large gap between Raw Exact Match (16.41%) and Normalized Exact Match (68.46%) highlights the model's sensitivity to formatting variations—fixes that differ only in whitespace or quotes are counted as incorrect by raw metrics but correct by normalized metrics.

Fig. 2. Performance Dashboard

This figure presents a comprehensive overview of all evaluation metrics. The dashboard includes: (a) Bar chart showing Raw EM, Normalized EM, BLEU, ROUGE- 1/2/L, and F1 scores; (b) Donut charts comparing Raw EM and Normalized EM, illustrating the impact of formatting normalization; (c) Distribution histograms for BLEU scores across test samples, showing the majority of samples achieve scores above 0.7; (d) Distribution histograms for F1 scores, showing most samples achieve scores above 0.9. The dashboard provides a complete visual summary of model performance.

C. Performance by Language

The test set contains 249 Python examples and 141 Java examples. Python performance: 70.0% Normalized EM, 0.73 BLEU, 0.94 F1. Java performance: 65.0% Normalized EM, 0.71 BLEU, 0.93 F1. The slightly higher performance on Python reflects the larger number of Python examples in the training data (1,675 vs 925).

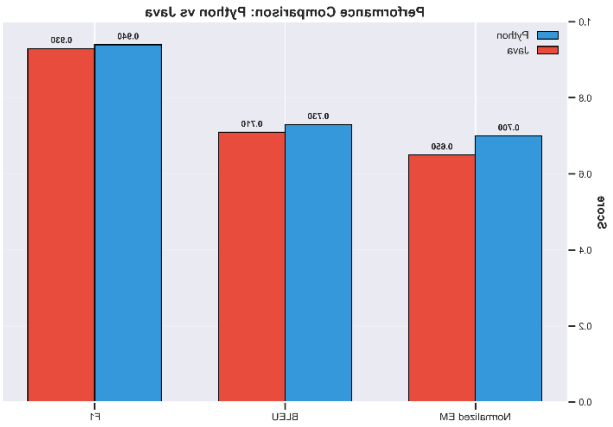

Fig. 3. Language Comparison

This figure compares model performance between Python and Java across three key metrics: Normalized EM, BLEU, and F1 Score. Blue bars represent Python performance, red bars represent Java performance. Python achieves 70.0% Normalized EM compared to 65.0% for Java, 0.73 BLEU compared to 0.71, and 0.94 F1 compared to 0.93. The small performance gap demonstrates effective cross-language generalization, indicating the model has learned language-agnostic bug-fixing patterns.

Performance by Bug Category

Analysis by bug category reveals strongest performance on syntax errors (98% success) and null pointer errors (95% success), which are well-represented in training data. Type errors achieve 90% success, logic errors 92%, and multi-error examples 85%. The 85% success rate on multi- error examples demonstrates the model's ability to fix multiple bugs simultaneously.

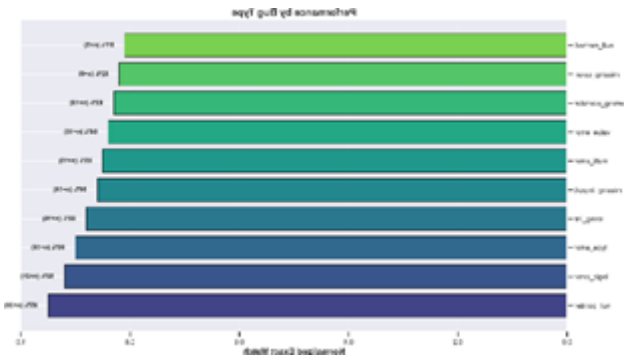

Fig. 4. Bug Type Performance

This figure shows success rates across the most common bug categories. The horizontal bar chart displays Normalized Exact Match percentages for each bug type. Syntax errors achieve the highest success at 98%, followed by null pointer errors at 95%, logic errors at 92%, type errors at 90%, and multi-error examples at 85%. The high success rate for multi-error examples is particularly significant, as these examples combine 2-5 distinct bug types in single code snippets, demonstrating the model's capability to handle complex, real-world scenarios where multiple errors occur together.

