



iJRASET

International Journal For Research in
Applied Science and Engineering Technology



INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 14 **Issue:** X **Month of publication:** October 2026

DOI: <https://doi.org/10.22214/ijraset.2026.85168>

www.ijraset.com

Call: ☎ 08813907089

E-mail ID: ijraset@gmail.com

SmartCode AI: Intelligent Code Refactoring and Automated Bug Repair Using Retrieval Augmented Generation Architecture

Mrs. M. Nithya¹, Mohanapriya A², Mahilmalathi V³

¹Assistant Professor, ^{2,3}Student, Department of Computer Science and Engineering, K.L.N. College of Engineering, Tamil Nadu, India

Abstract: Software maintenance, especially locating defects and repairing faulty logic, consumes a large share of development effort and remains error prone when performed manually. This paper presents SmartCode AI, an explainable bug localization and automated repair system that grounds a large language model in retrieved repair knowledge using retrieval augmented generation. The system parses submitted Python code with abstract syntax tree analysis to validate syntax, extract structure and detect code smells such as bare exception handlers, empty functions and mutable default arguments. A lightweight hashing based embedding, computed fully offline, queries a persistent ChromaDB vector store of verified repair patterns, and the retrieved patterns together with the structural facts form a grounded prompt for the Llama-3.3-70B model hosted on Groq. The model returns a bug localization report, a refactoring plan and a corrected program. Every patch is executed in an isolated subprocess with a five second timeout, so that runtime failures and infinite loops are captured before the developer sees the result, and accepted patches are written back to the vector store through a feedback loop. An interactive Streamlit dashboard exposes each stage of the pipeline. Experiments on common runtime defects show that the system localizes and repairs division by zero, unguarded indexing and unsafe dictionary access errors while keeping the host environment safe. By combining static analysis, semantic retrieval, generative repair and sandbox validation, the system reduces debugging effort and delivers transparent fixes that developers can review before adoption.

Keywords: abstract syntax tree, automated program repair, bug localization, retrieval augmented generation, sandbox validation.

I. INTRODUCTION

Locating defects, repairing faulty logic and refactoring poorly written code take up a major portion of the effort in modern software projects. Developers must trace runtime exceptions by hand, understand unfamiliar code and apply corrections without introducing new faults, which is slow and error prone. Automated program repair (APR) addresses this problem by localizing a fault and generating a candidate patch with little human intervention [1], [2].

Large language models (LLMs) trained on source code have recently made repair far more practical [3]–[5]. However, a purely generative tool behaves as a black box: it may return code that looks correct but fails at runtime, invent library functions that do not exist, or silently change program behaviour [6]. Many tools also return a rewritten file without any explanation of what was wrong or any proof that the new version executes, which limits the trust that developers place in them.

Retrieval augmented generation (RAG) reduces these weaknesses by supplying the model with relevant external knowledge at inference time instead of relying on its parameters alone [7], [8]. For code repair, this means that proven fix patterns for errors such as division by zero, invalid indexing and missing dictionary keys are presented to the model together with the faulty program. A second requirement is safe verification, because running unverified generated code directly on a developer machine can exhaust resources or never terminate.

This paper presents SmartCode AI, a system that integrates abstract syntax tree (AST) analysis, vector based semantic retrieval, LLM driven repair and sandbox execution in one pipeline. The main contributions are as follows. First, the system explains where a defect is located before it proposes a fix, and shows the retrieved evidence behind each correction. Second, it uses an offline hashing based embedding with a persistent ChromaDB knowledge base that grows through a feedback loop of accepted patches. Third, it executes every generated patch in an isolated subprocess with a strict timeout and reports the real execution log. The rest of the paper reviews related work, describes the method, discusses the results and concludes with future directions.

