iJRASET
International Journal For Research in Applied Science and Engineering Technology

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 14 **Issue:** IX **Month of publication:** September 2026

DOI: https://doi.org/10.22214/ijraset.2026.84782

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 14 Issue IX Sep 2026- Available at www.ijraset.com

Curriculum Learning for Fine-Tuning Code-Review Language Models: A Case Study in Diagnosing and Correcting Fabrication in Vulnerability Explanation

Jesshwanth S Joshua[1], Dr V. Shanmuganeethi[2]
[1]*M.Tech AIML (Scholar),* [2]*Supervisor, Professor, Department of Computer Science and Engineering, National Institute of Technical Teachers Training and Research (NITTTR), Chennai – 600113, India*

Abstract: ***Automated code-review systems can identify potential defects, but they often provide limited explanations for developers. This study investigates a curriculum-based approach for fine-tuning CodeT5+ 770M to generate code-review and vulnerability explanations for Python and JavaScript. Training was divided into three stages: code understanding, semantic review, and vulnerability explanation. Rehearsal examples from earlier stages were included during later training to reduce forgetting. We also compared this staged approach with an earlier mixed-phase setup that trained on all tasks together, to see whether staging changed how the training process behaved. During Stage 3, we found that using case-specific CVE descriptions as targets led the model to generate unsupported package names, versions, and repository references. We replaced these targets with CWE-level descriptions and retrained the model. In a manual review of 20 outputs, the revised model did not produce the same type of unsupported specific details. BLEU increased from 7.54 to 34.43, and ROUGE-1 increased from 0.2765 to 0.4180. However, the revised model still confused CWE categories: exact category agreement was 27.3% after excluding samples with incomplete ground-truth labels, and Cross-Site Scripting was predicted more often than its true frequency. We also tested whether increasing the Stage 2 target length from 192 to 512 tokens improved later Stage 3 performance. Across six epochs, the average validation-loss difference was +0.0008, and the final Stage 3 metrics showed no meaningful change. These results show that target design and manual inspection are important when evaluating generated vulnerability explanations, and that negative results, like the target-length test here, are worth reporting alongside positive ones.***

Keywords: Code Review Automation, Large Language Models, Curriculum Learning, Catastrophic Forgetting, Vulnerability Detection, CodeT5, Hallucination, Static Analysis.

I. Introduction

Manual code review is difficult to scale as software projects grow. Static-analysis tools can detect known patterns, such as coding errors or security issues, but their output is often limited to warnings or rule violations. Large language models can complement these tools by generating explanations that may help developers understand a finding and decide how to address it. This creates two practical training questions. First, when a model must learn code understanding, review-comment generation, and vulnerability explanation, is it better to train these tasks in stages or to mix them throughout training? Second, how can unsupported details in security explanations be detected and reduced? This question is important because a fluent but incorrect explanation may be misleading in a security setting.

We address both questions empirically by fine-tuning CodeT5+ 770M on Python and JavaScript code. Our contributions:

1) A three-stage curriculum (syntax → semantic review → vulnerability) with per-stage rehearsal to reduce catastrophic forgetting. This design was motivated by an observed forgetting failure in a naive sequential fine-tuning baseline (Section III-B).
2) A controlled ablation against a mixed-phase (jointly-trained) baseline using the same base model and a similar compute budget.
3) A fabrication failure mode in vulnerability-explanation generation that we identified, reproduced, and corrected. We analyze its likely cause and apply a dataset-level (not model-level) fix.
4) A description of the fix's remaining limitation (category confusion under class imbalance) and a secondary experiment that did not show an improvement. We include this negative result because such findings are rarely reported.

II. LITERATURE REVIEW

Automated code review has been studied using static analysis, machine learning, and large language models. Tang et al. [1] present CodeAgent, a multi-agent system that performs vulnerability analysis, format checking, and revision suggestion. Pearce et al. [2] evaluate large language models for zero-shot vulnerability repair and report a reliability gap between synthetic and real-world vulnerabilities. Curriculum learning and continual-learning methods are also relevant to this work. Naïr et al. [3] show that curriculum learning can help small code language models, although the effect depends on the task. Buzzega et al. [4] compare approaches for reducing catastrophic forgetting and find that rehearsal-based methods perform strongly in their experiments. Lipp et al. [5] evaluate static C/C++ analyzers on real vulnerabilities and show that detection performance remains limited. Together, these studies motivate examining both model training design and the quality of generated security explanations.

