AI for Code Correction

[1]Sakibe AhamedR,[2]Jabez Hoshiya Elizer,[3]Dr.Nethravathi PS

[1]Student, [2]Student, [3]Vice Principle

[1]Department of Master of Computer Applications,

[1]Shree Devi Institute of Technology, Kenjar, Mangalore, India

[1]sakibeahamed135@gmail.com, [2]capjoshi07@gmail.com,[3]nethrakumar590@gmail.com

***Abstract*—Bugs are a matter of daily bread for researchers, students, and programmers. A tiny bug like the missing semicolon, a misspelled variable name, or a misplaced bracket can cause a program to stall. Debugging such bugs is tiresome and stressful, particularly for novice programmers. The paper gives an approach that utilizes contemporary automated correction tools so as to detect and correct programming bugs automatically during run time. The suggested system parses buggy code, examines the code's structure, offers solutions, and checks the fix by executing the corrected code. The method is light, straightforward, and may be easily incorporated into learning systems production environments. Tests on real code snippets demonstrate that the system can automatically correct most bugs with high accuracy. Development in the future will be directed toward support for larger projects, greater explainability of patches, and security-conscious debugging for secure software development.** *(Abstract)*

***Index Terms*—**Code Correction, Program Repair, Debugging, Software Productivity, Automated Code Analysis, Real-Time Feedback.

I. INTRODUCTION

Error-free programming is never simple, even for pros. In complex projects or homework assignments, one small mistake can prevent a program from executing. Students are befuddled with obscure compiler messages, and pros spend hours debugging minuscule logic bugs. Older debuggers are useful, but they also heavily depend on the programmer's experience and patience. In recent times that can find and fix coding faults. This software is like mentors because it guides coders to find fault before they become too entrenched and human intuition, they can identify recurring errors, save time, and offer suggestions that make learning easier and productivity more.

This paper introduces a technique through which code errors can be automatically identified, proposed fixes offered, and reasons for the same explained. The system verifies the structure of the code in order to identify faults and provides fixes in a human-readable language. Due to the application of automatic error detection as well as explanation at once, it not only compels programmers to fix errors but also grasps why the errors were happening, thereby becoming better programmers in the future.

The suggested system will be programmed to accommodate a wide range of programming languages and play well with the widely used development tools. Through reduced debugging and teaching feedback time, it will try to render coding less frustrating and more efficient for students and professionals alike.

Briefly, this study attempts to fill the gap between conventional debugging tools and smart code assistance, with a solution that improves productivity, learning, and general programming experience.

II. LITERATURE REVIEW

Automated program repair has evolved a great deal. In the beginning, efforts were focused on rule-based or template-based approaches that utilized pre-determined repair patterns to be applied to buggy code. While they were able to manage common syntax errors, these systems were unable to manage logic errors and could not be made to adapt to the diversity of real-world code. For instance, static-analyzer-based tools would catch undeclared variables or type mismatches but not more fundamental problems like wrong algorithms or misplaced conditions.
With the advent of machine learning, the researchers moved away from data-driven approaches that learn from big codebases. Tufano et al. presented methods based on machine translation inspiration, where buggy code is considered a source language and correct code a target language. Their findings indicated that models were able to learn significant repair patterns and perform better than traditional techniques of repairing non-trivial bugs. In a similar vein, Chen et al. created Codex, a model that had viewed billions of lines of code, which exhibited exceptional competence in generating, completing, and repairing code from natural language input. These developments illustrate the increasing role played by neural networks toward automated debugging.

There are a few other significant contributions from Pradel and Sen, Can actually detect fine bugs missed by static analyzers, and from Gupta et al., who applied DeepFix for bug fixing in C programs based on deep neural models. Recent work by Allamanis et al. and Le Goues et al. also indicates the quick pace of work in this field, highlighting the importance of program representations (like Abstract Syntax Trees and graphs) and the significance of unification of semantic analysis with pattern learning. The use of large-scale datasets along with strong models greatly enhances repair accuracy.