II. RELATED WORK

Search based repair was popularized by GenProg, which evolves patches by mutating program statements and uses an existing test suite as the fitness function [2]. The approach repairs real defects without a formal specification, but the search is computationally expensive and depends on a strong test suite. Surveys of the field identify patch correctness as the central open problem and categorize repair techniques into fault localization, patch generation and patch validation [1], [9].

Learning based repair treats the task as translation from buggy to fixed code. SequenceR applies sequence to sequence learning with a copy mechanism to produce single line patches [10], and CoCoNuT combines context aware neural translation models in an ensemble [11]. Execution based feedback has also been used to improve neural repair [12]. These models, however, need a large curated training corpus and often generate patches in a narrow format.

Pre-trained code models changed this picture. Codex-style models, CodeT5 and CodeBERT show that general purpose models already encode substantial knowledge of programming languages [3], [13], [14]. Xia and Zhang demonstrated that such models repair defects in a zero-shot setting and outperform several traditional tools [4], and later studies examined the impact of code language models on repair and their ability to resolve real GitHub issues [15], [5]. A limitation of this line of work is that generated patches are not automatically validated, so plausible but incorrect fixes can be reported as successful.

Retrieval augmented generation, introduced by Lewis *et al.* for knowledge intensive language tasks, grounds a generator in documents returned by a dense retriever and reduces hallucination [7]. Izacard and Grave showed the same benefit for open domain question answering [8]. These works target natural language rather than source code. The gap addressed in this paper is the combination of retrieval grounded generation with static analysis and mandatory execution based verification for source code repair.

III. MATERIALS AND METHODS

A. System Overview

SmartCode AI is a modular web application. A developer pastes a faulty Python snippet into the interface, and the application passes it through an analysis layer, a retrieval layer, a generation layer and an execution layer. Fig. 1 shows the architecture. Results of every layer are returned to the dashboard so that nothing in the pipeline is hidden from the user.

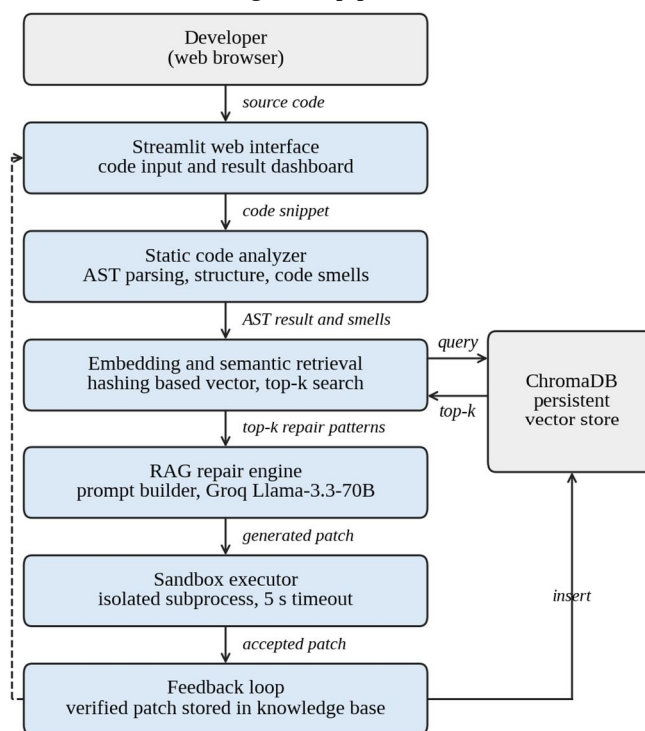


Fig. 1 System architecture of SmartCode AI

The workflow is summarized in Fig. 2. If parsing fails, the pipeline stops and reports the syntax error with its line number. Otherwise the code is embedded and the closest repair patterns are retrieved, a grounded prompt is built, the language model returns a localization plan with a corrected program, and the program is executed in the sandbox. The execution status decides whether the success output or the error log is displayed.

