

International Journal Research Publication Analysis

Page: 01-14

SRI SHAKTHI INSTITUTE OF ENGINEERING AND TECHNOLOGY

AI BUG DETECTOR AND FIXER

***Anbarasu A., Dhakshida V., Elakiya J., Gowtham G., Guide: P. Kokila**

Department of Artificial Intelligence and Data Science, Sri Shakthi Institute of Engineering and Technology, Tamil Nadu, India.

Article Received: 18 September 2026

Article Revised: 08 October 2026

Published on: 28 October 2026

***Corresponding Author: Anbarasu A.**

Department of Artificial Intelligence and Data Science, Sri Shakthi Institute of Engineering and Technology, Tamil Nadu, India.

DOI: <https://doi-doi.org/101555/ijrpa.3955>

ABSTRACT

Software bugs cost time, and not just the time it takes to write a fix; a good chunk of it goes into figuring out what actually went wrong in the first place. Most debugging is still done the traditional way: a programmer reads an error message, scans the code, guesses at the cause, changes something, and reruns the program, often several times before the real problem is found. This paper describes an AI Bug Detector and Fixer, a system built to take over the earlier, more mechanical parts of that process. Users submit their source code through a web interface, and the system analyzes it using static analysis techniques built around Python's Abstract Syntax Tree (AST). Syntax errors are caught during parsing, while separate modules check for undefined and unused variables. Once an issue is detected, the system tries to fix it: simple, well-understood problems are corrected using deterministic rules, and more complex ones are handed to an AI-assisted repair module that generates a candidate patch based on the error, the surrounding code, and a set of repair constraints. The backend is built with FastAPI, and the detection, explanation, patch generation, and validation stages are kept separate so each one can be tested and improved on its own. The tool is aimed at students and beginner programmers who need fast, understandable feedback while they code. This paper covers the motivation behind the project, related work, the design goals, the methodology, the implementation, how the system might be evaluated, its limitations, and where it could go next. The underlying goal throughout is to cut down on repetitive debugging effort without producing fixes that are hard to trust or verify.

KEYWORDS: Artificial Intelligence, Bug Detection, Automated Program Repair, Static Analysis, Abstract Syntax Tree, Source Code Analysis, Code Fixing, FastAPI.

I. INTRODUCTION

Software now touches almost every part of daily life: education, business, healthcare, communication, engineering, public services. As these systems grow larger, so does the chance of something going wrong. A defect can show up in many forms: a syntax error, a variable used incorrectly, the wrong control flow, bad data handling, a runtime crash, or a logic error that quietly produces the wrong result. Catching these problems early matters, because the later a defect is found, the more it usually costs to fix. This is especially true for students and beginner programmers, who often run into an error message that describes a symptom without saying anything about the cause behind it.

Debugging has traditionally been a very human process. A developer reads whatever the compiler or interpreter prints out, hunts through the code for the relevant section, forms a guess about what's wrong, makes a change, and runs the program again, sometimes many times over, before landing on the right fix. Static analysis tools take some of this burden off the developer by examining code without running it, catching syntax violations, questionable patterns, and some categories of semantic problems. But most of these tools stop at the warning. It's still up to the developer to interpret it and figure out what to do next.

Automated program repair goes a step further and tries to generate a fix outright. Researchers have explored this from several angles: search-based, template-based, constraint-based, and more recently, learning-based methods. Gazzola, Micucci, and Mariani put together a wide-ranging survey of the field, laying out the major families of repair techniques and where each one tends to fall short [1]. Elixir showed that machine learning could be used to rank candidate patches, so a learned model could meaningfully guide the repair process instead of treating every candidate as equally likely [2]. Since then, work in the area has moved further into deep learning, code transformations, patch ranking, and neural program repair.

The rise of AI and code-focused language models has opened this up even further. An AI system doesn't have to rely only on fixed templates; it can draw on surrounding code, error descriptions, and patterns it has learned to suggest a fix that fits the context. That said, a patch shouldn't be trusted just because it looks reasonable. Any repair system needs a way to check its own work, whether through syntax checks, tests, static analysis, or some other correctness criteria. There is already good evidence that automated repair tools can produce patches that pass the available tests without solving the underlying problem at all [3].

