IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940



International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 10, October 2026

DOI: 10.17148/IJARCCE.2026.1510130

AutoFix: A Verification-Driven Framework for Automated Program Repair and Security Patching of Python Code

Pranav K[1], Devabala Kurup S[2], Rekha P S[3], Abel Markose[4], Sidharth S[5], Soja S Padmam[6], Girija V R[7]

Department of Computer Science and Engineering (AI & ML), College of Engineering Kottarakkara, Kollam, Kerala, India[1-7]

Abstract: Automated program repair (APR) tools still frequently struggle to generate reliable code. Often, they produce patches that fail to compile, reference undefined names, or trick the system into passing tests simply by deleting core functionality. Furthermore, most of the existing research in security repair is heavily focused on C/C++. In this paper, we review twenty studies covering neural, LLM-based, and vulnerability repair, pinpointing six recurring gaps in the field: uncompilable output, execution-blind training, test-suite overfitting, narrow security coverage, assuming fault locations in advance, and evaluation leakage. To address these limitations, we introduce AutoFix, a verification-driven framework built specifically for Python. AutoFix tracks down and localizes faults by combining AST analysis, tracebacks, and Ochiai ranking. It then maps the underlying weaknesses to their respective CWE categories and uses a LoRA-tuned transformer to generate a ranked list of candidate patches. Before any patch is accepted, it must survive four rigorous sandboxed checks: syntax, compilation, standard tests, and property-based testing. The outcomes of this verification process are then fed back into the system as a graded reward to further fine-tune the model. This paper outlines the framework's overall design, architecture, and evaluation plan, with the actual implementation results set aside for future work.

Keywords: Automated program repair, vulnerability patching, large language models, LoRA fine-tuning, CWE, fault localization, patch verification

I. INTRODUCTION

Software defects are expensive. When a developer encounters a failing test, they have to track down the exact line causing the issue, figure out a fix, and run the entire test suite again just to make sure nothing else broke; doing this manually creates huge maintenance backlogs and makes it dangerously easy for security flaws to slip into production. Automated program repair (APR) tries to speed up this cycle by generating potential patches without requiring human effort [1], [2]. While early tools relied on basic repair templates or limited edit spaces, learning-based tools eventually took over by using models trained on historical bug-fix pairs, and today, large pre-trained code models have pushed the boundaries even further [3], [4]. We now have systems where language models can actually call tools, read files, and run tests to fix bugs that older techniques couldn't touch [5]. Along with this, repair research has expanded beyond standard functional bugs to tackle vulnerabilities, which is critical because a bad patch in those cases can silently leave an entire system exposed [6]–[8].

Despite all this progress, a developer who pastes broken code into a general-purpose assistant still faces a major trust issue. The generated fixes might fail to compile, use variables that don't exist in the current scope, or "pass" a unit test simply by deleting the exact behavior the test was supposed to protect. In fact, a survey of 331 practitioners showed that developers still prefer relying on their colleagues and standard chat assistants over dedicated repair tools, noting that trust remains a major roadblock to adoption [9]. On top of that, the bulk of current security-repair research is heavily focused on C and C++, leaving massively popular languages like Python with surprisingly little support.

This paper makes four main contributions. (i) It brings together twenty studies into a structured review to highlight six clear research gaps. (ii) It proposes AutoFix, a verification-first repair framework for Python designed so that no patch ever reaches the developer unless it successfully passes a strict, sandboxed pipeline covering syntax, compilation, standard tests, and property-based testing. (iii) It formulates a five-level graded verification reward along with a reward-scaled training loss. (iv) It lays out a completely leakage-free evaluation plan. Right now, AutoFix is being presented at the design stage; no empirical results are being claimed just yet. The remainder of the paper is organized as follows:

 This work is licensed under a Creative Commons Attribution 4.0 International License

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940



International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 10, October 2026

DOI: 10.17148/IJARCCE.2026.1510130

Section II reviews the existing literature, Section III details the methods and materials, Section IV discusses findings, current status, and the evaluation plan, and Section V concludes.

II. LITERATURE REVIEW

A. Review Methodology

This review looks at twenty studies published between 2017 and 2025, carefully selected to cover the entire process—from finding the fault to producing verified patches. The selection includes journal articles, conference papers, and preprints. We examined each study to understand its repair objective, the techniques used, the reported results, and any limitations mentioned. From there, we grouped the studies into five main themes: execution-aware neural repair, large language models with parameter-efficient tuning, iterative and agent-based repair, vulnerability repair alongside patch assessment, and finally, datasets, fault localization, and everyday developer practices. Throughout the review, we paid special attention to things like compilability, execution feedback, test-suite overfitting, security coverage, fault localization, and whether the evaluations were actually valid.