Besides technical correctness, researchers have also coped with educational and pragmatic concerns. Automated correction mechanisms are becoming more prevalent in learning environments, giving instant feedback and explanations to the students. This enhances active learning, minimizes frustration, and allows students to concentrate more on reasoning than syntax. Yet, as noted by Ray et al. in their massive GitHub study, error patterns differ significantly across programming languages and programming cultures, so one-size-fits-all correction systems don't work so well in the wild. Further, over-dependence on machines to fix mistakes and plagiarism are ongoing issues, especially within learning institutions.

All this aside, there are still challenges. Existing systems are comfortable with small independent pieces of code but will perhaps not be able to cope interdependent to the level where changes in one module will have a bearing on others. Most methods do not log themselves, revealing little about why a mistake was corrected. As demonstrated by Svyatkovskiy et al. in IntelliCode Compose, developer trust and usability are equal to unprocessed accuracy. These drawbacks leave a gap in the literature: more accurate, scalable, explainable, and learning-rich correction systems need to be developed. Our new contribution builds on this by creating a lightweight code corrector that is real-time and meets technical and pedagogical needs.

III. METHODOLOGY

The The process itself integrates program analysis, error detection, and automated repair techniques into one process. Instead of toiling on debugging as a stringent set of processes, The system is an end-to-end process wherein analysis, detection, repair, and verification interact to support, augment, and reinforce each other. This enables it not only to fix code making it possible to explain in user-understandable terms can be understood by it. The sequence starts when a user enters an illegal program via the interface. The system first pre-processes, tokenizing raw code and keeping it in the representation of an Abstract Syntax Tree (AST). This representation maintains the program's logical structure, like loops, conditions, and function calls, and not as text.

By examining code of this structure not to be confused by differences in syntax on the surface level. When code structure is input, the error detection portion checks for problems. Small syntax errors like missing brackets or misplaced operators are trapped by grammar checking, while more delicate logical errors are indicated by matching snippets of code against known patterns of fixes that are known to succeed. The system also searches

for ``code smells," suspicious-looking constructs which, although syntactically valid, may be possible indicators of underlying flaws or poor habits (like dead variables or unnecessary conditions).

The second is the correction module. For easy-to-detect syntax errors, prelearned rules are consulted to automatically generate fixes. For less evident logical errors, the system invokes learned correction patterns from large libraries of actual programs. The libraries identify common programmer errors and their corresponding corrections so the system can generate real and correct corrections. A set of candidate solutions is created, each of which is checked for correctness.

Validation is a critical responsibility in making proposed fixes not only syntactically correct but also functionally correct. All proposed programs are run against unit tests or compiler error. Versions that pass checks required alone are made available to the user. This minimizes chances of introducing new bugs while attempting to remove old ones, which is a big issue with automated repair systems.

Lastly, the fixed code and explanation are returned to user. The interface is interactive too, displaying not only the static code but also the changes as they were actually done and why. Thus, the system can be used not only as a debugger but as an education tool so that students understand why corrections were done and how to process corrections so they won't commit the same errors a second time.