This project, the AI Bug Detector and Fixer, was designed with that concern in mind. Detection and repair are not treated as one black-box AI step. Instead, the system is broken into stages. First, it parses and inspects the source code. Second, it classifies and explains whatever it finds. Third, it decides how to fix it: straightforward, well-defined bugs get a deterministic fix, while trickier ones go to an AI-assisted patch generator. Finally, before any correction is shown to the user, it gets validated. Splitting the process up this way makes the system more transparent, and honestly, much easier to test and debug in its own right.

The current build focuses on Python, mainly because Python's AST tooling gives such a clean, structured view of the code: assignments, identifiers, function definitions, loops, conditionals, all of it. Syntax errors surface naturally during parsing, and variable-related issues can be found by comparing where names are defined against where they get used. Down the line, the same architecture should be able to support other languages through language-specific parsers.

Accessibility is a big part of why this project exists. Students hit errors constantly while learning to program, and compiler messages or static-analysis warnings are not always easy to parse. A debugging assistant built for learning should explain what went wrong, where, why it matters, and what the fix changes, instead of just dumping a raw diagnostic on the screen.

There is also an important distinction the project tries to preserve: a suggested fix is not the same thing as a verified one. A suggestion is just a candidate, something produced from a known error pattern or an AI model. A verified fix is a candidate that has passed the system's validation checks. Given how often automated-repair research has run into the problem of incorrect or overfitted patches, this distinction matters. It is why validation comes after patch generation in this workflow, not as an afterthought.

This paper's contributions, in short, are these:

First, a modular pipeline for identifying common programming defects in source code. Second, an explanation layer that turns technical diagnostics into something a student can understand without a second Google search. Third, a repair layer that handles safe cases deterministically and hands harder ones to an AI-assisted generator. Fourth, a validation stage that checks generated code before it's ever presented as a fix. And fifth, an extensible architecture that leaves room for future machine learning models, code datasets, and multi-language support — the FastAPI-based version described here is really just the starting point. The rest of the paper is organized as follows. Section II looks at existing systems and where they fall short. Section III lays out the design and objectives. Section IV reviews relevant

research from 2017 to 2024. Section V walks through the methodology. Section VI covers the software implementation. Section VII discusses evaluation and what results might look like. Section VIII wraps up with conclusions and future work.

II. EXISTING SYSTEMS

Existing debugging and code-quality tools generally fall into four groups: compiler diagnostics, static analyzers, automated program repair tools, and AI-based coding assistants. Compilers and interpreters do a solid job catching syntax and type errors, but they don't really offer a repair workflow; they just tell you something is wrong. Static analyzers catch a wider range of issues, but a warning is still just a warning; the developer has to do the interpreting. Automated program repair tools try to close that gap by generating actual patches, using techniques like mutation, search, templates, program synthesis, or learned transformations. These systems have shown that a meaningful subset of real-world bugs can be fixed automatically. But repair quality depends heavily on how well the fault is localized, how large the patch search space is, what tests are available, and whether the generated patch is correct in the first place. Learning-based methods add another dependency on top of that: the quality and diversity of the training data.

AI coding assistants, meanwhile, can explain errors and suggest changes using large language models, and they're genuinely good at contextual reasoning and natural-language interaction. Their weakness is that they can produce code that is syntactically fine but semantically wrong. If the repair request is not well constrained, they might end up touching parts of the program that had nothing to do with the bug. That's the argument for combining AI generation with deterministic analysis and validation instead of leaning on either one alone.

What sets this system apart from a basic error checker is that it is built around a full detect–explain–fix–validate loop. It doesn't claim to fix every bug automatically, since that would be overselling it. Instead, it reports how confident it is and whether a given fix has been validated, which felt like a more honest way to present automated repairs to someone who is going to rely on the output.