B. Neural Repair and Execution-Aware Training

Neural repair essentially treats bug fixing like translating a foreign language—turning buggy code into fixed code. For instance, CURE pre-trains on source code, breaks identifiers down into subwords, and limits its beam search to identifiers actually in scope and realistic patch lengths; this approach successfully fixed 57 Defects4J bugs and 26 QuixBugs [2]. Ye et al. noticed something interesting: because cross-entropy rewards textual similarity to the original reference fix, a completely broken patch that just looks similar to the reference can actually outscore a working patch that happens to be worded differently. By scaling the loss based on whether the code compiled and passed tests, they managed to push the compilable share of the top thirty candidates above 45% [1]. The trade-off is that you have to compile candidate patches during training, and both of these systems are still limited to fixing just a single chunk of code. On a different track, PropR combines typed holes with property-based testing and managed to drop overfitting from 85% to 63%, though it's mainly suited for typed functional languages [10].C. Large Language Models and Parameter-Efficient Tuning

C. Large Language Models and Parameter-Efficient Tuning

When Xia et al. compared nine different models, they showed that larger models predictably repair more bugs, and that giving the model the code that comes after the bug location actually improves the patches [4]. Huang et al. found that fine-tuning ten different code models yielded massive gains ranging from 31% to 1,267% [3]. A later study looking at billion-scale models found that LoRA was the most effective of three adaptation methods, while all of them successfully cut down on memory use [11]. InstructRepair took a different angle, adding bug descriptions and test data to a CodeT5+ prompt, which allowed it to fix 122 bugs—19 of which no other method had managed to fix [12]. However, two major warnings keep popping up in the research: benchmark leakage can artificially inflate results [3], [4], [13], and many of these studies just assume the bug location is already known.

D. Iterative, Agent-Based and Feedback-Driven Repair

Often, a single attempt just isn't enough to fix a bug. ITER takes an iterative approach by repeating the localization, patch generation, and validation steps, which bumped its correct fixes from 41 to 74 on its benchmark [14]. RepairAgent takes things further by letting an LLM decide which tool to call next; it successfully repaired 164 out of 835 bugs—including tough multi-line and multi-file cases—though it comes with a high compute cost [5]. Interestingly, Hanna et al. showed that feedback is only helpful when it actually tells the model something useful: a simple pass/fail binary reward just didn't provide enough signal to properly train a mutation selector, even though the overall share of high-quality patches did go up [15].E. Vulnerability Repair and Patch Assessment

E. Vulnerability Repair and Patch Assessment

Looking at security, Huang et al. turned safety properties into concrete checks and produced correct patches for 32 out of 42 real memory vulnerabilities without causing any runtime regressions; however, their approach hits a wall if all the needed variables can't be found within a single function [7]. Qiang et al. used symbolic execution to ask a crucial question: is the generated patch actually safe itself? [8]. Data-driven systems are also pushing this boundary: VGX generated over 150,000 synthetic vulnerable samples [16], while AIBugHunter bundles localization, classification, severity estimation, and repair suggestions straight into an editor plug-in [17]. VulMaster combines flattened syntax trees with CWE knowledge and alarmingly found about 60% data leakage in a standard benchmark [13], and Feng et al. added a strict validity-and-test gate to any AI-generated security fixes [6]. The catch? Almost all of this incredible work is strictly targeted at C and C++.*F. Datasets, Fault Localization and Developer Practice*

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940



International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 10, October 2026

DOI: 10.17148/IJARCCE.2026.1510130

F. Datasets, Fault Localization and Developer Practice

When it comes to generating datasets, SonarQube-based approaches are great for harvesting defects with exact positions, but they require very careful configuration and Git history parsing to accurately pair every defect with its fix [18]. RepairCAT demonstrated that even a small model can learn to spot typical mistakes in AI-written code, yet trying to read a whole file without knowing where the fault is remains inefficient [19]—this is where crash stack traces prove to be an incredibly valuable signal for localization [20]. Finally, on the human side of things, whether these tools actually get adopted depends heavily on someone's job role, company size, experience level, and preferred language, with a lack of trust remaining the biggest barrier [9]. Table I summarizes all twenty of these studies.