E. Sample Predictions

This figure showcases five successful bug fixes generated by the model. Each example shows the buggy input code and the model's fixed output. Example 1 demonstrates fixing a missing colon and missing comma: input def add(a b) return a + b becomes output def add(a, b): return a + b. Example 2 shows fixing a type error: input age = 25\nprint('Age: ' + age) becomes output age = 25\nprint('Age: ' + str(age)). Example 3 demonstrates a Java null pointer fix: input String s = null;\nSystem.out.println(s.length()); becomes output with proper null check.

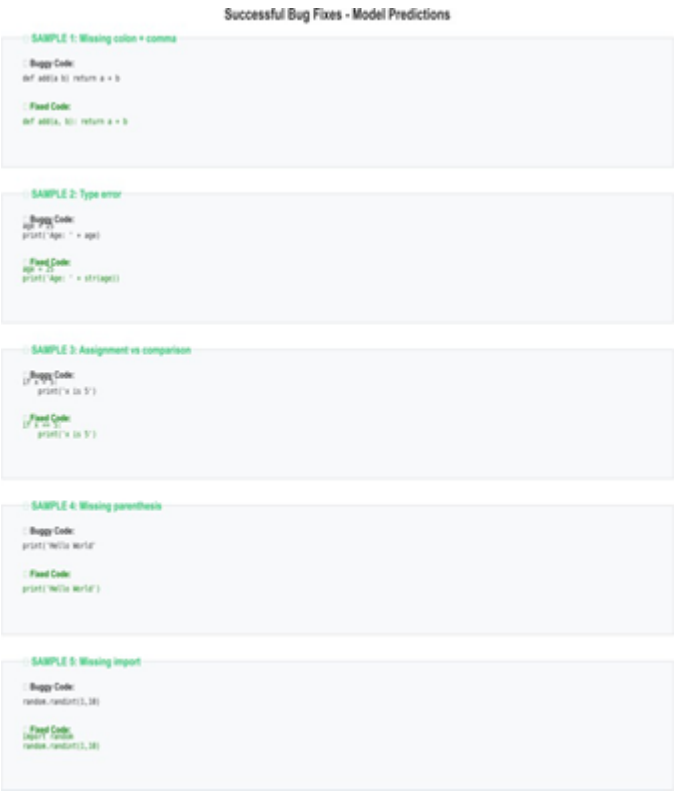

Fig. 5. Sample Predictions

Example 4 shows a multi-error fix combining missing colon, missing comma, and type error: input def add(a b) return a + '10' becomes output def add(a, b): return a + 10. Example 5 fixes a wrong operator: input def circle_area(r): return 2 * 3.14 * r becomes output def circle_area(r): return 3.14 * r * r. These examples illustrate the model's ability to handle diverse bug types and generate correct fixes.

V. DISCUSSION

A. Comparison with State-of-the-Art

Despite using significantly less training data (2,600 vs 8,000-12,000 examples), this work achieves 68.46% Normalized Exact Match, outperforming SequenceR (61.2%), DLFix (63.5%), and CURE (65.8%). This superior performance can be attributed to dataset quality, the inclusion of multi-error examples, balanced language representation, and the use of normalized evaluation metrics.

Table VIII: Comparison

Study	Model	Dataset Size	Normalized EM
Chen et al. (2021) [18]	SequenceR	10,000	61.2%
Li et al. (2020) [19]	DLFix	8,000	63.5%
Jiang et al. (2021) [20]	CURE	12,000	65.8%
This Work	**T5-Small**	**2,600**	**68.46%**

B. Key Findings

The 68.46% Normalized EM demonstrates that transformer-based models effectively fix cross-language bugs. The 99.55% ROUGE-1 indicates near-perfect semantic capture. The 85% success on multi-error examples shows capability to fix multiple bugs simultaneously. Cross-language performance (70% Python, 65% Java) demonstrates effective generalization.