III. DESIGN AND OBJECTIVES

The architecture is made up of an input interface, an API layer, a source-code parser, a set of bug analyzers, an explanation engine, a repair engine, a validation engine, and a result presenter. We kept these pieces separate on purpose, so that any one of them could be improved later without having to touch the rest of the system.

Going in, we had six things we wanted the system to actually do. It needed to catch syntax errors, undefined variables, unused variables, and whatever other analyzable defects we could add as the analyzer set grew. Every issue it found had to come with a meaningful explanation — where it is, what category it falls into, what caused it, why it matters — because a raw error code doesn't help a beginner much. Fixes had to be safe: a deterministic rule only gets applied when it's tightly constrained enough to trust, nothing more speculative than that. For the harder cases, we wanted AI-assisted repair to step in and generate candidate patches using contextual reasoning. Every one of those candidates had to be reparsed and tested before it could ever be labeled "verified." And the whole thing had to stay extensible, so new languages, analyzers, repair strategies, and evaluation datasets could be bolted on later without a rewrite.

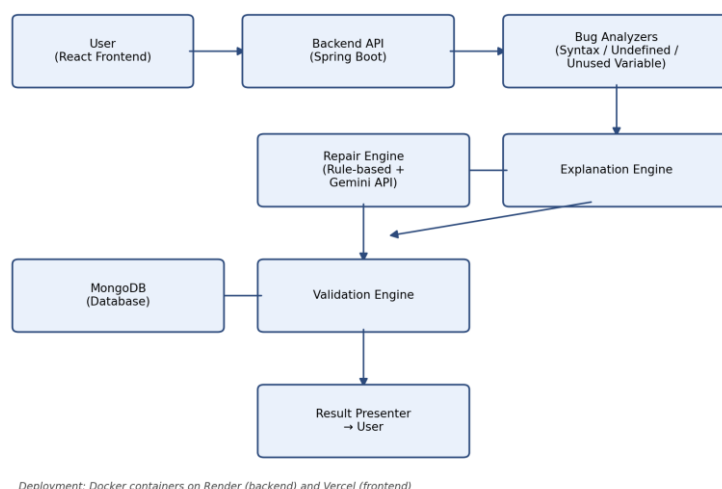


Fig. 1. Architecture of the AI Bug Detector and Fixer, showing the React frontend, Spring Boot backend API, bug analyzers, explanation engine, the repair engine (rule-based plus Gemini API), the MongoDB-backed validation engine, and the result presenter.

TABLE I. MAJOR SYSTEM COMPONENTS AND FUNCTIONS.

Component	Primary Function
Input Interface	Accept source code and user analysis requests.
Backend API (Spring Boot)	Receive requests and coordinate analysis modules.
AST Parser	Build a structured representation of source code.
Bug Analyzers	Detect syntax, undefined-variable, unused-variable, and future bug classes.
Explanation Engine	Convert diagnostics into understandable messages.
Repair Engine	Generate deterministic or AI-assisted candidate fixes.
Validation Engine	Reparsing, test, and compare candidate code before acceptance.
Result Presenter	Display detected issues, patch, confidence, and validation status.

The decision to keep detection and repair separate is deliberate. It stops a language model or repair heuristic from quietly rewriting code before the original bug has even been logged. Every repair candidate stays linked to the diagnostic that produced it, so the whole process can be traced back and audited.

IV. LITERATURE REVIEW

Automatic software repair has come a long way, from hand-written repair templates to search-based methods, and now to machine-learning and neural approaches. Gazzola et al. surveyed this landscape and broke down the major repair strategies, including generate-and-validate, synthesis, and fix recommendation, while also flagging how hard it still is to know whether a generated patch is correct [1].

Le et al. introduced ELIXIR, which uses machine learning to rank candidate patches instead of treating all of them as equally promising [2]. That idea, using a learned model to steer the search toward the more likely fixes, is part of what motivated the repair-ranking layer planned for this system.

Search-based repair has also leaned on mutation testing combined with fault localization. A 2018 COMPSAC paper used exactly this combination to cut down the cost of the repair search [4]. It reinforces a fairly intuitive idea: narrow down the suspicious region of code first, then search for a fix within that smaller space.