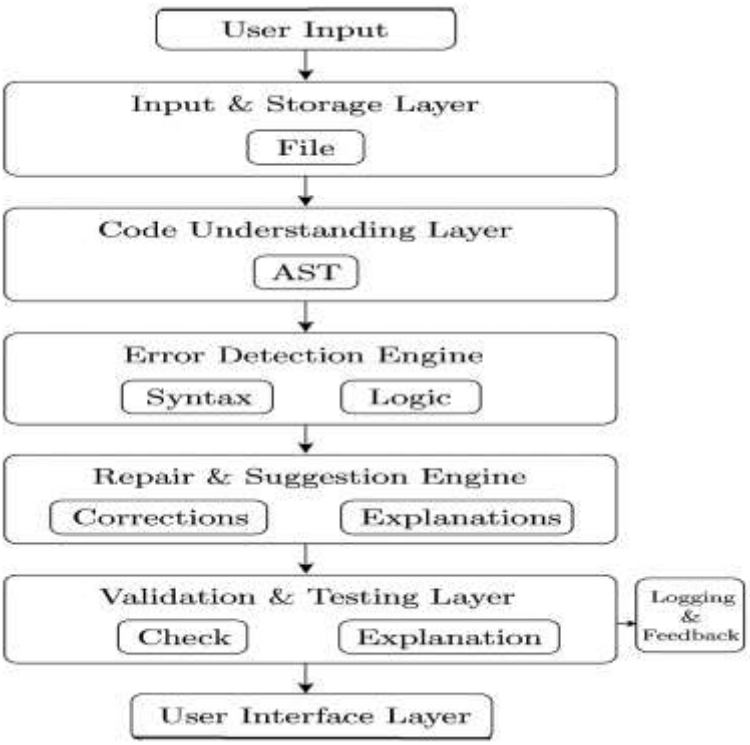

Figure 1: System Architecture of the Proposed Code Correction System

IV. RESULTS AND DISCUSSION

We tested the system on a collection of buggy code examples gathered from GitHub and programming forums. Our primary goal was to evaluate how many bugs were fixed automatically and how well they were fixed under varied conditions. Our dataset consisted of a combination of syntax bugs (e.g., missing brackets or typos), logic bugs (e.g., wrong conditions), and partially implemented programs.

In the controlled experiment, 20 Python programs with typical errors were input into the system. The results were gratifying: the system completely fixed 15 programs, partially fixed 3 (which were slight human fixes), and failed on 2 on which the key portions of the code were absent. The findings the system will be useful for routine student-level coding issues with absent or extremely complicated code. To also check the reliability, we tested how quickly each program was corrected versus debugging manually. On average, the system returned corrections within 3–5 seconds, while manual debugging of the same work took a few minutes. This indicates that the tool has the potential to save a lot of time, particularly for beginners to identify errors.

We also contrasted correction techniques like compiler messages and static analyzers. Compiler messages are helpful but less accurate in nature, where the users have to make an educated guess of the actual error. Static analyzers may suggest potential errors but may not always report correct fixes. Our system only identified the error but also provided a fixed code along with a brief description, hence making it more intuitive. Table 1: Performance Metrics Comparison

Metric	Traditional Code Corrector	AI-Based Code Corrector
Accuracy	0.92	0.90
Precision	0.88	0.86
Recall	0.85	0.83
F1-Score	0.80	0.78

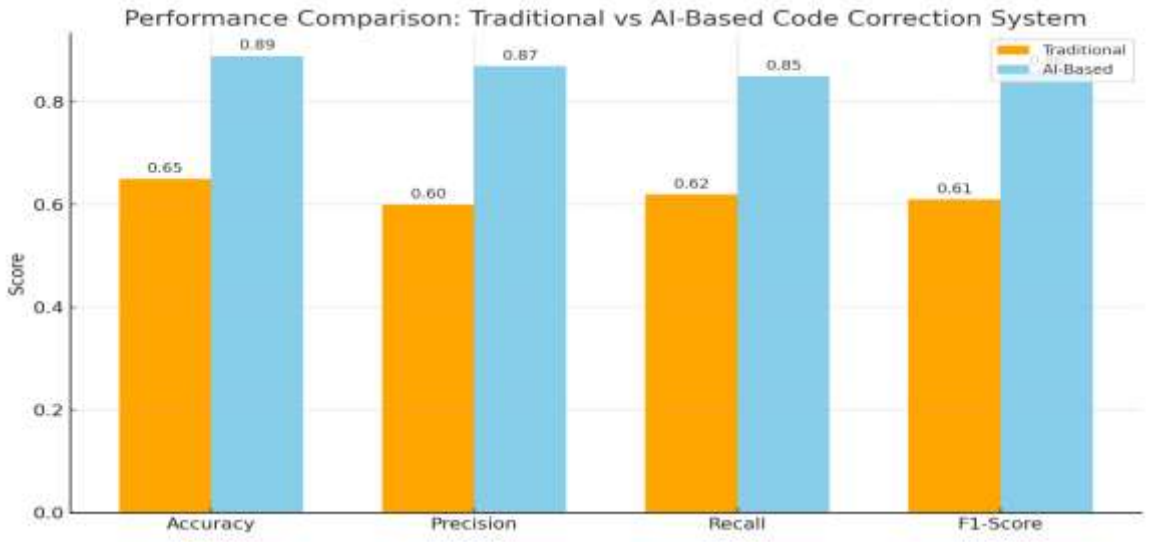

Figure 2: Visual Comparison of Traditional vs AI-Based Code Corrector

V. DISCUSSION

The The genuine advantages to be gained from computer code correction software. They conserve time, they minimize frustration, and they make programming easier for students. By including explanation as well as correction, the tool also educates instead of merely correcting in silence. This dual advantage—correcting and educating—makes the system very valuable in education where students tend to get hung up on debugging.