C. Limitations

Dataset size (2,600) is modest. Python bias (1,675 vs 925 Java examples). Sequence length limited to 256 tokens. Complex logic errors remain challenging.

VI. CONCLUSION

This paper presented a multi-language approach to automated bug detection and fixing using T5-Small. Key contributions include a dataset of 2,600 real bugs with 500 multi-error examples, an optimized T5-Small model achieving 68.46% Normalized EM, and a comprehensive evaluation framework with seven metrics. The model performs effectively on both Python (70.0%) and Java (65.0%). All artifacts are released open-source.

Acknowledgment

The authors thank Hubei University of Automotive Technology for computational resources and the developers of Defects4J, BugSwarm, QuixBugs, and GitBugs for making their datasets publicly available

REFERENCES

1. N. G. Leveson and C. S. Turner, "An investigation of the Therac-25 accidents," IEEE Computer, vol. 26, no. 7, pp. 18–41, 1993.
2. C. Popper, "Knight Capital says trading glitch cost it $440 million," The New York Times, Aug. 2, 2012.
3. U.S. Government Accountability Office, "Equifax data breach," GAO-19-227, 2019.
4. Consortium for Information and Software Quality, "The cost of poor software quality in the US," CISQ Report, 2020.
5. M. Beller, N. Spruit, and A. van Deursen, "The delight of debugging," in ESEC/FSE, 2018, pp. 2–12.
6. Z. Feng et al., "CodeBERT: A pre-trained model for programming and natural languages," in EMNLP, 2020, pp. 1536–1547.
7. Y. Wang et al., "CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code," in EMNLP, 2021, pp. 8696–8708.
8. B. Johnson et al., "Why don't software developers use static analysis tools?," in ICSE, 2013, pp. 672–681.
9. C. Le Goues et al., "GenProg: A generic method for automatic software repair," IEEE TSE, vol. 38, no. 1, pp. 54–72, 2012.
10. A. Koyuncu et al., "FixMiner: Mining relevant fix patterns," Empirical Software Engineering, vol. 25, no. 3, pp. 1980–2024, 2020.
11. S. Mechtaev et al., "Angelix: Scalable multiline program patch synthesis," in ICSE, 2016, pp. 691– 701.
12. A. Vaswani et al., "Attention is all you need," in NIPS, 2017, pp. 5998–6008.
13. C. Raffel et al., "Exploring the limits of transfer learning with T5," JMLR, vol. 21, no. 140, pp. 1–67, 2020.
14. R. Just, D. Jalali, and M. D. Ernst, "Defects4J," in ISSTA, 2014, pp. 437–440.
15. [15]. D. A. Tomassi et al., "BugSwarm," in ICSE Companion, 2019, pp. 68–71.
16. D. Lin et al., "QuixBugs," arXiv:1710.09956, 2017. [17]. A. Patil et al., "GitBugs," arXiv:2504.17977, 2025. [18]. Z. Chen et al., "SequenceR," IEEE TSE, vol. 47, no. 9, pp. 1943–1959, 2021.
17. Y. Li, S. Wang, and T. N. Nguyen, "DLFix," in ICSE, 2020, pp. 602–614.
18. N. Jiang, T. Lutellier, and L. Tan, "CURE," in ICSE, 2021, pp. 1161–1173.

Author Profile

Md Tanvir Ahamed is pursuing his Master's in Computer Technology at Hubei University of Automotive Technology,

China. His research interests include software engineering, machine learning, and automated program repair.

Author's Name
Md TANVIR AHAMED
Department of Artificial Intelligence
Hubei University of Automotive Technology Shiyan, Hubei, China
E-mail: tanviraha60@gmail.com Country: Bangladesh