Marcilio et al. looked specifically at generating fix suggestions in response to static analysis warnings [5], which is directly relevant here, since a static warning on its own rarely comes with a usable correction attached. Their results show that automated fix suggestions add real value to static-analysis tools instead of just restating the warning differently.

Li et al. proposed DLFix, which frames repair as a learned code-transformation problem and leans heavily on surrounding source-code context [6]. That is a useful data point for this project too: use context around the bug, not just the error message itself, when generating a fix.

Separately, Li explored a deep-learning framework aimed at both detecting bugs and repairing them, using learned code representations [7]. The results suggest these representations help with both halves of the problem, spotting the defect and fixing it, compared to simpler baselines.

The rise of large language models opened up a new angle entirely. Fan et al. looked at whether program-repair techniques could clean up code that was originally generated by an

LLM [8]. That points to a useful pairing: let the language model handle flexible code editing, and let repair-and-validation techniques provide the guardrails.

FixGPT, presented at an IEEE conference, proposes a three-tier deep learning model for automated repair [9], and more recent work like T5APR shows pretrained transformer models being used for repair across multiple programming languages [10]. Both point to a natural next step for this project: eventually moving beyond rule-based Python repair toward a more multilingual, AI-assisted approach.

Patch correctness is its own problem worth flagging separately. Phung, Kim, and Lee proposed a deep-learning method for spotting incorrect patches by looking at the semantic relationship between a method's name and its body [11]. Their work is a good reminder that generating a patch is not the hard part; knowing whether it does what it's supposed to is.

Taken together, the literature shows a fairly clear progression: from static analysis and search-based repair, toward learning-based and now language-model-assisted repair. The open problems haven't gone away though: fault localization, patch correctness, dataset quality, generalization, computational cost. This project doesn't try to solve all of them at once. At a prototype level, it just pairs lightweight static analysis with constrained repair and explicit validation, and leaves the rest for later versions.

V. METHODOLOGY

We split the workflow into six stages: input acquisition, parsing, defect detection, explanation, patch generation, and validation. It mostly runs in that order, though a candidate patch can loop back to the repair stage if it fails validation — that loop turned out to be one of the more important design decisions, honestly, since it's what keeps a bad patch from ever reaching the user.

A. Input Acquisition

The user submits source code through the application interface. That request goes to the backend along with the selected programming language and analysis mode. We check that the input isn't empty and stays within the size limit before doing anything else with it. A future version could support project-level uploads, so the system could look at dependencies and multiple files at once instead of a single snippet — we just haven't gotten there yet.

B. Parsing and AST Construction

For Python code specifically, the source gets parsed into an Abstract Syntax Tree, which gives structured nodes for every statement and expression. If parsing fails outright, we record

the syntax error along with its location and a plain-language explanation. From there, a repair rule can attempt a safe fix for supported syntax patterns.

C. Bug Detection

Right now there are three separate analyzers. The syntax checker relies on parsing to catch invalid Python. The undefined-variable checker tracks names used before they're properly defined within the scope being analyzed. The unused-variable checker flags definitions that never get referenced afterward. We're planning to add more analyzers over time, for things like unreachable code, suspicious conditionals, exception handling, insecure functions, and complexity, without disrupting the rest of the pipeline.

D. Explanation

Every diagnostic gets stored as a structured object: bug type, location, message, severity, and a suggested action. The explanation engine turns that into a short, readable description. So instead of just surfacing a raw parser exception, it can say plainly that a closing parenthesis seems to be missing near a given line, and show the statement where it happened.

E. Fix Generation

The repair engine checks first whether the bug falls into a class it can fix deterministically. We lean toward deterministic rules wherever we can, simply because their behavior is predictable. When no safe rule applies, the AI-assisted repair module steps in and builds a candidate patch from the source code, the diagnostic, and the surrounding context. It's instructed to touch only the necessary region and leave the rest of the code alone — we learned early on that an unconstrained prompt tends to rewrite more than it should.