No.	Reference	Focus	Limitation
1	Tang et al. (2024) [1]	Multi-agent LLM system (CodeAgent) for vulnerability analysis, format-consistency checking, and revision suggestion	Orchestrates off-the-shelf models (GPT-4, CodeBERT); does not check whether generated explanations are factually grounded
2	Pearce et al. (2023) [2]	Zero-shot LLM vulnerability repair using off-the-shelf models (Codex, Jurassic-1)	Reliability gap on real-world bugs (vs. synthetic ones) is reported but never diagnosed
3	Naïr et al. (2024) [3]	Curriculum learning for small code language models	Orders difficulty within one fixed task only; no rehearsal mechanism against forgetting
4	Buzzega et al. (2020) [4]	Rehearsal (Dark Experience Replay) vs. regularization (EWC, SI) for catastrophic forgetting	Evaluated on general vision/MNIST benchmarks, not code models or LLMs
5	Lipp et al. (2022) [5]	Real-world effectiveness of static C/C++ analyzers against validated CVEs	Detection-only; misses 47-80% of vulnerabilities and provides no explanation layer

TABLE I. LITERATURE REVIEW

The reviewed studies address related parts of the problem. Code-review and repair systems ([1], [2]) show that LLMs can assist with software-engineering tasks, but they do not always assess whether generated explanations are supported by the available evidence. Curriculum-learning and rehearsal studies ([3], [4]) provide motivation for staged training and forgetting mitigation, although they are not focused on vulnerability-explanation generation. Static-analysis studies ([5]) show that detection tools have important limitations, which motivates using language models as an explanation layer rather than relying on warning messages alone.

III. METHODOLOGY

A. Model and Task Design

We fine-tune CodeT5+ 770M (Salesforce/codet5p-770m). We selected this model after comparing it with CodeT5-base and CodeT5-large. The task is formulated as sequence-to-sequence generation: given a code input tagged with its language and the nature of the request, generate a natural-language explanation.

All inputs share one format:

[LANG] {python|javascript} [DETECTED_BY] ai [ISSUE] {code_understanding|full_review|vulnerability} [CODE] {code}

B. Model Selection and the Case Against Naive Sequential Fine-Tuning

A prior iteration of this work (CodeT5-base) achieved BLEU 35.03 / ROUGE-1 0.6765 on a general review-comment task. Scaling to CodeT5-large via sequential fine-tuning (continuing training across successive data phases without any rehearsal) caused catastrophic forgetting: validation loss degraded from 1.31 to 3.36 over the course of training. This observation informed two later decisions. First, we selected CodeT5+ 770M instead of CodeT5-large. Second, we included rehearsal examples during staged training to reduce the risk of forgetting earlier tasks.

C. Curriculum Design

Three stages, each with a strictly separate, self-contained dataset (no blending across task types within a stage) and its own fresh 80/10/10 train/val/test split (seed=42).

Stage	Task	Rationale for ordering
1 — Syntax/structure	Code understanding (docstring-style generation)	Foundational — establishes basic code comprehension before review-level reasoning
2 — Semantic/review	Code review comment generation	The idea was that fixing basic understanding first would leave fewer semantic issues to handle at this stage.
3 — Vulnerability	Security vulnerability explanation	The assumption was that resolving semantic issues earlier would rule out some vulnerability types before this stage.

TABLE II. CURRICULUM STAGE ORDERING RATIONALE

D. Datasets

Stage 1 contained 20,000 CodeSearchNet examples: 10,000 Python and 10,000 JavaScript function–docstring pairs. CodeSearchNet was used because its function-level code examples had already been extracted and verified during dataset construction.

Stage 2 contained 49,193 examples before rehearsal. The data came from three sources: ronantakizawa/github-codereview, Microsoft's CodeReviewer dataset, and YoungPhlo's forum-sourced review pairs. We applied language-specific quality thresholds to the GitHub review data: 0.7 for Python and 0.5 for JavaScript. We manually inspected examples in the 0.5–0.7 range and did not find a clear decline in JavaScript quality. Including these examples increased the number of usable JavaScript samples and improved the Python-to-JavaScript balance from about 6.6:1 to 3.4:1.