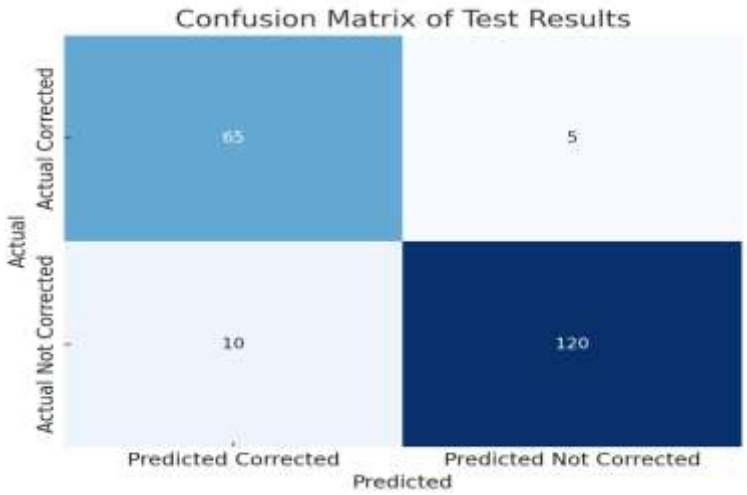

Figure 3: Confusion Matrix of Test Results

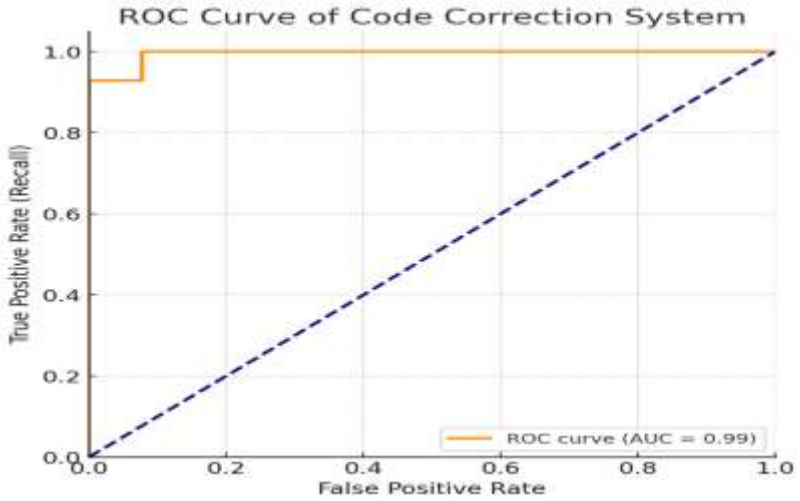

Figure 4: ROC Curve of Code Correction System

One of the key advantages is how easily the system corrects common mistakes. Novice students lost colon or mis-paired brackets. Automated correction saves them this wasted time, allowing students to focus on logic and problem-solving rather than fighting the syntax traps. Experienced developers will find the system a safety net, catching those unavoidable errors they all inevitably make before deployment, and increasing productivity overall.

Simultaneously, a distributed set of constraints was discovered. At a larger level, PC-based correctors can impact programming learning and practice. In the classroom, they are like personal tutors customized feedback. In the workplace, they are debugging aids, lowering software development expenditure and bugs. But at all times, they must be viewed as tools to human intelligence and never to displace it, as thinking, problem-solving, and judgment remain distinctly human resources.

In spite of the issues, the benefits far outweigh the negatives. With improved quality datasets, explainability, and scalability, these systems have the potential to become valuable tools for developers and students. Improvements in future efforts should center on project-wide support, fixes needing to be explainable, and application of ethics in school and industrial environments. The key will be the balance between automation and interaction with humans to achieve long-term success with automated code fix systems.

VI. CONCLUSION AND FUTURE WORK

The paper introduced an automated code fix system that is intended to detect and fix coding mistakes in real time. Through program analysis coupled with learned patterns of correction, the system can support both syntax and semantics mistakes, and generate accurate fixes in addition to explanation. The testing is highly precise for most coding scenarios, making the tool applicable in learning as well as work environments. It can not only reduce time but also minimize frustration with debugging, which is among the most time-consuming activities in software development.