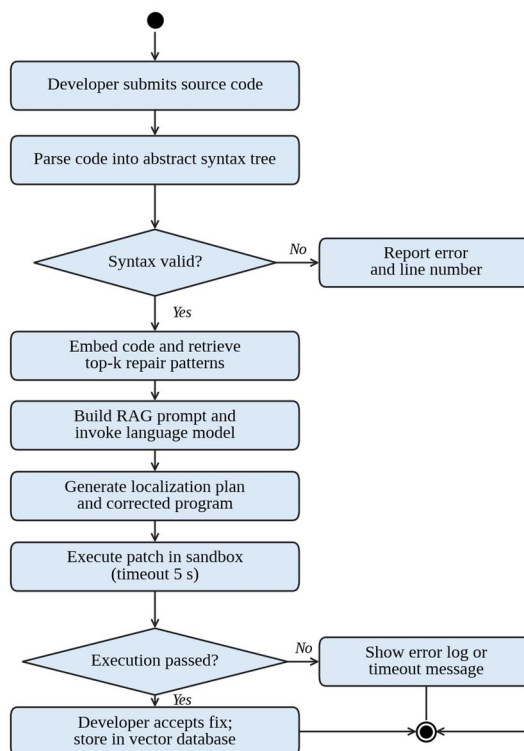


Fig. 2 Repair workflow with decision points

B. Static Code Analysis

The submitted code is parsed with the AST module of the Python standard library. A successful parse validates the syntax and allows the analyser to walk every node of the tree and list the defined functions and classes. A failed parse returns the error message with the exact line number. The same traversal detects three code smells: functions whose body contains only a pass statement, functions that use a mutable list, dictionary or set as a default argument, and bare except blocks written without an exception type.

C. Semantic Retrieval

A persistent ChromaDB collection stores verified repair patterns for frequent runtime faults. On first launch it is seeded with patterns for index errors, key errors, division by zero and mutable default arguments. Each document and each query is converted to a fixed length vector by a hashing based embedding that needs no model download. Let T be the sequence of lower case words of a text, $h(w)$ the MD5 hash of word w interpreted as an integer, and d the vector dimension, which is 128 in this work. The count in bucket i is given by (1), and the normalised vector by (2).

$$c_i = |\{w \in T: h(w) \bmod d = i\}| \quad (1)$$

$$v = \frac{c}{\|c\|_2} \quad (2)$$

Normalisation makes the similarity comparison depend on the direction of the vector and not on the length of the submitted code. The top- k nearest documents ($k = 2$ in the default configuration) are returned and displayed on the dashboard, so the developer can see which knowledge influenced the repair.

D. RAG Based Bug Localization and Patch Generation

The code, the structural analysis record and the retrieved patterns are assembled into a single prompt that defines the role of the model as an automated code repair system. The prompt requires the explanation and the corrected program to be separated by a fixed delimiter, which lets the application divide the response into a human readable bug localization report with a refactoring plan and an executable section. Markdown fences are removed so that the stored patch is plain Python source.

The hosted Llama-3.3-70B model is called through the Groq inference API with a low temperature of 0.1 so that repairs remain conservative. A fallback list of candidate models is tried in order, and if no API key is supplied or the service is unreachable, a built in rule based explanation is returned so that the workflow can still be demonstrated.

E. Sandbox Validation and Feedback Loop

No patch is presented as correct until it has been executed. The generated code is written to a temporary file and run as an independent subprocess using the current Python interpreter, with a five second timeout. A zero exit status is reported as passed together with the captured output, a non-zero status displays the standard error stream as the failure log, and an expired timeout reports a suspected infinite loop and terminates the process. The temporary file is deleted in every branch.

When the developer accepts a verified fix, the patch and a short summary are embedded with the same function and inserted into the collection under a new identifier with a feedback category tag. Later retrievals can therefore return project specific fix patterns.