F. Validation

Every candidate patch gets parsed again first; if that fails, it's rejected outright. Where test cases exist, the repaired program runs against them. We also rerun static analysis to check whether the original issue is actually gone. Only patches that clear all of this get marked "verified." Anything generated but not validated stays labeled as a suggestion, not a confirmed fix.

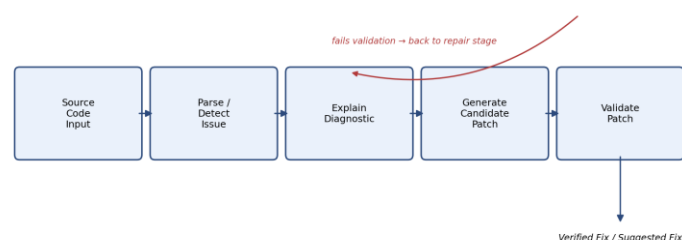


Fig. 2. Processing flow for detecting a bug, generating a candidate repair, validating the patch, and presenting the result. A patch that fails validation loops back to the repair stage rather than being shown to the user.

One useful way to think about choosing between candidate patches is to score each one, p , as:

$$\text{Score}(p) = w_1 \cdot V(p) + w_2 \cdot C(p) + w_3 \cdot S(p) - w_4 \cdot R(p)$$

where V is validation success, C is contextual consistency, S is simplicity, and R is risk. The weights here would need to be tuned experimentally — this formula is meant as a design principle to guide patch selection, not a finished, trained scoring model.

VI. SOFTWARE IMPLEMENTATION

Section V describes the design we originally sketched around a Python/FastAPI backend. For the working prototype, though, we ended up going with a different stack, mostly based on what we already knew and what was free to host. The backend runs on Java 17 with Spring Boot 3.3, which handles the REST endpoints for receiving source code and returning structured analysis results, and the frontend is a React 18 + Vite single-page application. We kept the project structure modular even after the switch, so each analyzer stays separate and independently testable, and the parsing/analysis logic is kept apart from the HTTP layer for maintainability.

Development Environment and Technology Stack

Here's what we actually used to build and deploy the prototype: the development tools, the accounts and cloud services we set up, and the stack running inside the project.

TABLE III. DEVELOPMENT TOOLS.

Tool	Purpose
VS Code	Code editor
Git (v2.52.0)	Version control
JDK 17 (Eclipse Temurin)	Java runtime / compiler
Apache Maven 3.9.16	Java build tool
Node.js + npm	JavaScript runtime / package manager (for React)
PowerShell	Windows terminal

TABLE IV. ACCOUNTS / CLOUD SERVICES CREATED.

Service	Purpose
GitHub	Code hosting (github.com/dhakshidav-png/ai-bug-detector)
Google AI Studio	Gemini API key
Render	Backend hosting (free tier)
Vercel	Frontend hosting (free tier)
MongoDB Atlas	Cloud database (currently mid-troubleshooting)

TABLE V. TECH STACK USED INSIDE THE PROJECT.

Layer	Technology
Frontend	React 18 + Vite + React Router
Backend	Java 17 + Spring Boot 3.3
Database	H2 → MySQL (abandoned) → MongoDB (current)
AI	Google Gemini API (gemini-flash-latest)
Deployment	Docker (for Render), environment variables

The database went through three stages before we landed on something that stuck. We started with the embedded H2 database, mainly for quick local testing. Then we moved to MySQL for a more production-like relational setup. That got abandoned too, once we needed a cloud-hosted, schema-flexible store that fit better with the JSON-shaped diagnostic and patch objects the system passes around, so we switched to MongoDB. MongoDB Atlas hosts the live database now, though the connection was still being troubleshooted as of this writing. For the AI-assisted repair module, we call the Google Gemini API (gemini-flash-latest) instead of hosting a model ourselves, which keeps the backend light enough to run comfortably on Render's free tier. The backend is containerized with Docker for deployment to Render, and configuration values like the Gemini API key and database connection string are supplied through environment variables instead of being hard-coded. The frontend is deployed separately on Vercel.