From CodeReviewer, we used only the validation_generation split because the larger train_generation split did not include language metadata in the sources we examined. For the forum-sourced data, HTML content was removed before training. We excluded Tomo-Melb/CodeReviewQA because it is a benchmark dataset that was not licensed for training.

Stage 3 contained 1,872 examples before rehearsal. The data came from two vulnerability-pattern datasets—darkknight25/software_vulnerabilities_dataset and darkknight25/Vulnerable_Programming_Dataset—and CVEfixes. We selected these sources because they contained code and explanatory text relevant to vulnerability analysis.

We excluded several other candidate datasets. One contained only binary vulnerability labels, another was fully synthetic or LLM-generated, and another had missing descriptions for 78% of examples. The usable dataset size was lower than our initial estimate of approximately 3,100 examples. This reflects the limited availability of Python and JavaScript vulnerability-explanation data, since many larger vulnerability datasets focus on C and C++.

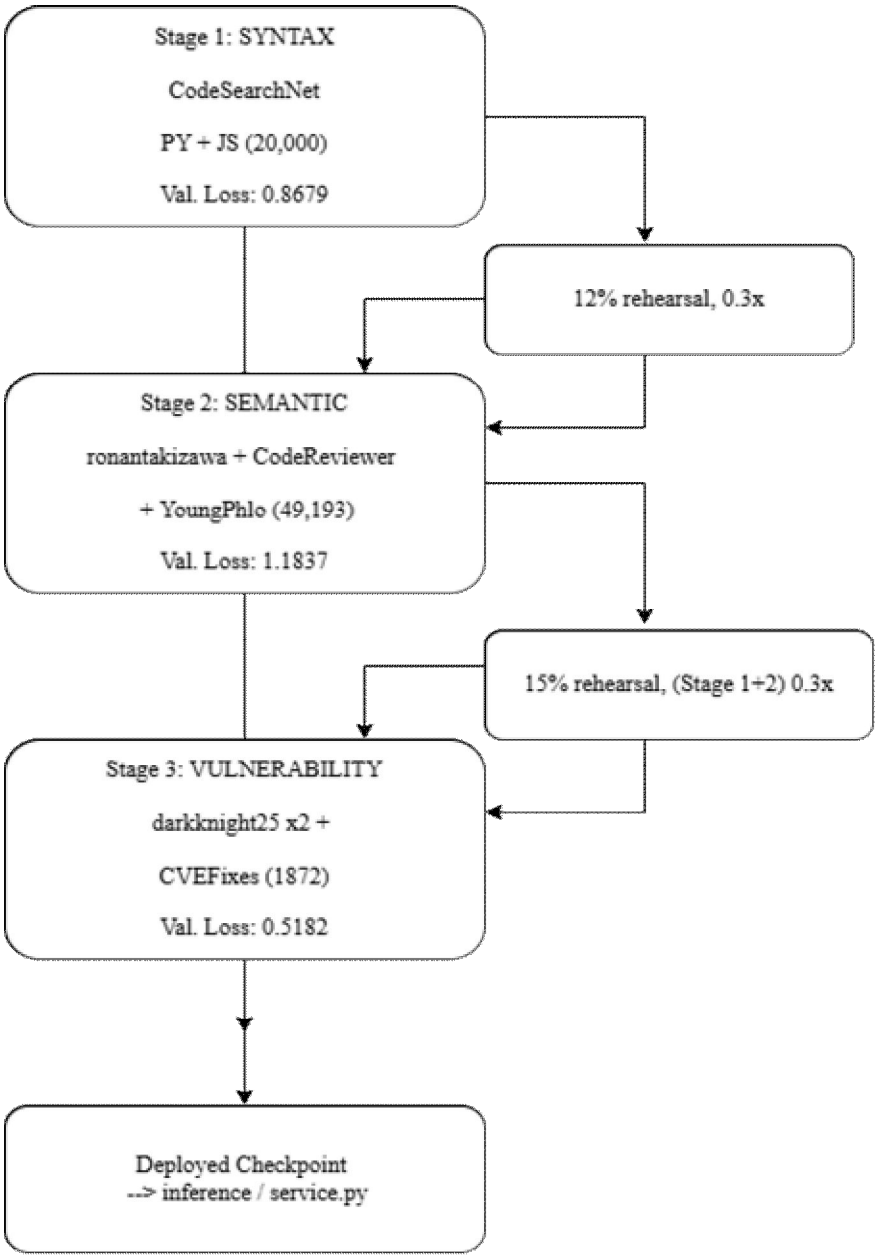


Fig. 1 Three-Stage Curriculum Pipeline

E. Training Configuration

Setting	Stage 1	Stage 2	Stage 3
Max input length	1024 tokens	1024 tokens	1024 tokens
Max target length	192 tokens	192 tokens	512 tokens
Learning rate	3e-5	2e-5	1e-5
Rehearsal	N/A	12% of Stage 1 train, 0.3× weight	15% of Stage 1+2, 0.3× weight
Optimizer	AdamW8bit	AdamW8bit	AdamW8bit

TABLE III. STAGE-WISE TRAINING CONFIGURATION

Stage 3's target-length increase to 512 was a deliberate, data-driven decision: a post-hoc distribution check found 23.4% of Stage 2's test targets exceeded 192 tokens (a measured contributing factor to Stage 2's lower BLEU/ROUGE), so Stage 3's actual target-length distribution was checked before training rather than defaulting to the prior stage's setting — confirming only ~0.8% of Stage 3 targets would exceed 512.

F. Ablation Baseline: Mixed-Phase Training

We also report results from an earlier mixed-phase training run using the same base model. In that setup, examples from all dataset sources were mixed at different proportions during each phase. The mixed-phase approach was initially used because of limited GPU availability rather than as a planned curriculum-learning baseline. It is included here as a descriptive comparison with the staged training procedure.

IV. THE VULNERABILITY-EXPLANATION FABRICATION FAILURE

A. Discovery