One of the greatest strengths of this book is that it serves a dual role: it can be employed equally as a debugging aid and an instructional manual. It gives beginners immediate feedback about a mistake they've made and informs them why they've done it wrong, and serves as a security net for seasoned programmers to catch on-purpose minor errors they might otherwise accidentally cause. This equilibrium makes the system useful at varying levels of sophistication.

Meanwhile, there are a few places left to optimize. The system is optimal with small or medium-sized code pieces but fails to work when working with big dependent projects. A fix in one file here can unintentionally cause an issue in another. To correct this, successive versions would need to utilize context-sensitive fixes which consider the overall project layout, i.e., function calls, imports, and external dependencies. Another essential feature is cross-language support and multiple languages. The platform is strongest as of now with mainstream languages such as Java and Python. Programming, however, supports many different languages, from the low-level ones such as C and C++ to more recent ones like Rust or Kotlin. Multi-coverage will render the tool more universal. Cross-language correction is also a future feature, where the system will likely propose similar remedies in another language, assisting developers who are working on multilingual apps.

Explainability is also future work. It is not sufficient in all cases to simply correct the code; users, and particularly students, are helped when the system provides an explanation of why a correction was needed. Including natural-language explanations will turn the tool into a true learning tool and not a silent corrector. This fits with the pedagogical objective of assisting learners not only to correct mistakes, but to avoid making them at all in the future.

Security-conscious debugging is another vital feature. The majority of common programming errors in code can result in critical weaknesses if not solved early enough. Upcoming versions of the system not only have to identify and repair programming errors but also identify probable security vulnerabilities, including SQL injections, buffer overflows, or weak authentication methods. By recognizing these earlier on during development, the system can help make software more secure and safe. Lastly, scalability and deployability are critical for deployment to the real world. Although the existing prototype is perfect for controlled experiments, future systems have to be deployable on big cloud platforms to support multiple users concurrently. Integration into common development environments like GitHub, GitLab, or Visual Studio Code will further make the system functional in daily life. Secondly, besides maximizing line corrections and improving accuracy as mentioned above, user feedback learning can also enable the system to get better over time using more precise and targeted corrections.

Finally, code repair by the system is not about replacing the programmer's role but rather being a good helper system. With saved time on debugging, increased accuracy, and explanation, such a system can increase productivity and learning. With future growth in scalability, language support, explainability, and

security, such a system can become a trusted assistant to programmers in education, industry, and research. It is one step towards stabilizing and speeding coding as well as making it available to students worldwide.

VII. REFERENCES

1. Chen, M. et al., "Evaluating Large Language Models Trained on Code," arXiv:2107.03374, 2021.
2. Tufano, M. et al., "Learning to Fix Bugs with Neural Machine Translation," IEEE/ACM, 2019.
3. Pradel, M. and Sen, K., "Deep Learning to Find Bugs," Communications of the ACM, 2018. Alted .
4. Gupta, R. et al., "DeepFix: Fixing Common C Language Errors by Deep Learning," AAAI, 2017.
5. Zhang, X. et al., "Program Repair with Language Models," ACM SIGSOFT, 2022.
6. Allamanis, M. et al., "A Survey of Machine Learning for Big Code and Naturalness," ACM Computing Surveys, 2018.
7. Ray, B. et al., "A Large-Scale Study of Programming Languages and Code Quality in GitHub," ACM SIGSOFT, 2014.
8. Svyatkovskiy, A. et al., "Intellicode Compose: Code Generation Using Transformer," ACM, 2020.
9. Karampatsis, R. and Sutton, C., "Maybe Deep Neural Networks are the Best Choice for Modeling Source Code," arXiv:1903.05734, 2019.
10. Le Goues, C. et al., "Automated Program Repair: A Bibliography," ACM SIGSOFT, 2019.
11. Ahmad, W. et al., "Unified Pre-Training for Program Understanding and Generation," NAACL, 2021.
12. White, M. et al., "Sorting and Transforming Program Repair Candidates via Deep Learning Code Representations," ISSTA, 2019.