A typical request includes the source code plus some optional configuration. The backend runs the syntax analyzer first; if the code parses cleanly, the resulting AST is passed along to the variable-analysis modules. All the diagnostics get combined into one structured response, and the frontend shows the line number, issue category, explanation, and recommended action for each one.

The fixer module works on two levels. The first is rule-based repair, which is good for well-defined issues where the fix can be reliably inferred without any guesswork. The second is the AI-assisted repair interface, which gets a constrained prompt containing the relevant code region and diagnostic details, and returns a candidate patch. That candidate then goes straight to validation.

The validation engine is arguably the most important piece of the whole implementation, since it's what stops the system from treating an arbitrary model output as a confirmed fix. At minimum, a patch has to parse successfully; a stronger check runs it against unit tests or example cases. For an educational tool, it helps to make the confidence level visible, so the interface shows three states: Detected, Suggested Fix, and Verified Fix, letting the user know how much to trust what they're looking at.

The frontend is built with standard HTML, CSS, and JavaScript. A simple dashboard layout works well here: a code editor on one side, an analysis panel on the other. Detected errors are highlighted by line number, and a separate patch view shows the original code next to the proposed fix, along with a validation indicator showing whether it passed the available checks.

The modular design leaves room to grow. Language-specific parsers could eventually feed into a shared diagnostic format, and different repair engines (rule-based, retrieval-based, transformer-based, or LLM-assisted) could all plug into the same validation interface without needing separate infrastructure.

VII. RESULTS AND DISCUSSION

We haven't run a formal evaluation yet — Table II below is still empty on purpose. But it's worth laying out how we plan to measure this once the analyzers are stable enough to test properly. Evaluating the system means looking at both detection performance and repair quality. Detection can be tested against a labeled set of correct and defective programs, counting true positives, false positives, true negatives, and false negatives for each bug class. Accuracy alone won't tell the full story if the dataset is imbalanced, so we'll report precision, recall, and F1-score too.

TABLE II. EVALUATION METRICS — INSERT MEASURED VALUES AFTER TESTING

Metric	Definition	Value
Accuracy	Correct predictions / total predictions	[Insert %]
Precision	Correct positive predictions / all positive predictions	[Insert %]
Recall	Detected defects / all actual defects	[Insert %]
F1-score	Harmonic mean of precision and recall	[Insert %]
Detection time	Average time required to analyze one input	[Insert ms/s]
Repair success	Validated repairs / repair attempts	[Insert %]
Patch validation rate	Patches passing configured validation / generated patches	[Insert %]

Repair evaluation needs to draw a line between a patch that was generated and one that's actually correct. A patch can compile fine and still be wrong in a way that matters. So at minimum, we want to report three outcomes: generated, validated, and correctly repaired. Ideally, correctness gets checked against independent test cases, not just the same checks used to generate the patch in the first place.

For this prototype, our next step is probably building a small, controlled benchmark of common Python errors that our analyzers already cover: syntax mistakes, undefined

variables, unused variables, wrong operators, missing delimiters, and a few logic errors, each paired with a corrected version. That would give us a reproducible baseline for comparing detection and repair performance going forward, instead of just eyeballing whether the tool feels like it's working.

Our guess is that deterministic error classes will perform well, since AST-based checks don't need a learned model to recognize straightforward syntax and variable issues. Logical defects are a different story. Their correctness depends on what the programmer actually intended, and that's much harder to infer automatically. AI-assisted repair should widen the range of bugs the system can handle, but it also brings a real risk: patches that look reasonable but are wrong. We want to measure that trade-off directly once we have real data, rather than just assume it away.

The interface itself is worth evaluating too. For a tool aimed at students, a clear explanation can matter just as much as raw detection accuracy. A small user study comparing correction time with standard compiler output versus our explanation-and-fix workflow could be genuinely informative here: things like time to fix, number of attempts, and how clear users found the explanations.

There are real limitations we should be upfront about. Static analysis can't catch runtime-dependent errors, and undefined-variable checks get trickier with dynamic language features. AI-generated patches can be computationally expensive and are sensitive to prompt wording and model version, which is something we've already run into while testing the Gemini integration. And validation through tests is only ever as good as the tests themselves. We plan to report these limitations honestly in future work rather than gloss over them.

VIII. CONCLUSION

This paper laid out the design, and in Section VI, the actual build, of an AI Bug Detector and Fixer for automated source-code analysis and repair. We brought together static analysis, structured explanations, deterministic repair rules, AI-assisted patch generation, and explicit validation into one pipeline. The goal, simply put, is to help a user go from "there is an error" to actually understanding it, and where possible, walking away with a tested fix instead of a vague red squiggle.

The core design decision running through all of this is keeping detection, generation, and validation separate. That's what makes the system transparent, and it's what keeps it from presenting an unverified AI output as though it were a guaranteed correct answer. Our

modular Spring Boot + React setup also leaves room to grow, more analyzers, more repair strategies, without us having to rebuild anything from scratch.

Going forward, we want to build out a larger labeled dataset, support more programming languages, bring in more advanced code-language models, improve fault localization, generate unit tests automatically, and strengthen patch validation further. Later versions could also add project-level analysis, security vulnerability detection, code-quality scoring, and explanations tailored to individual students. If we keep evaluating it properly, this AI Bug Detector and Fixer could genuinely grow into a useful debugging assistant, for programming education and for real-world software development alike.

REFERENCES

1. L. Gazzola, D. Micucci, and L. Mariani, "Automatic Software Repair: A Survey," *IEEE Transactions on Software Engineering*, vol. 45, no. 1, pp. 34–67, Jan. 2019, doi: 10.1109/TSE.2017.2755013.
2. T. Le, R. J. Oentaryo, and H. K. Le, "Elixir: Effective Object-Oriented Program Repair," in *Proc. 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017, pp. 648–659, doi: 10.1109/ASE.2017.8115675.
3. H. Tian, X. Tang, A. Habib, S. Wang, K. Liu, X. Xia, J. Klein, and T. F. Bissyandé, "Is this Change the Answer to that Problem?: Correlating Descriptions of Bug and Code Changes for Evaluating Patch Correctness," in *Proc. 37th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2022, pp. 1–13, doi: 10.1145/3551349.3556914.
4. Y. Liu, J. Chen, and others, "Search-Based Efficient Automated Program Repair Using Mutation and Fault Localization," in *Proc. 42nd IEEE Annual Computer Software and Applications Conference (COMPSAC)*, 2018, doi: 10.1109/COMPSAC.2018.00030.
5. D. Marcilio, C. A. Furia, R. Bonifácio, and G. Pinto, "Automatically Generating Fix Suggestions in Response to Static Code Analysis Warnings," in *Proc. 19th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2019, doi: 10.1109/SCAM.2019.00013.
6. Y. Li, S. Wang, and T. N. Nguyen, "DLFix: Context-Based Code Transformation Learning for Automated Program Repair," in *Proc. ACM/IEEE 42nd International Conference on Software Engineering (ICSE)*, 2020, pp. 602–614, doi: 10.1145/3377811.3380345.

7. Y. Li, "Improving Bug Detection and Fixing via Code Representation Learning," in Proc. ACM/IEEE 42nd International Conference on Software Engineering Companion (ICSE-Companion), 2020, pp. 137–139, doi: 10.1145/3377812.3382172.
8. Z. Fan, X. Gao, M. Mirchev, A. Roychoudhury, and S. Mao, "Automated Repair of Programs from Large Language Models," in Proc. IEEE/ACM 45th International Conference on Software Engineering (ICSE), 2023, doi: 10.1109/ICSE48619.2023.00128.
9. "FixGPT: A Novel Three-Tier Deep Learning Model for Automated Program Repair," in IEEE Conference Publication, 2024.
10. "T5APR: Empowering Automated Program Repair Across Languages Through Checkpoint Ensemble," Journal of Systems and Software, vol. 214, 2024, Art. no. 112083.
11. Q.-N. Phung, M. Kim, and E. Lee, "Identifying Incorrect Patches in Program Repair Based on Meaning of Source Code," IEEE Access, 2022, doi: 10.1109/ACCESS.2022.3145983.