In the first Stage 3 run, we used the cve_description field from CVEfixes as the generation target. These descriptions contain case-specific information, including package names, versions, and repository references. The model trained for six epochs and reached a best validation loss of 2.0823. Its automatic scores were BLEU 7.54, ROUGE-1 0.2765, and BERTScore F1 0.8601. We then manually reviewed 20 generated outputs. Several outputs included package names, version numbers, or repository references that were not supported by the input code. We observed this pattern in checkpoints from epochs 3 through 6. Therefore, the issue did not appear to be caused by selecting a single checkpoint or stopping training at a particular epoch.

B. Root Cause

We believe that the target field was a major cause of this behavior. The cve_description targets contained detailed facts that differed from one CVE to another. With approximately 1,500 non-rehearsal training examples, the model may not have received enough examples to reliably associate those details with the corresponding code. Instead, it appeared to learn the style of detailed vulnerability descriptions and sometimes generated unsupported specifics. This interpretation is based on the behavior observed before and after changing the target field. It does not rule out other factors, such as dataset size, class imbalance, or the limited context available in individual code samples.

C. Fix and Retrain

We replaced the cve_description target with the cwe_description field from CVEfixes. Unlike CVE descriptions, CWE descriptions provide category-level explanations and are shared across examples within a CWE category. The selected field was complete for the 1,560 samples used in this retraining experiment. We also used repetition_penalty=1.3 and no_repeat_ngram_size=3 during generation. These settings were added because two of the 20 original outputs repeated the same phrase several times. The model was retrained from the Stage 2 checkpoint using the same training settings as the original Stage 3 run, except for the new target field.

D. Results

The lower validation loss and higher BLEU and ROUGE scores should be interpreted carefully. CWE-level descriptions are more standardized than case-specific CVE descriptions, so the revised task is easier to predict. The automatic-score improvement therefore does not by itself show better vulnerability understanding. The main observation is that the revised model did not produce the same type of unsupported specific details in the 20 manually reviewed outputs.