F. Implementation Environment

The system is implemented in Python 3.10 and developed in Visual Studio Code. Table I lists the components and their roles. Since model inference is performed through a hosted API, the local requirement is modest: a processor such as an Intel Core i5 or AMD Ryzen 5, 8 GB of RAM (16 GB preferred for larger knowledge bases), a 256 GB solid state drive and an internet connection.

TABLE I
SOFTWARE COMPONENTS AND THEIR ROLES

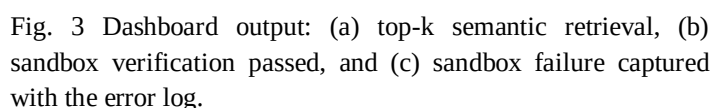
Component	Role
ast module	Syntax check, structure and code smell detection
hashlib, NumPy	Offline hashing based embedding
ChromaDB	Persistent vector knowledge base
Groq, Llama-3.3-70B	Localization report and patch
subprocess	Isolated patch execution
Streamlit	Interactive dashboard

IV. RESULTS AND DISCUSSION

A. Testing

- 1) *Module Testing*: Each module was first tested independently. The analyser was run on valid programs, on programs containing each supported code smell and on programs with deliberate syntax errors, and the reported function names, class names and error line numbers matched the inputs. The embedding function produced identical vectors for identical text, always returned the configured length and gave unit magnitude for non-empty input. The sandbox was exercised with a program that prints output, a program that raises an exception and a program with an infinite loop, and the passed status, the error log and the timeout message were produced correctly, with the temporary file removed each time.
- 2) *Functional, Integration and Performance Testing*: Functional and integration tests followed the complete workflow. The dashboard displayed the analysis summary, the retrieved patterns, the localization report, the patch and the execution result in order, and the model response was split correctly at the delimiter so that the corrected code reached the sandbox unchanged. Seeded documents persisted across restarts, and a patch inserted through the feedback loop was retrieved in a later session. In performance tests the parsing and embedding stages completed almost instantaneously, the query time stayed stable as feedback insertions increased the collection size, and the timeout stopped a non-terminating loop at the configured limit. Remote model inference was the largest share of the total pipeline time.

The default example in the dashboard is a function that divides by the length of a list and therefore raises a division by zero error for an empty list. For this input the retrieval stage returns the guidance to verify bounds and to validate the denominator before division, as shown in Fig. 3(a), which confirms that the embedding and similarity search operate correctly on code text. The localization report names the division by the length of the sequence as the failing statement and the refactoring plan describes a guard before the division.

TABLE II
SYSTEM RESPONSE TO EACH CLASS OF INPUT

Input Condition	System Response
Division by zero on an empty list	Guard pattern retrieved; patch executed and passed
Unguarded list indexing	Bounds-check pattern retrieved; patch generated
Unsafe dictionary access	Default value access recommended; patch verified
Syntax error	Error with line number; pipeline halted
Non-terminating loop	Stopped at 5 s; infinite loop reported
Model service unavailable	Rule based fallback used

C. Discussion

Compared with linters and editor assistants, the proposed system returns more than a rewritten file. It reports which statement failed, which retrieved pattern influenced the correction and whether the repaired program executed, so the developer keeps full control of the final decision. Because every stage is shown separately, syntax errors, an unreachable model endpoint, runtime exceptions in a patch and timeouts each produce a distinct message and none is silently ignored.

Several limitations should be noted. Only Python source code is supported, and the five second budget is unsuitable for long running programs. The hashing based embedding captures lexical similarity rather than deep semantic meaning. The subprocess sandbox protects the application process and bounds execution time but is not a full container level isolation. In addition, the evaluation reported here is functional and qualitative; a quantitative comparison on a public repair benchmark remains to be done. Generated patches are suggestions that should be reviewed and tested before they are merged.