TABLE I SUMMARY OF REVIEWED STUDIES

Study (year)	Contribution and reported result	Open limitation
Ye et al. (2022) [1]	Compilation and test outcomes added to the loss; 207 bugs repaired	Live compilation costly; single-hunk
CURE (2021) [2]	Code-aware search; 57 Defects4J and 26 QuixBugs bugs fixed	Single-hunk; assumes localization
PropR (2022) [10]	Typed holes plus property tests; overfitting 85% to 63%	Search-space growth; typed languages
Xia et al. (2022) [4]	Nine LLMs, three generation styles compared	Prompt-dependent; leakage, cost
Huang et al. (2023) [3]	Ten code models; fine-tuning gains 31%–1,267%	Leakage hampers comparison
Huang et al. (2025) [11]	LoRA, AdaLoRA, IA3 compared; LoRA best	Longer training; complex vulnerabilities
InstructRepair (2025) [12]	Bug details and tests in prompt; 122 bugs fixed	Single-part bugs; location needed
ITER (2024) [14]	Iterative cycles; 74 of 476 Defects4J bugs fixed	Java only; extra time per iteration
RepairAgent (2024) [5]	LLM agent with tools; 164 of 835 bugs fixed	Needs location and failing tests; costly
Hanna et al. (2025) [15]	RL mutation selection; 34.0% vs 24.3% high-quality patches	Fewer bugs fixed (46 vs 49); coarse signal
Huang et al. (2019) [7]	Safety properties; 32 of 42 vulnerabilities patched	Single-function scope only
Qiang et al. (2017) [8]	Symbolic execution checks patches; 15 of 16 detected	Memory errors; external libraries
VGX (2024) [16]	Over 150K vulnerable samples generated	Mainly C; distribution may differ
AIBugHunter (2024) [17]	Editor tool; 90% of practitioners would adopt	C/C++ only; needs industrial validation
VulMaster (2024) [13]	FiD, flattened ASTs, CWE data; 60% leakage found	Multi-hunk; out-of-function context
Feng et al. (2024) [6]	Validity and test gate; 5/16 and 6/17 correct	Single-line fixes; checking time
Del-Hoyo-Gabaldon et al. (2024) [18]	SonarQube API to JSON datasets with positions	False positives; needs Git history
RepairCAT (2024) [19]	Two-stage StarCoder-1B; 14 of 100 bugs fixed	Whole-file reading; 2% perfect fixes
Zhang et al. (2021) [20]	Crash stack traces improve localization	Defects4J only; null-check bugs hard
Meem et al. (2024) [9]	Survey of 331 practitioners on APR use	Small sample; self-reported

G. Research Gaps

When we compare all these studies, six recurring gaps clearly stand out. We've labeled them G1 through G6, and we'll be referring to them throughout the rest of this paper.

1. G1 Uncompilable and hallucinated output: Older neural tools often generate a lot of candidate patches that simply won't build. On top of that, general AI assistants have a bad habit of making up variables that don't exist or quietly watering down the tests just to get a passing grade [1], [2].

 This work is licensed under a Creative Commons Attribution 4.0 International License

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 10, October 2026

DOI: 10.17148/IJARCCE.2026.1510130

2. G2 Training detached from execution: Standard cross-entropy training can't tell the difference between a patch that actually works and one that just looks similar to the right answer. Doing live compilations during the training phase is incredibly expensive [1], and just giving the model a simple binary pass/fail score doesn't provide enough detailed feedback for it to learn effectively [15].
3. G3 Test-suite overfitting: Standard unit tests can easily be tricked into accepting patches that don't actually fix the root cause of the bug. While using property-based checks has helped reduce this issue, that success has mostly been limited to functional programming languages [10].
4. G4 Narrow security coverage: Right now, vulnerability repair research is overwhelmingly focused on C and C++. The current tools really struggle to handle less common "long-tail" vulnerabilities or to understand what's happening in the code outside of the specific target function [7], [13], [16], [17].
5. G5 Assumed localization and single-hunk edits: Most current systems just take it for granted that the exact buggy line is already known, or they operate under the assumption that a single, isolated block of code (a single hunk) is all it takes to fix the problem [1], [2], [12].
6. G6 Evaluation validity: The way these models are evaluated isn't always foolproof. Issues like data leakage and duplicated test samples can artificially inflate the results, making a tool's accuracy look much higher than it actually is [3], [4], [13].

III. METHODS AND MATERIALS

A. System Overview and Architecture

AutoFix operates on a continuous, closed-loop cycle of testing, locating the issue, generating a fix, and verifying the result. It starts with a Python file or an entire repository along with its test suite. The process moves through seven distinct stages: (1) figuring out exactly where the fault is using AST parsing, (2) figuring out how to represent the bug, (3) generating the actual patch, (4) decoding the code carefully, (5) running everything through closed-loop verification, (6) applying the patch and generating a report, and finally (7) using the results to fine-tune the system via reward-based learning.

Alongside this main pipeline, there is a dedicated CWE security subsystem (8) that feeds weakness labels and hints for the fix directly into both the localizer and the generator. Everything is tracked in a database (9) that logs the bugs, the attempted patches, and the final outcomes. If a patch fails the checks, the system either tries the next best candidate or goes back to try and find the fault again. If a patch passes, it's presented to the developer as a clean diff that they can choose to accept or reject (Fig. 1). The entire system is built across five distinct layers—input, analysis, synthesis, verification, and output/storage—and every single module is designed with fixed interfaces. This means you can swap out the localizer, the generator, or the verifier at any time without breaking the rest of the system

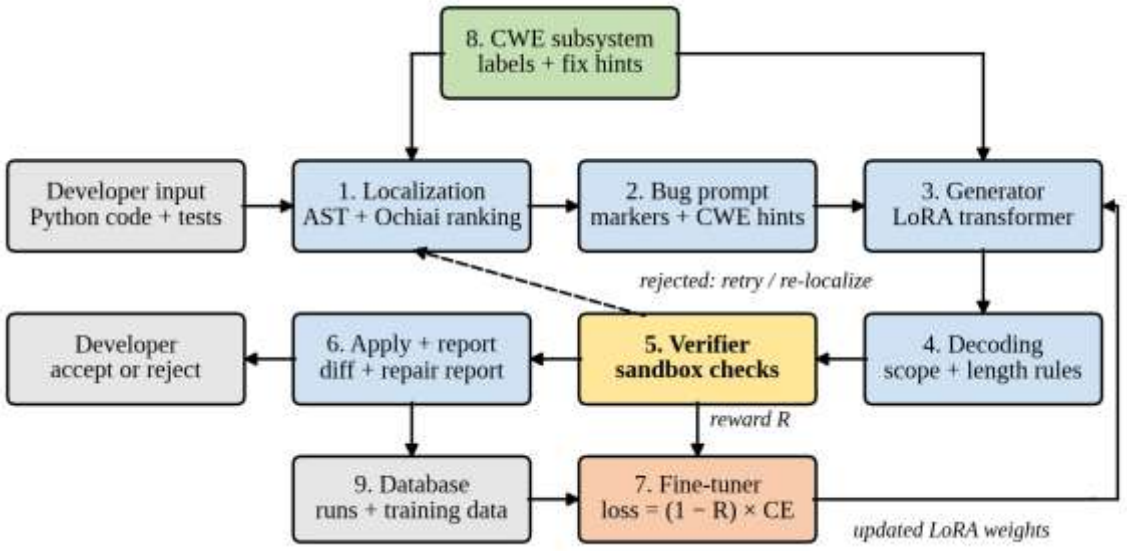


Fig. 1 Architecture of AutoFix. A candidate leaves the verifier only if it passes every check; rejected candidates trigger a retry or re-localization, and the verification outcome becomes the reward that updates the generator.

 This work is licensed under a Creative Commons Attribution 4.0 International License

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 10, October 2026

DOI: 10.17148/IJARCCE.2026.1510130

B. Fault Localization

To track down the source of an error, the localizer first reads the source code using Python's built-in ast module. It then runs the project's test suite and merges the information from crash stack traces [20] with Ochiai spectrum-based ranking to calculate a suspicion score for every single statement:

$$\text{score}(s) = e_f / \sqrt{[(e_f + n_f)(e_f + e_p)]}$$

Here, *ef* represents the number of failing tests that actually run a specific statement *s*, *nf* is the number of failing tests that bypass it, and *ep* stands for the number of passing tests that run it. Because of how this is calculated, statements that are triggered by a lot of failing tests—but very few passing ones—naturally rise to the top of the suspect list. From there, the AST links each specific line back to its surrounding code structure, making sure the generator gets the complete context of the method rather than just an isolated line. Also, unlike older tools that just assume you already know exactly where the bug is, AutoFix will actually go back and restart the localization process from scratch if a round of generated patches fails to fix the problem.

C. CWE Security Subsystem

To handle security issues, a combined rule- and model-based scanner combs through the AST looking for common Python security anti-patterns—things like command injection, SQL injection, and unsafe deserialization. When it spots one, it flags it with the correct CWE label and adds a clear, structured repair hint directly into the prompt. To start, the system is focusing heavily on CWE-787, CWE-416, and CWE-89. Because some vulnerabilities are pretty rare, the system deliberately oversamples them during the fine-tuning stage to balance things out, addressing the long-tail problem pointed out in previous studies [13], [16], [17]. Once a patch is applied, the system runs the exact same scan again just to make absolutely sure the fix didn't accidentally introduce a new security hole

D. Prompt Construction and Patch Generation

For the actual generation, AutoFix relies on a pre-trained code transformer (like CodeT5, StarCoder, or CodeLlama, though the exact choice will be locked in during Phase II). To keep things efficient, it uses Low-Rank Adaptation (LoRA) and 8-bit quantization; this setup makes it entirely possible to tune massive, billion-parameter models on just a standard, single Colab-class GPU [11]. The prompt itself is built in a very specific, consistent order: first, it provides the CWE security guidance, then the context from the tests that failed, and finally the buggy function itself. It wraps the most suspicious lines with CR3-style fault markers (using <BUGS> and <BUGE>) and includes the full method-level context so the model can see the big picture. From there, a beam search generates the top N candidate patches. But before the system actually tries running any of them, a code-aware decoding step kicks in. Building on the approach used by CURE [2], this step strictly blocks any variable names that don't exist in the current scope, uses byte-pair subword tokens, and actively penalizes patches that look suspiciously short or unnaturally long.

E. Sandboxed Closed-Loop Verification

To keep things secure, every candidate patch runs inside a strictly isolated sandbox with a hard time limit, capped resources, zero network access, and its own copy of the source code. A patch must survive four strict gates in order (Fig. 2): (1) AST syntax validation to check the core structure, (2) bytecode compilation, (3) the developer's standard PyTest unit and regression tests, and (4) property-based testing (using Hypothesis or safety assertions) to ensure the fix didn't just overfit to the existing tests [10]. If a patch fails any gate, the error message or test trace feeds right back into the prompt or the localizer. This relies on the proven idea that repeating the localize-patch-validate cycle often catches fixes a single attempt would miss [14]. The loop runs until a solid fix is verified or the time/attempt budget runs out (usually ten minutes). Crucially, even when a patch passes every check, it's never applied automatically—the developer always has the final say and must manually approve the diff.

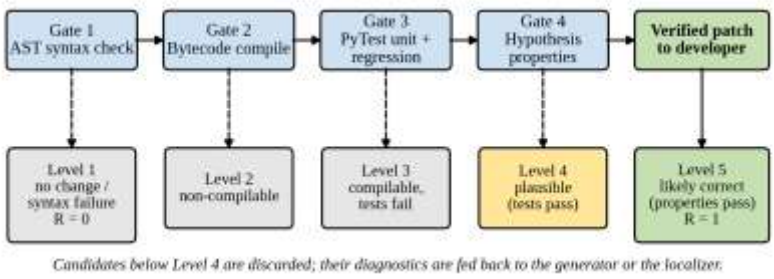


Fig. 2 Four sequential verification gates and the resulting verification levels.

 This work is licensed under a Creative Commons Attribution 4.0 International License

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940



International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 10, October 2026

DOI: 10.17148/IJARCCE.2026.1510130

F. Graded Reward and Fine-Tuning

Every candidate patch is assigned one of five specific verification levels (Table II). This level is then translated into a reward score, *R*, ranging from 0 to 1. The score naturally increases as the verification level gets higher—with *R = 0* assigned to the lowest level and *R = 1* to the highest. The exact scores for the levels in between will be fine-tuned and calibrated during Phase II. Any patch that falls below the "plausible" level is simply thrown out. By using this graded scale instead of a basic pass/fail system, we actively address the feedback weakness that was highlighted in previous research [15].

TABLE II VERIFICATION LEVELS AND TREATMENT

Level	Outcome	Reward R	Treatment
1	No change, or syntax failure	0 (lowest)	Discarded
2	Non-compilable	Low	Discarded
3	Compilable, but tests fail	Medium	Discarded
4	Plausible: unit and regression tests pass	High	Accepted as plausible
5	Likely correct: property tests also pass	1 (highest)	Accepted; best level

During offline fine-tuning, the results logged from the sandbox are used to update the LoRA adapter. We do this using a reward-scaled cross-entropy objective, which builds on the execution-aware concepts introduced in [1]:

$$L_{\text{reward}} = (1 - R) \times \text{CE} \qquad (2)$$

What this means in practice is that a fully verified patch contributes very little loss, so the model naturally reinforces that successful coding style over patches that fail to compile or pass tests (which retain a large loss). Because we pull these rewards from previously logged runs instead of forcing the system to compile code live during training, we completely avoid the heavy training-time costs pointed out in [1]. Finally, the combination of a post-patch CWE re-scan and the rigorous Hypothesis checks acts as our ultimate audit against test overfitting and lingering vulnerabilities.

G. Datasets and Preprocessing

We are planning to use three open datasets for this project (Table III). Since previous research has flagged up to 60% data leakage in a standard vulnerability benchmark [13], we are taking extra precautions. Every single Python bug-fix pair will go through strict AST-level de-duplication before we even think about splitting them into separate training, validation, and test sets. On top of that, we will carefully filter the SonarQube rules to cut down on false positives before pairing any defects with their corresponding fixes from the Git history [18]. Because all of the actual dataset extraction and cleaning is scheduled for Phase II, we aren't reporting any specific record counts just yet.

TABLE III CANDIDATE DATASETS

Dataset	Content	Planned role	Limitation
QuixBugs (Python)	Classic algorithms with single-line bugs and tests	Baseline evaluation	Small; single-line bugs
APPS	Python problems, solutions and test cases	Building AI-generated bug sets	Possible overlap with pre-training data
SonarQube extraction	Bugs, smells and vulnerabilities as JSON	LoRA and CWE training data	False positives; needs Git history

H. Implementation Plan

AutoFix will be built using Python 3.10 (or a newer version), relying on PyTorch, Hugging Face Transformers, and PEFT to power the model. For testing and verification, we'll use PyTest and Hypothesis, while SQLite will handle all the backend data storage. We are aiming to deliver this primarily as a command-line tool and a VS Code extension, though we're also looking into building a full graphical interface down the line. Behind the scenes, the relational database schema will keep track of user sessions, the bugs themselves, all candidate patches, the verification records (along with their specific levels and rewards), the CWE dictionary, and our training samples. As for our concrete design goals, we want at least 40% of our top-30 candidate patches to successfully compile (for context, the baseline in [1] sits just above 45%). We are also aiming to generate these candidates in under two minutes per bug on a standard Colab-class GPU, while making sure we use fixed random seeds and strictly versioned datasets and models so that all of our results are fully reproducible.

 This work is licensed under a Creative Commons Attribution 4.0 International License

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 10, October 2026

DOI: 10.17148/IJARCCE.2026.1510130

IV. RESULTS AND DISCUSSION

A. Findings of the Literature Analysis

Looking at the twenty reviewed studies, you can see a very clear evolution over time (Fig. 3a). Research published between 2019 and 2022 was heavily focused on just getting a single model to produce a compilable, single-chunk patch [1], [2], [7], [10]. From 2023 onward, however, the focus shifted dramatically toward scaling up, parameter-efficient tuning, iterative processes, and autonomous agents [3], [5], [11], [14]. At the same time, security-focused papers started leaning much more into detection, classification, and building new datasets [16]–[18]. The fact that twelve out of the twenty studies we reviewed were published in 2024 or later really highlights just how fast this entire field is moving right now. Our gap analysis (Fig. 3b) showed that narrow security coverage was the most common issue popping up across the studies, closely followed by researchers just assuming the fault location was already known, and issues with evaluation leakage. Just to clarify, the counts shown represent the actual number of reviewed studies we cited as evidence for each of these gaps, rather than a strict statistical measure.

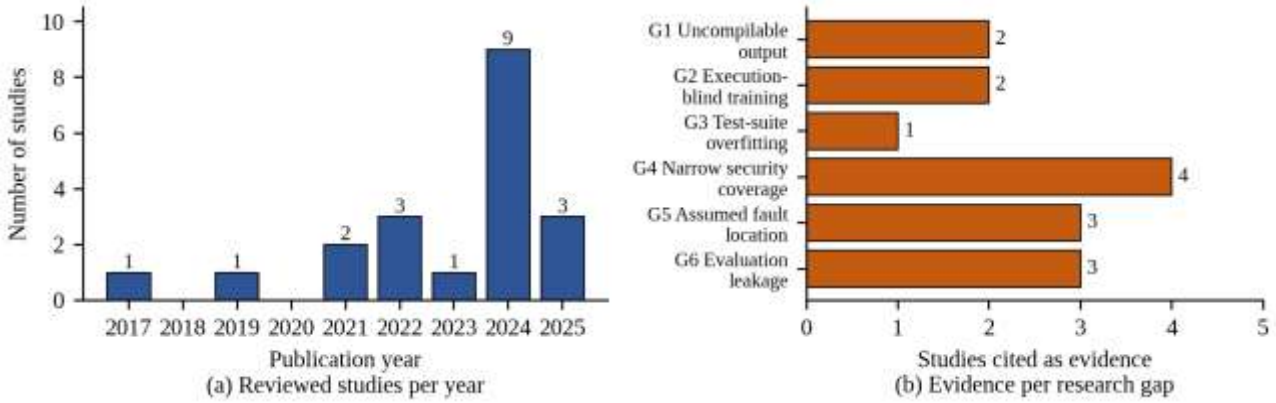


Fig. 3 Review statistics: (a) reviewed studies per publication year and (b) studies cited as evidence for each gap.

B. How the Design Addresses the Gaps

We deliberately mapped each of the identified gaps to a specific design decision in our system (Table IV). The biggest choice we made was placing the verifier directly between the AI generator and the human developer. This ensures that any patches that hallucinate variables or simply fail to compile get thrown out immediately. Better yet, we can then take those exact same verification results and reuse them as a graded training signal for the model, which saves us from having to pay the heavy computational cost of doing live compilations during the training process itself.

TABLE IV RESEARCH GAPS AND AUTOFIX DESIGN RESPONSES

Gap	Evidence	AutoFix response
G1 Uncompilable output	[1], [2]	Code-aware decoding; verifier discards candidates below plausible level
G2 Execution-blind training	[1], [15]	Reward-scaled loss (1 − R) × CE with five graded levels from logged runs
G3 Overfitting	[10]	PyTest regression suites plus Hypothesis property checks; CWE re-scan
G4 Narrow security coverage	[7], [13], [16], [17]	Python CWE subsystem with labels, hints and oversampling of rare CWEs
G5 Assumed localization	[1], [2], [12]	AST, tracebacks and Ochiai ranking; re-localization on failure
G6 Evaluation leakage	[3], [4], [13]	AST-level de-duplication and disjoint train, validation and test splits

C. Current Status

As we wrap up Phase I, we've completely finished formulating the problem, conducting the twenty-study survey, outlining our requirements, and running the feasibility study. We've also finalized the block-level architecture, the data-flow and use-case models, and the overall database schema. The rules for fault localization, prompt design, and exactly when the system should stop generating patches have all been fully mapped out. While the core mechanics for candidate generation, the verification loop, and the reward-based fine-tuning have been designed and architected, they are officially

 This work is licensed under a Creative Commons Attribution 4.0 International License



International Journal of Advanced Research in Computer and Communication Engineering
Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 10, October 2026
DOI: 10.17148/IJARCCE.2026.1510130

slated for implementation during Phase II. Because the system hasn't actually been trained or fully evaluated yet, this paper focuses purely on these design-stage contributions.

D. Planned Evaluation

When Phase II kicks off, we plan to evaluate AutoFix using QuixBugs, bugs derived from APPS, and carefully de-duplicated SonarQube vulnerability pairs, tracking everything against the metrics laid out in Table V. We're setting out to test three main hypotheses. H1: We believe that combining code-aware decoding with a reward-scaled loss will lead to a higher share of compilable patches in the top 30 candidates compared to standard cross-entropy fine-tuning. H2: We expect that using Ochiai ranking alongside direct CWE hints will boost both the exact-match rate and the plausible-patch rate when dealing with security flaws. H3: We hypothesize that running property-based Hypothesis checks will actively lower the test overfitting ratio compared to just relying on standard unit tests. To prove all this, we'll run ablation studies—essentially flipping the switches on and off for decoding, localization, CWE hints, iterative loops, and property checks to see exactly what difference they make—and we'll compare how different underlying models perform when tuned with LoRA.

TABLE V PLANNED EVALUATION METRICS

Aspect	Metric and definition
Syntactic validity	Compilation rate: share of patches that parse and compile without error
Search efficiency	Compilable rate in top-30: valid candidates among the top-30 beam outputs
Functional repair	Plausible patch rate: bugs with at least one patch passing all sandbox checks
Semantic equivalence	Exact match and CodeBLEU against the developer's reference patch
Generalization	Overfitting ratio: unit-test-passing patches rejected by property or held-out tests
Efficiency	Time per bug and memory use under LoRA

E. Limitations and Threats to Validity

Right now, AutoFix is built exclusively for Python, which means our findings might not perfectly translate to other programming languages. Also, just like most of the tools we surveyed [1], [2], [13], AutoFix is held back by the model's context window when it comes to fixing complex bugs that span multiple blocks of code or different files. While property-based testing definitely helps cut down on test overfitting [10], it doesn't get rid of it entirely. Ultimately, testing a patch isn't the same as mathematically proving it works, which is exactly why we require a human developer to approve every single patch. As for the data, datasets built from static analyzers can sometimes include false positives [18]. QuixBugs is a fairly small dataset limited to single-line bugs, and there is always a chance that APPS data might still overlap with the massive datasets these models were originally trained on, even after we try to clean it up. Finally, it's worth remembering that this review only covers twenty studies, and because those studies all used different benchmarks, you can't always do a perfect apples-to-apples comparison of their numbers.

V. CONCLUSION

In this paper, we reviewed twenty different studies on automated program repair—covering everything from execution-aware neural models and fine-tuned code LLMs to iterative agents and specialized security tools. What we found is that while the industry has gotten incredibly good at *generating* code, they haven't quite figured out how to make developers actually *trust* the patches they produce. We pinpointed six major gaps in the current research and proposed AutoFix as a verification-first framework specifically designed for Python. AutoFix uses multiple signals to track down where a fault is, generates ranked candidate patches using code-aware decoding, puts every single candidate through four strict, sandboxed checks, and then actually learns from the results of those checks using a graded reward system and reward-scaled loss. We aren't claiming to have invented a brand-new neural architecture here; the real contribution of AutoFix is taking proven techniques and combining them into a single, cohesive, Python-focused system that refuses to pass on a patch unless it's proven to work, ensuring the developer always remains in complete control. Moving forward, the next steps are to actually build out the modules, fine-tune the generator using LoRA on clean, de-duplicated bug-fix and CWE data, calibrate the specific reward levels, and test our three main hypotheses (H1–H3). Once that's done, we hope to expand the system to handle multi-chunk repairs and eventually roll it out to other programming languages.

REFERENCES

[1] H. Ye, M. Martinez, and M. Monperrus, "Neural program repair with execution-based backpropagation," in *Proc. 44th Int. Conf. Software Engineering (ICSE)*, 2022.

 This work is licensed under a Creative Commons Attribution 4.0 International License



International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 10, October 2026

DOI: 10.17148/IJARCCE.2026.1510130

[2] N. Jiang, T. Lutellier, and L. Tan, "CURE: Code-aware neural machine translation for automatic program repair," in *Proc. 43rd Int. Conf. Software Engineering (ICSE)*, 2021.

[3] K. Huang *et al.*, "Impact of code language models on automated program repair," in *Proc. 45th Int. Conf. Software Engineering (ICSE)*, 2023.

[4] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," *arXiv preprint*, 2022.

[5] I. Bouzenia *et al.*, "RepairAgent: An autonomous LLM-based agent for program repair," *arXiv preprint*, 2024.

[6] S. Feng *et al.*, "Automated and controlled patch generation for communication software vulnerabilities," *Intelligent and Converged Networks*, 2024.

[7] Z. Huang, D. Lie, G. Tan, and T. Jaeger, "Using safety properties to generate vulnerability patches," in *Proc. IEEE Symp. Security and Privacy*, 2019.

[8] W. Qiang *et al.*, "Patch-related vulnerability detection using symbolic execution," *IEEE Access*, 2017.

[9] F. N. Meem, J. Smith, and B. Johnson, "Exploring experiences with automated program repair in practice," in *Proc. 46th IEEE/ACM Int. Conf. Software Engineering (ICSE)*, Lisbon, Portugal, 2024.

[10] M. P. Gissurarson *et al.*, "PropR: Property-based automatic program repair," in *Proc. 44th Int. Conf. Software Engineering (ICSE)*, 2022.

[11] K. Huang, J. Zhang, X. Bao, X. Wang, and Y. Liu, "Comprehensive fine-tuning large language models of code for automated program repair," *IEEE Trans. Softw. Eng.*, 2025.

[12] A. Fu *et al.*, "InstructRepair: Instruct large language models with rich bug information for automated program repair," *IEEE Trans. Inf. Forensics Security*, 2025.

[13] X. Zhou, K. Kim, B. Xu, D. Han, and D. Lo, "Out of sight, out of mind: Better automatic vulnerability repair by broadening input ranges and sources," in *Proc. 46th Int. Conf. Software Engineering (ICSE)*, 2024.

[14] H. Ye and M. Monperrus, "ITER: Iterative neural repair for multi-location patches," in *Proc. 46th Int. Conf. Software Engineering (ICSE)*, 2024.

[15] C. Hanna, A. Blot, and J. Petke, "Reinforcement learning for mutation operator selection in automated program repair," *Automated Software Engineering*, 2025.

[16] Y. Nong, R. Fang, G. Yi, K. Zhao, X. Luo, F. Chen, and H. Cai, "VGX: Large-scale sample generation for boosting learning-based software vulnerability analyses," 2024.

[17] M. Fu, C. Tantithamthavorn, T. Le, Y. Kume, V. Nguyen, D. Phung, and J. Grundy, "AIBugHunter: A practical tool for predicting, classifying and repairing software vulnerabilities," 2024.

[18] del-Hoyo-Gabaldon *et al.*, "Automatic dataset generation for automated program repair of bugs and vulnerabilities through SonarQube," 2024.

[19] Jiang and Wu, "RepairCAT: Applying large language model to fix bugs in AI-generated programs," 2024.

[20] J. Zhang, K. Liu, D. Kim, L. Li, Z. Liu, J. Klein, and T. F. Bissyandé, "Revisiting test cases to boost generate-and-validate program repair," 2021.

 This work is licensed under a Creative Commons Attribution 4.0 International License