Result Classes	Original (cve_description)	Retrained (cwe_description)
Best val loss	2.0823	0.5182
BLEU	7.54	34.43
ROUGE-1	0.2765	0.4180
ROUGE-2	—	0.2863
ROUGE-L	—	0.3768
BERTScore F1	0.8601	0.8857
Fabrication (20-sample manual review)	Confirmed present	0/20 — eliminated

TABLE IV. FABRICATION FIX: RESULTS BEFORE AND AFTER CHANGING THE TARGET FIELD

E. Residual Limitation: Category Confusion Under Class Imbalance

On the full Stage 3 test set of 188 samples, the model predicted the exact CWE category for 22.3% of examples. After excluding 56 examples with the ground-truth label "Insufficient Information," exact agreement was 27.3%. We excluded these examples because this label reflects incomplete CWE information in the source data rather than a specific vulnerability category. Cross-Site Scripting was predicted more frequently than it occurred in the cleaned test set: 37.1% of predictions compared with 19.7% of ground-truth labels. XSS also accounted for 19% of the Stage 3 training data. This pattern suggests that the model may favour the most common category when it is uncertain. Some incorrect predictions involved related security concepts. For example, some SSRF, CSRF, access-control, and path-traversal examples were predicted as XSS. ReDoS examples were sometimes predicted as broader resource-consumption issues. The revised target reduced unsupported case-specific details, but it did not provide reliable fine-grained CWE classification.

V. EVALUATION

A. Per-Stage Results

Stage	BLEU	ROUGE-1	ROUGE-2	ROUGE-L	BERTScore F1
Stage 1	50.96	0.6639	0.5740	0.6441	0.9349
Stage 2	14.64	0.3898	0.3039	0.3662	0.8912
Stage 3	34.43	0.4180	0.2863	0.3768	0.8857

TABLE V. PER-STAGE RESULTS

B. Interpreting Stage 2's Low n-gram Scores

Stage 2 had lower BLEU and ROUGE scores than Stage 1, while BERTScore remained relatively high (0.8912 compared with 0.9349). Review-comment generation is a one-to-many task: more than one comment may be appropriate for the same code snippet. As a result, an output can be semantically relevant while sharing few n-grams with the reference comment. Target length was another possible factor. A length analysis showed that 23.4% of Stage 2 test targets were longer than the 192-token limit. These examples may have been truncated during training or evaluation, which could reduce overlap-based metric scores.

C. Secondary Validation Experiment: Target-Length Ablation

We evaluated whether increasing the maximum Stage 2 target length from 192 to 512 tokens affected later performance. Stage 2 was retrained from the Stage 1 checkpoint for six epochs, with all other settings unchanged.

Epoch	192-token (real Stage 2)	512-token	Difference
1	1.3831	1.3797	−0.0034
2	1.3032	1.3075	+0.0043
3	1.2553	1.2437	−0.0116
4	1.2192	1.2333	+0.0141
5	1.1939	1.1908	−0.0031
6	1.1837	1.1882	+0.0045

TABLE VI. TARGET-LENGTH ABLATION, PER-EPOCH VALIDATION LOSS

Across the six epochs, the average validation-loss difference between the two target-length settings was +0.0008. The differences changed direction across epochs and did not show a consistent pattern. We then trained a new Stage 3 model from the 512-token Stage 2 checkpoint. The resulting Stage 3 scores were validation loss 0.5164, BLEU 34.40, ROUGE-1 0.4170, and BERTScore F1 0.8862. These values were very close to those of the original Stage 3 model. We therefore found no clear evidence that the longer Stage 2 target length improved downstream Stage 3 performance. The larger target length may still help preserve long review comments that would otherwise be truncated. However, in this experiment, it did not improve the aggregate validation-loss or final Stage 3 metrics.

D. Curriculum vs. Mixed-Phase Ablation

Regime	Stage/Phase	Best val loss
Mixed-phase	Phase 1	2.0513
Mixed-phase	Phase 2	1.9563
Curriculum	Stage 1 (Syntax)	0.8679
Curriculum	Stage 2 (Semantic)	1.1837
Curriculum	Stage 3 (Vulnerability)	0.5182

TABLE VII. MIXED-PHASE VS. CURRICULUM BEST VALIDATION LOSS

Table VII lists the best validation losses from the earlier mixed-phase run and the staged training runs. These values should not be treated as a direct performance comparison because the training objectives differed. The mixed-phase model was trained on a blend of all seven dataset sources, whereas each curriculum stage was trained on a separate task. In the mixed-phase run, Phase 1 reached a best validation loss of 2.0513 and Phase 2 reached 1.9563. Phases 3 and 4 were not run after the training procedure was redesigned. The staged runs reached validation losses of 0.8679, 1.1837, and 0.5182 for Stages 1, 2, and 3, respectively. These values are reported for context rather than as evidence that one regime outperformed the other by a particular numerical margin.