V. CONCLUSIONS

This paper presented SmartCode AI, a system that combines AST static analysis, vector based semantic retrieval, LLM driven repair and sandbox verification for explainable automated bug repair. The pipeline returns a structural analysis, retrieved repair patterns, a bug localization report, a corrected program and real execution evidence, and a feedback loop stores accepted patches to improve later retrievals. Testing showed that the system repairs common runtime defects and contains unsafe or non-terminating programs without affecting the host application. The use of Python, ChromaDB, a hosted Llama-3.3-70B model and Streamlit keeps the setup cost low and the design modular.

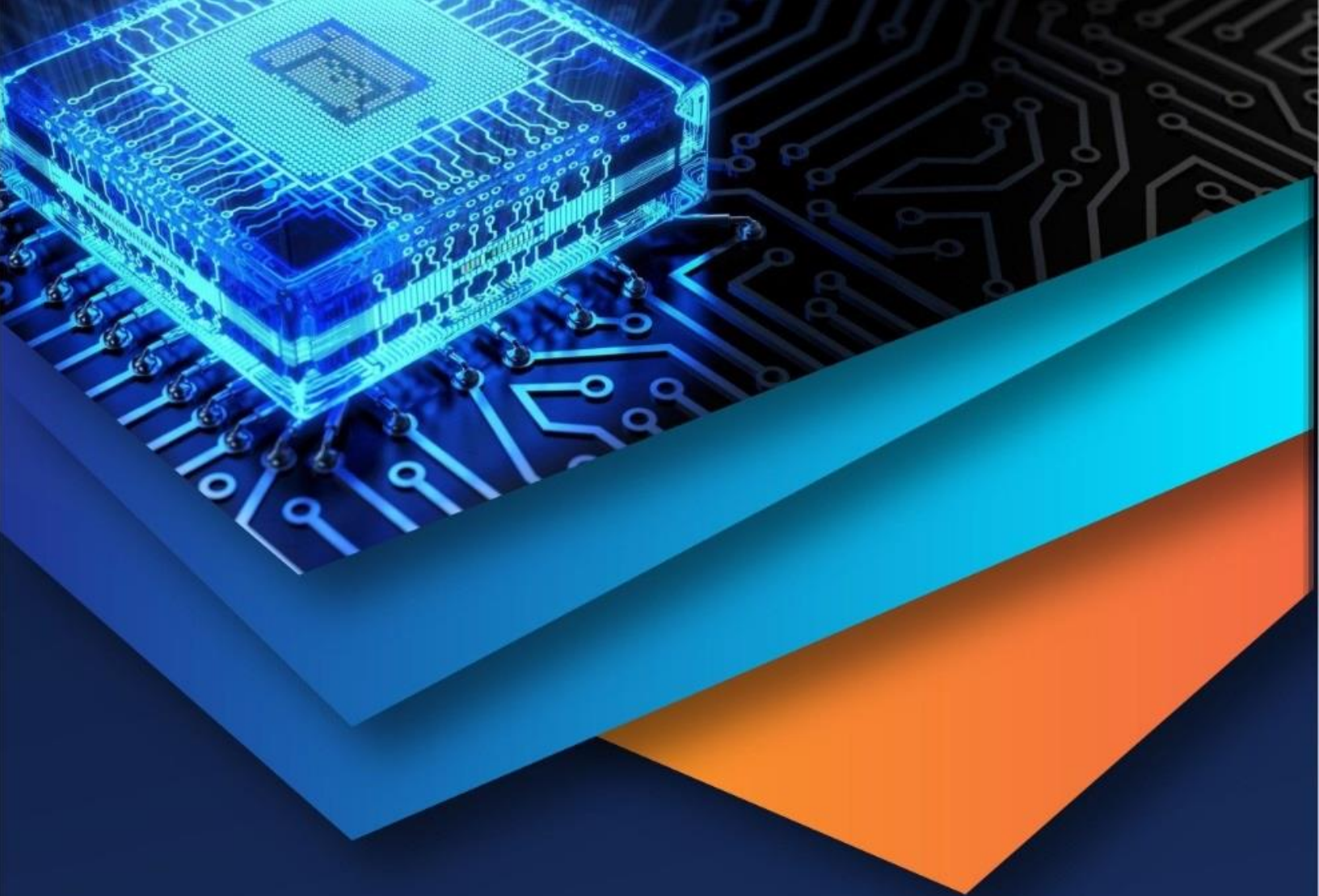
Future work includes support for several programming languages through a language agnostic parser, learned code embeddings such as CodeBERT [14] to retrieve by meaning rather than by lexical overlap, automatic test generation to confirm that the original behaviour is preserved, container based isolation for larger programs, integration with version control platforms to open pull requests, an editor extension, and ranking of several candidate patches by validation confidence.

VI. ACKNOWLEDGMENT

The authors thank the Head of the Department of Computer Science and Engineering and the mini project coordinator of K.L.N. College of Engineering for the facilities and encouragement provided during this work. The authors received no specific funding for this research and declare that there is no conflict of interest regarding the publication of this paper.

REFERENCES

- [1] M. Monperrus, "Automatic software repair: A bibliography," *ACM Comput. Surv.*, vol. 51, no. 1, pp. 1–24, 2018.
- [2] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "GenProg: A generic method for automatic software repair," *IEEE Trans. Softw. Eng.*, vol. 38, no. 1, pp. 54–72, 2012.
- [3] M. Chen et al., "Evaluating large language models trained on code," *arXiv:2107.03374*, 2021.
- [4] C. S. Xia and L. Zhang, "Less training, more repairing please: Revisiting automated program repair via zero-shot learning," in *Proc. ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE)*, 2022.
- [5] C. E. Jimenez et al., "SWE-bench: Can language models resolve real-world GitHub issues?" in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2024.
- [6] A. Fan et al., "Large language models for software engineering: Survey and open problems," in *Proc. IEEE/ACM Int. Conf. Softw. Eng.: Future Softw. Eng.*, 2023, pp. 31–53.
- [7] P. Lewis et al., "Retrieval-augmented generation for knowledge-intensive NLP tasks," in *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, vol. 33, 2020, pp. 9459–9474.
- [8] G. Izacard and E. Grave, "Leveraging passage retrieval with generative models for open domain question answering," in *Proc. Conf. Eur. Chapter Assoc. Comput. Linguistics (EACL)*, 2021, pp. 874–880.
- [9] Q. Zhang, C. Fang, Y. Ma, W. Sun, and Z. Chen, "A survey of learning-based automated program repair," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 2, 2024.
- [10] Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyanyk, and M. Monperrus, "SequenceR: Sequence-to-sequence learning for end-to-end program repair," *IEEE Trans. Softw. Eng.*, vol. 47, no. 9, 2021.
- [11] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "CoCoNuT: Combining context-aware neural translation models using ensemble for program repair," in *Proc. ACM SIGSOFT Int. Symp. Softw. Test. Anal. (ISSTA)*, 2020, pp. 101–114.
- [12] H. Ye, M. Martinez, and M. Monperrus, "Neural program repair with execution-based backpropagation," in *Proc. Int. Conf. Softw. Eng. (ICSE)*, 2022.
- [13] Y. Wang, W. Wang, S. Joty, and S. C. H. Hoi, "CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proc. Conf. Empirical Methods Natural Lang. Process. (EMNLP)*, 2021, pp. 8696–8708.
- [14] Z. Feng et al., "CodeBERT: A pre-trained model for programming and natural languages," in *Findings Assoc. Comput. Linguistics: EMNLP*, 2020, pp. 1536–1547.
- [15] N. Jiang, K. Liu, T. Lutellier, and L. Tan, "Impact of code language models on automated program repair," in *Proc. Int. Conf. Softw. Eng. (ICSE)*, 2023.



10.22214/IJRASET



45.98



IMPACT FACTOR:
7.129



IMPACT FACTOR:
7.429



INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Call : 08813907089  (24*7 Support on Whatsapp)