One practical benefit of the staged design was that Stage 3 could be examined separately. This made it easier to notice that the unsupported details appeared during vulnerability-explanation training and to test whether the target field contributed to the problem. In the mixed-phase setup, the same CVEfixes examples were combined with data from other tasks, which would have made the source of the failure more difficult to investigate. This is a practical advantage of task separation, although it is not a direct quantitative comparison of overall model quality.

VI. LIMITATIONS

The input format was simpler than originally planned. The initial design included a tool-provenance field, [DETECTED_BY] {tool_name}, to distinguish findings from static-analysis tools and findings produced by the model. In the final training data, however, all examples used [DETECTED_BY] ai, and the [ISSUE] field included only three broad categories. The model also showed substantial CWE-category confusion, particularly for common categories such as Cross-Site Scripting. We did not evaluate class-balanced sampling or loss reweighting because of the project timeline. The vulnerability-explanation dataset was relatively small for Python and JavaScript, and this limits the conclusions that can be drawn from the Stage 3 results. The deployment approach chunks long code inputs, so the model may miss relationships between methods or between separate chunks of a file. Lightweight class-level headers provide limited additional context but do not solve this issue. Finally, the manual evaluation was limited to 20 outputs. A larger study with multiple reviewers and ratings for factual correctness, clarity, and usefulness would provide a stronger evaluation.

VII. CONCLUSION AND FUTURE WORK

This study evaluated a three-stage fine-tuning procedure for generating code-review and vulnerability explanations with CodeT5+ 770M. The staged setup allowed the vulnerability-explanation task to be evaluated separately from code understanding and review-comment generation. When the model was trained with case-specific CVE descriptions, some outputs included unsupported package names, versions, and repository references. Replacing these targets with CWE-level descriptions removed this behavior in the 20 manually reviewed outputs and improved automatic evaluation scores. However, the revised task used more standardized targets, so the metric improvement should not be interpreted only as improved vulnerability understanding. Fine-grained CWE classification remained limited, with category confusion that was likely affected by class imbalance. Increasing the Stage 2 target length from 192 to 512 tokens also did not produce a clear improvement in downstream Stage 3 performance. We think findings like this are worth reporting even when they don't show an improvement, since knowing what does not help is still useful for anyone building on this work. Future work should evaluate class-balanced training, larger manually reviewed test sets, better grounding in verified vulnerability metadata, a separate classifier stage to strengthen detection alongside the existing static-analysis tools, and models with larger context windows.

VIII. ACKNOWLEDGEMENT

Thanks are due to the Department of Computer Science and Engineering, NITTTR Chennai, for the resources and working environment that made this project possible, and to the project supervisor for guidance throughout its development.

REFERENCES

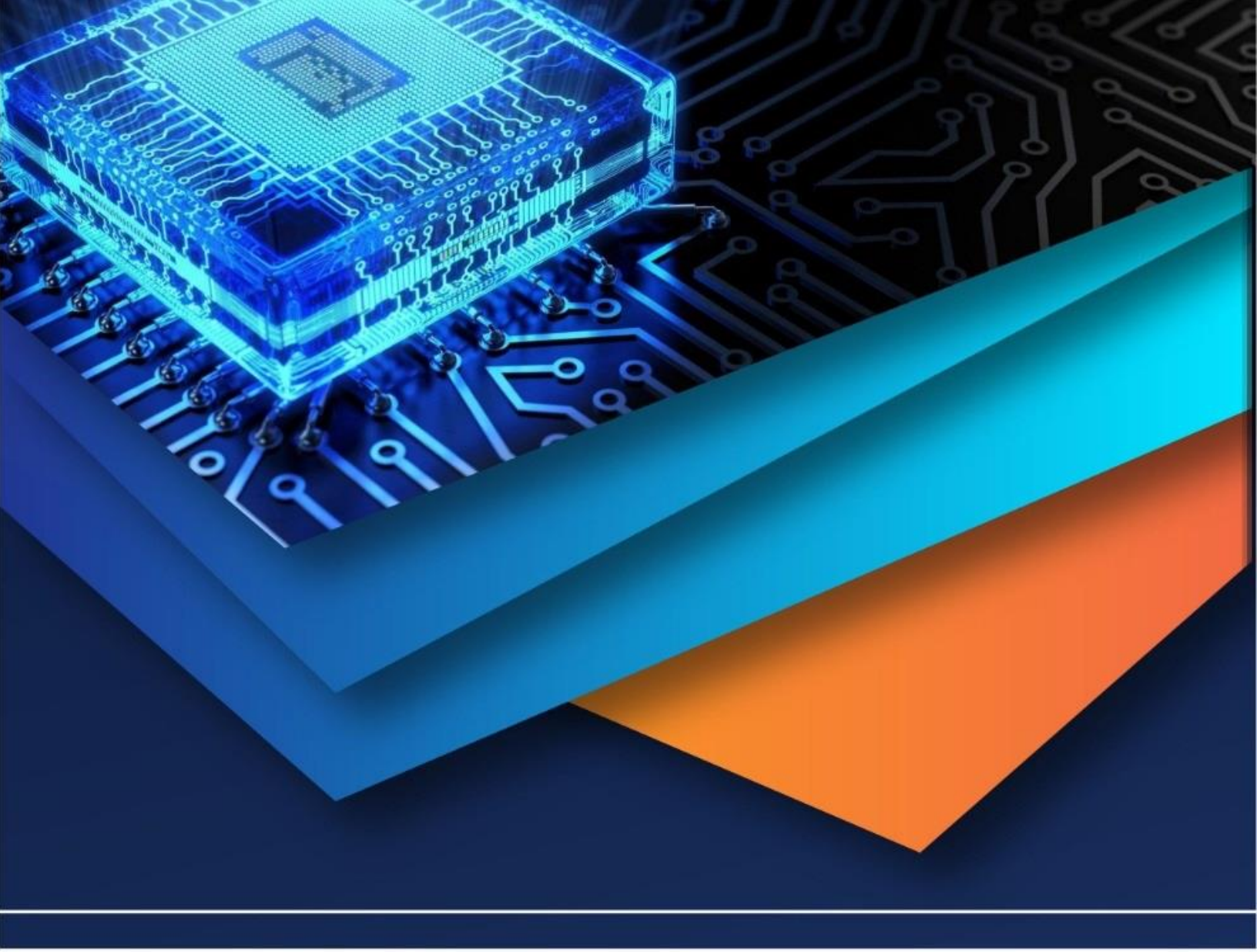
[1] X. Tang, K. Kim, Y. Song, C. Lothritz, B. Li, S. Ezzini, H. Tian, J. Klein, and T. F. Bissyandé, "CodeAgent: Autonomous communicative agents for code review," in Proc. 2024 Conf. Empirical Methods in Natural Language Processing (EMNLP), Miami, FL, USA, 2024, pp. 11279-11313.

[2] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Examining zero-shot vulnerability repair with large language models," in Proc. 44th IEEE Symp. Security and Privacy (SP), 2023, pp. 2339-2356, doi: 10.1109/SP46215.2023.10179420.

[3] M. Naïr, K. Yamani, L. Said Lhadj, and R. Baghdadi, "Curriculum learning for small code language models," in Proc. 62nd Annu. Meeting Assoc. Comput. Linguistics (ACL), Student Research Workshop, Bangkok, Thailand, 2024, pp. 390-401, doi: 10.18653/v1/2024.acl-srw.44.

[4] P. Buzzega, M. Boschini, A. Porrello, D. Abati, and S. Calderara, "Dark experience for general continual learning: A strong, simple baseline," in Adv. Neural Inf. Process. Syst. (NeurIPS), vol. 33, 2020, pp. 15920-15930.

[5] S. Lipp, S. Banescu, and A. Pretschner, "An empirical study on the effectiveness of static C code analyzers for vulnerability detection," in Proc. 31st ACM SIGSOFT Int. Symp. Software Testing and Analysis (ISSTA '22), Virtual, South Korea, 2022, pp. 544-555, doi: 10.1145/3533767.3534380.



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY