Using Artificial Intelligence Techniques for Error Detection and Repair in Software Development

Abdulaziz Aladwani, Sultan Alsamaani, Turki Alrumaykhani, Mohamed Tahar Ben Othman
Department of Computer Science-College of Computer, Qassim University, Qassim, Saudi Arabia

***Abstract*—Software bugs remain one of the most costly challenge in software engineering, consuming significant development time and resources. Recent advances in Artificial Intelligence (AI), particularly deep learning and large language models (LLMs), have shown remarkable potential in automating the detection and repair of software errors. This study presents a comprehensive survey and comparative analysis of AI-based techniques for error detection and automated program repair (APR). We categorize existing approaches into traditional search-based methods, learning-based neural machine translation models, and emerging LLM-based repair systems. We evaluate these techniques across standard benchmarks, including Defects4J and SWE-bench, comparing their effectiveness in terms of bugs fixed, patch correctness, and scalability. Our analysis reveals that while LLM-based approaches significantly outperform traditional methods in repair capability, challenges remain in patch correctness validation, computational cost, and generalization to real-world codebases. Our results show that LLM-based tools, such as ChatRepair, can correctly fix 114 out of 395 benchmark bugs at just $0.42 per fix, fixing 2.6× as many bugs as the best traditional method (a 165% increase). We discuss open challenges and propose future research directions toward more reliable AI-assisted software development.**

***Keywords*—Automated program repair; bug detection; deep learning; large language models; software engineering; neural machine translation; defect prediction; code analysis**

I. Introduction

Software defects are an inherent part of the software development lifecycle, costing the global economy an estimated $1.56 trillion annually, according to the Consortium for Information and Software Quality (CISQ). Debugging and fixing software errors account for a substantial portion of development effort, with studies estimating that developers spend 35–50% of their time on debugging activities. The consequences of undetected bugs range from minor inconveniences to catastrophic failures: the 2014 Heartbleed vulnerability in OpenSSL affected an estimated 17% of all secure web servers, while the 2017 Equifax breach—caused by a single unpatched vulnerability—exposed the personal data of 147 million individuals. As software systems grow in complexity, with modern applications spanning millions of lines of code across distributed architectures, the need for automated approaches to detect and repair errors becomes increasingly critical.

The traditional software debugging workflow follows a manual process: a developer identifies a failing test or user report, localizes the fault through debugging tools and code inspection, designs a fix, implements the patch, and validates it against the test suite. This process is time-consuming, error-prone, and does not scale with the pace of modern software development where thousands of commits are made daily in large organizations.

Artificial Intelligence has emerged as a transformative force in software engineering, offering new paradigms for automating tasks traditionally performed by human developers. The application of AI to error detection and repair spans multiple generations of techniques: from early search-based approaches like GenProg [1] that use genetic programming to evolve patches, to modern large language models that can understand and generate code with remarkable fluency [2]. The key insight driving this progress is that software bugs follow recurring patterns—a study of over 700 real-world Java bugs found that approximately 25% involve conditional statement errors, 20% involve method call changes, and 15% involve variable replacements [3].

Such single-statement errors—for instance, a bisection method in the Apache Commons Math library (Defects4J, Math-70) that returns its initial endpoint instead of the computed root, illustrated concretely in Fig. 3 of Section III—are precisely the kind of recurring pattern that AI techniques can learn to detect and correct.

The evolution of AI-based program repair can be broadly characterized in three waves:

- Search-based and constraint-based repair (2009–2017): Techniques such as GenProg [1], Prophet [4], and Nopol [5] that search for patches using test suites as correctness oracles.
- Learning-based repair (2018–2021): Neural machine translation models like SequenceR [6], CoCoNuT [7], and CURE [8] that learn bug-fix patterns from large code corpora.
- LLM-based repair (2022–present): Pre-trained language models, including CodeBERT [9], Codex [2], and ChatGPT-based systems [10] that leverage massive pre-training for zero-shot or few-shot repair.

Similarly, AI-driven bug detection has progressed from traditional machine learning classifiers for defect prediction [11] to sophisticated deep learning models that analyze program semantics through graph neural networks [12] and pre-trained code representations [13].

This study makes the following contributions:

- A comprehensive taxonomy of AI techniques for software error detection and repair, covering 32 key works from 2012 to 2025.

- A comparative analysis of traditional, learning-based, and LLM-based approaches across standard benchmarks.
- An identification of open challenges and future research directions in AI-assisted software debugging.

Fig. 1 illustrates the taxonomy of AI techniques for error detection and repair covered in this survey.

The remainder of this study is organized as follows: Section II reviews related work. Section III provides theoretical background on the key AI techniques. Section IV describes our survey methodology. Section V presents our comparative case studies. Section VI discusses results and findings. Section VII discusses threats to validity. Section VIII concludes with future work directions.

II. Related Work

A. Traditional Automated Program Repair

The field of automated program repair was pioneered by GenProg [1], which introduced genetic programming to evolve patches by mutating program statements guided by test suite fitness. While groundbreaking, GenProg's patches often lacked semantic correctness. Prophet [4] addressed this by learning probabilistic models from successful human patches to prioritize correct repairs. Angelix [14] introduced symbolic analysis for scalable multiline patch synthesis, while Nopol [5] specialized in repairing conditional statement bugs through constraint-based synthesis.

Template-based approaches further improved repair quality. SimFix [15] leveraged existing patches and similar code at AST granularity, fixing 34 bugs on the Defects4J benchmark. TBar [3] revisited template-based repair with systematically refined fix templates, achieving 43 bug fixes on Defects4J.

B. Learning-Based Program Repair

The application of neural machine translation (NMT) to program repair marked a paradigm shift. SequenceR [6] was among the first end-to-end sequence-to-sequence models for repair, incorporating a copy mechanism to handle out-of-vocabulary tokens. DLFix [16] employed a two-tier tree-based RNN to learn contextual code transformations, fixing 2.5× more bugs than previous baselines.

CoCoNuT [7] combined context-aware NMT models using ensemble methods with a multi-stage attention mechanism. CURE [8] pre-trained NMT models on large code corpora and integrated static checking to ensure generated patches contain valid identifiers.

Industrial deployments validated the practical impact of learning-based repair. Getafix [17], deployed at Facebook, used hierarchical clustering of fix patterns from static analysis violations, demonstrating that learning-based repair can operate at production scale in industrial environments.

C. LLM-Based Error Detection and Repair

The emergence of large language models has substantially advanced the state-of-the-art. Xia and Zhang [18] demonstrated that LLMs can repair 3.3× more bugs than prior approaches through zero-shot cloze-style repair (AlphaRepair). ChatRepair [10] leveraged conversational LLMs with iterative feedback, fixing 162 out of 337 Defects4J bugs at $0.42 per fix. Chen et al. [19] taught LLMs to self-debug by explaining and correcting their own generated code. Most recently, RepairAgent [20] introduced autonomous agent-based repair with planning and iterative patch refinement.

For vulnerability-specific repair, VulRepair [21] fine-tuned T5 models to fix 745 out of 1,706 real-world CVE vulnerabilities. RewardRepair [22] incorporated reinforcement learning with execution-based feedback to guide neural repair models.

D. AI-Based Bug and Vulnerability Detection

DeepBugs [23] pioneered learning-based bug detection by training neural networks on identifier names to detect swapped arguments and wrong operators. For vulnerability detection, Devign [12] applied graph neural networks to learn comprehensive program semantics, while VulDeePecker [24] used bidirectional LSTMs on code gadgets extracted through program slicing.

Pre-trained code models significantly advanced detection capabilities. CodeBERT [9] provided the first bimodal pre-trained representation for code and natural language. GraphCodeBERT [13] incorporated data flow information into pre-training, improving code understanding tasks. LineVul [25] achieved 160–379% higher F1-measure using Transformer-based architectures for line-level vulnerability prediction.

E. Software Defect Prediction

Machine learning has been extensively applied to predict defective software modules. Yang et al. [11] introduced deep belief networks for just-in-time defect prediction from code commits. Transfer learning approaches [26] enabled cross-project defect prediction by aligning feature distributions across different software projects.

F. Existing Surveys

Several surveys have systematically reviewed this field. Gazzola et al. [27] provided a comprehensive taxonomy of APR techniques. Hou et al. [28] reviewed 395 papers on LLMs for software engineering. Our work differs by providing an integrated comparative analysis spanning detection and repair, with a focus on the recent LLM-based paradigm shift.

III. Theoretical Background

Fig. 2 shows the chronological evolution of key techniques across the three waves of AI-based program repair, with bug/vulnerability *detection* milestones marked by a dagger (†) to distinguish them from the repair techniques.

A. Formal Foundations of Program Repair

Automated program repair can be formally defined as follows. Given a buggy program P_b and a test suite $T = \{t_1, t_2, \ldots, t_n\}$ where at least one test $t_f \in T$ fails on P_b, the goal is to produce a patched program P_p such that all tests in T pass:

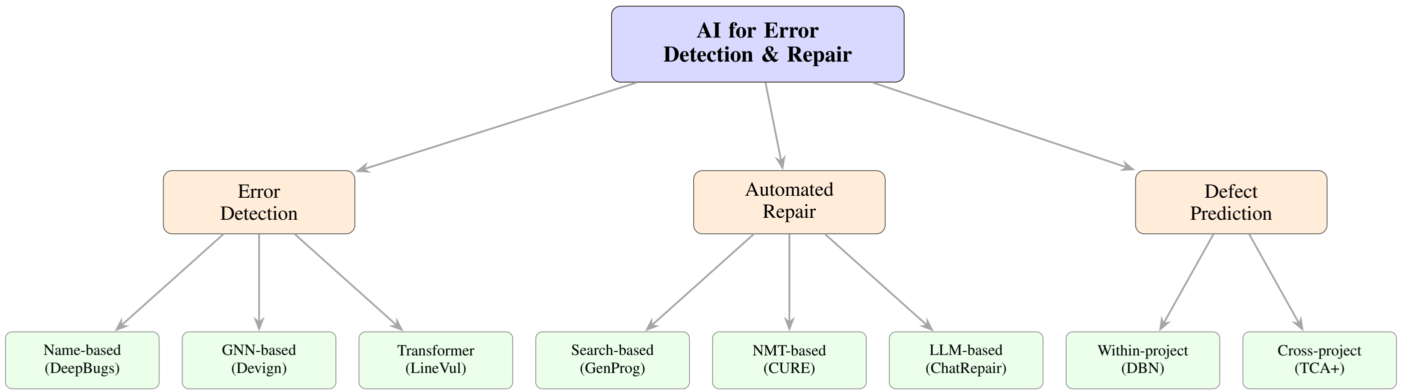

Fig. 1. Taxonomy of AI techniques for software error detection and repair.

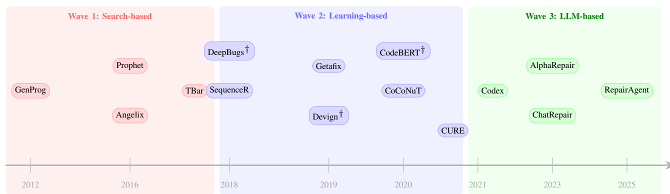

Fig. 2. Evolution of AI-based program repair and detection (†) techniques (2012–2025). Repair techniques are organized into three waves. Detection milestones (DeepBugs, Devign, CodeBERT), marked with †, are shown on the same axis for chronological context only and do not constitute repair "waves".

$$\mathrm{APR}(P_b, T) = P_p \text{ such that } \forall t_i \in T : t_i(P_p) = \text{pass} \quad (1)$$

This formulation connects to the theory of computation through the concept of program equivalence and the undecidability of determining semantic correctness from a finite test suite (the *overfitting problem*).

To make this formulation concrete, Fig. 3 shows a representative single-statement repair from Defects4J (Math-70): a bisection method that incorrectly returns its initial endpoint instead of invoking the root-finding computation. This is precisely the class of recurring single-statement error that an AI-based repair tool can detect and correct automatically.

```
// Buggy version:
return initial;     // <-- always returns
                    //     the initial guess
// Fixed version (by AI repair):
return solve(f, min, max);  // <-- computes
                             //     the root
```

Fig. 3. Example of bug fix from Defects4J Math-70: a single-line change that an LLM-based tool can detect and repair automatically.

B. General APR Workflow

Algorithm 1 presents the general workflow shared by most automated program repair systems, regardless of the underlying technique used to generate candidate patches.

Algorithm 1 Conversational Automated Program Repair

Require: Buggy program P_b, test suite T, patch generator G, max iterations N
Ensure: Repaired program P_p or failure
1: $loc \leftarrow$ FaultLocalization(P_b, T)
2: **for** each suspicious location $l \in loc$ **do**
3: $ctx \leftarrow \emptyset$ {conversational feedback context}
4: **for** $iter \leftarrow 1$ to N **do**
5: $candidates \leftarrow G(P_b, l, ctx)$ {G conditioned on feedback}
6: **for** each patch $p \in candidates$ **do**
7: $P_p \leftarrow$ Apply(P_b, p)
8: $result \leftarrow$ RunTests(P_p, T)
9: **if** $result$ = *all pass* **then**
10: **return** P_p
11: **else**
12: $ctx \leftarrow ctx \cup$ Diagnostics($result$) {errors, failing tests}
13: **end if**
14: **end for**
15: **end for**
16: **end for**
17: **return** *failure*

The key differentiator between approaches is the patch generator G: search-based methods use genetic programming, NMT-based methods use encoder-decoder neural networks, and LLM-based methods use pre-trained language models. Crucially, the outer iteration loop (lines 4–5) captures the multi-turn, conversational behavior of modern LLM-based and agentic tools such as ChatRepair [10] and RepairAgent [20]: after each failed validation attempt, diagnostic feedback—compiler error messages and failing-test traces—is accumulated in the context ctx and fed back into the generator, allowing it to refine subsequent patches. Traditional search- and template-based methods are recovered as the special case $N = 1$, where the generator receives no execution feedback (a passive generate-and-validate loop).

C. Search-Based Approaches

Search-based APR techniques explore the space of possible patches using metaheuristic algorithms. GenProg [1] models

repair as a search over program variants using genetic programming, where mutations (insert, delete, replace) are applied to AST nodes and fitness is measured by test suite passing rate.

D. Neural Machine Translation for Code

Learning-based APR formulates repair as a translation task from buggy code to fixed code:

$$P(\text{fix} \mid \text{bug}) = \prod_{i=1}^{m} P(y_i \mid y_1, \ldots, y_{i-1}, \mathbf{x}) \tag{2}$$

where, $\mathbf{x}$ is the buggy code sequence and y_i are tokens of the fixed code, generated autoregressively by an encoder-decoder architecture.

E. Transformer Architecture and Pre-training

The Transformer [29] architecture, based on self-attention mechanisms, has become the foundation for modern code models. The core self-attention mechanism computes weighted relationships between all token pairs in the input sequence, enabling the model to capture long-range dependencies in code—such as matching a variable declaration with its usage hundreds of lines later. Pre-trained models like CodeBERT [9] learn bidirectional code representations through masked language modeling (MLM) on large code corpora. This enables zero-shot and few-shot capabilities for downstream tasks including bug detection and repair.

F. Graph Neural Networks for Code Analysis

Graph neural networks model code as graphs (e.g., control flow graphs, data flow graphs, ASTs) to capture structural properties. Devign [12] processes composite code property graphs through gated graph neural network layers, where each node iteratively aggregates information from its neighbors using a GRU-based update function. This graph-based formulation allows the model to reason about program semantics beyond the linear token sequence, capturing control dependencies, data flow relationships, and calling conventions that are critical for accurate vulnerability detection.

G. Reinforcement Learning for Program Repair

Reinforcement learning (RL) offers a complementary paradigm to supervised learning for program repair. In the RL formulation, the repair agent interacts with an environment (the program and its test suite) by generating candidate patches (actions) and receiving feedback based on test execution results (rewards). RewardRepair [22] uses a policy-gradient approach where the neural repair model is trained to maximize expected rewards from test execution, directly optimizing for test-passing patches rather than merely imitating human-written fixes. This approach has been extended by training critic models that predict patch correctness before execution, enabling more efficient exploration and early filtering of invalid candidates.

H. Transfer Learning and Domain Adaptation

Transfer learning has proven essential for addressing the data scarcity problem in software error detection. Within-project defect data is often limited, making it difficult to train accurate models. Transfer defect learning [26] addresses this by using Transfer Component Analysis (TCA+) to learn a transformation matrix that aligns the feature distributions of source and target projects in a shared feature space, minimizing the domain discrepancy while preserving the discriminative structure of the data. Pre-trained code models like CodeBERT [9] implicitly perform transfer learning by learning general-purpose code representations that can be fine-tuned for specific tasks with minimal labeled data.

IV. Methodology

This section describes the systematic methodology adopted for conducting this comparative survey, following established guidelines for systematic literature reviews in software engineering.

A. Research Questions

This study aims to address the following research questions:

- RQ1: How do traditional, learning-based, and LLM-based APR techniques compare in terms of bugs fixed and patch correctness?
- RQ2: What are the strengths and limitations of AI-based bug detection approaches across different vulnerability types?
- RQ3: How do repository-level, agent-based systems perform on real-world issue resolution (SWE-bench), and how does this compare with isolated-bug benchmarks?
- RQ4: What are the key challenges and future directions for AI-assisted error detection and repair?

B. Search Strategy and Paper Selection

We conducted a systematic search across four major digital libraries: IEEE Xplore, ACM Digital Library, Springer Link, and arXiv. The search was performed using the following query structure:

("automated program repair" OR "bug detection" OR "vulnerability detection" OR "defect prediction") AND ("deep learning" OR "neural" OR "machine learning" OR "language model" OR "transformer")

The initial search returned over 2,500 candidate papers. We applied the following inclusion and exclusion criteria to filter relevant works:

Inclusion criteria:

- Published in top-tier SE/AI venues (ICSE, FSE, ASE, ISSTA, NeurIPS, ICLR, ICML, TSE, TOSEM, OOPSLA)
- For papers published in 2012–2022, a minimum of 50 citations was required as an impact threshold. For

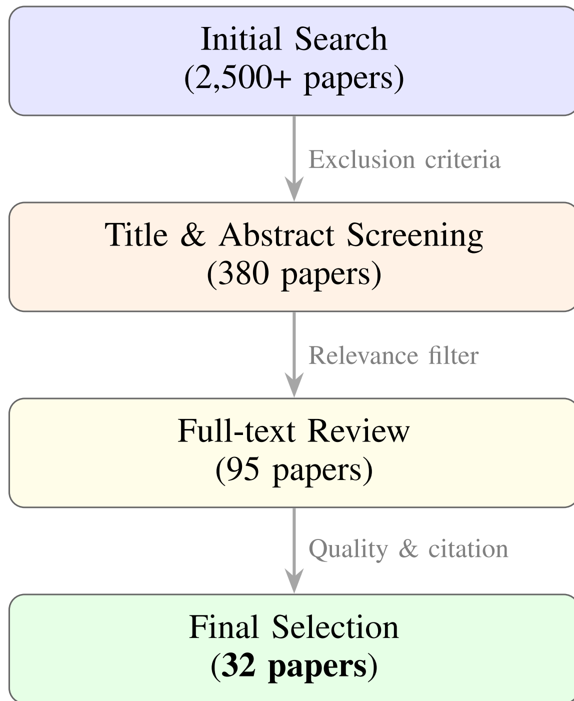

Fig. 4. Systematic paper selection process (PRISMA-inspired flow).

TABLE I. COMPARISON OF DIMENSIONS AND EVALUATION METRICS

Dimension	Metrics
Effectiveness	Bugs fixed, precision, recall, F1 score
Scalability	Training time, inference latency, memory usage
Generalizability	Cross-project, cross-language performance
Correctness	Ratio of correct to plausible patches
Practicality	Industrial deployment, cost per fix, ease of integration

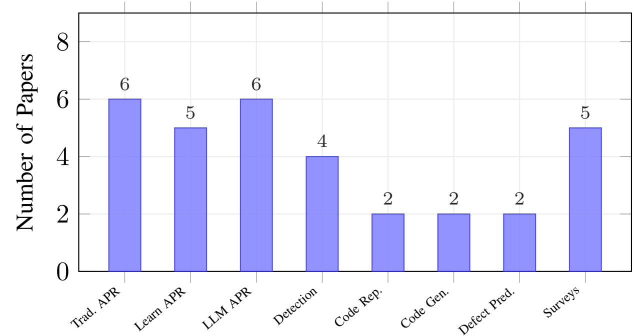

Fig. 5. Distribution of the 32 selected papers across research categories.

recent papers (2023–2025), the citation requirement was waived—since recent work cannot plausibly have accumulated comparable citations—and inclusion was instead based on venue quality, methodological novelty, and relevance, in order to capture emerging trends

- Primary focus on AI/ML/DL techniques for error detection, localization, or repair
- Published between 2012 and 2025

Exclusion criteria:

- Papers focusing solely on test generation without bug detection/repair
- Short papers, tool demonstrations, or posters under 6 pages
- Papers not evaluated on established benchmarks
- Duplicate or extended versions of already-included works

After applying these criteria, screening titles and abstracts, and performing full-text review, we selected 32 papers for detailed analysis. Fig. 4 illustrates the paper selection process.

C. Comparison Framework

We compare techniques across five key dimensions, each capturing a different aspect of practical utility. Table I summarizes these dimensions and their associated metrics.

D. Data Extraction and Analysis

For each selected paper, we extracted: 1) the AI technique category, 2) the training data source and size, 3) the evaluation benchmarks used, 4) quantitative results on standard metrics, and 5) reported limitations. We organized findings by technique category and synthesized cross-cutting observations to answer our research questions.

We emphasize that this work is a *secondary* study (a comparative literature survey): we did not train, fine-tune, or re-evaluate any model ourselves, and consequently performed no data cleaning, class balancing (e.g., SMOTE), or train/test partitioning. The only “pipeline” in this study is the paper-selection process of Fig. 4. All quantitative results reported in Section V and Section VI are taken verbatim from the respective primary studies under their original experimental protocols; questions of data leakage or resampling order therefore pertain to those primary works rather than to this survey. Where primary studies use incomparable configurations, we flag this explicitly in Section VII.

Fig. 5 shows the distribution of selected papers across the eight categories defined in our taxonomy.

V. CASE STUDIES AND COMPARATIVE ANALYSIS

A. Benchmark: Defects4J

Defects4J [30] is the most widely used benchmark for evaluating APR techniques, containing 835 real bugs from 17 open-source Java projects. Table II compares key approaches on Defects4J v1.2 (395 bugs).

Fig. 6 visualizes the comparison between correct and plausible patches across the three categories.

B. Benchmark: SWE-bench

SWE-bench [31] evaluates LLMs on 2,294 real GitHub issues from 12 popular Python repositories including Django,

TABLE II. APR PERFORMANCE ON DEFECTS4J V1.2 (395 BUGS)

Technique	Category	Correct	Plausible
GenProg	Search	5	27
Nopol	Constraint	5	22
SimFix	Template	34	56
TBar	Template	43	81
SequenceR	NMT	14	21
DLFix	NMT	30	46
CoCoNuT	NMT	28	44
CURE	NMT	44	57
AlphaRepair	LLM	80	110
ChatRepair	LLM	**114**	**162**

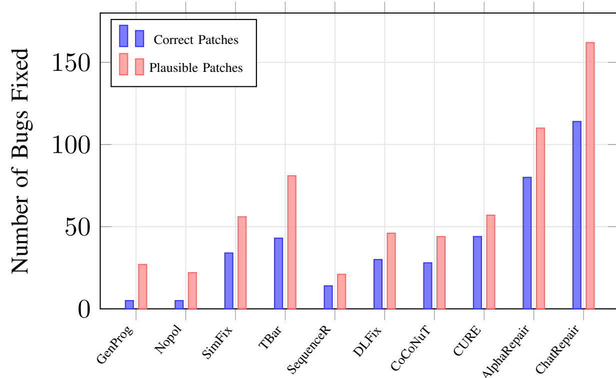

Fig. 6. Comparison of APR techniques on Defects4J v1.2: correct vs. plausible patches across three categories.

Flask, scikit-learn, and sympy. Unlike Defects4J which provides isolated single-function bugs with failing tests, SWE-bench presents realistic software engineering challenges that require: 1) understanding natural language issue descriptions, 2) navigating large codebases to localize the relevant files, 3) implementing multi-file patches that maintain consistency across the codebase, and 4) passing existing regression tests.

Table III summarizes the performance of various approaches on SWE-bench Lite (300 representative instances).

TABLE III. PERFORMANCE ON SWE-BENCH LITE (300 INSTANCES)

Approach	Resolved (%)	Year
RAG + GPT-4 (baseline)	1.96	2024
SWE-Agent + GPT-4	12.47	2024
AutoCodeRover	19.00	2024
Aider + GPT-4o	26.33	2024
RepairAgent	32.00	2025

Fig. 7 visualizes the rapid improvement on SWE-bench Lite.

The dramatic gap between isolated bug benchmarks (ChatRepair: 33.8% on Defects4J) and real-world issue resolution (best: 32% on SWE-bench Lite) underscores that real software maintenance requires significantly more than pattern-based patching—it demands holistic codebase understanding.

C. Cost and Efficiency Analysis

A critical practical consideration is the cost-effectiveness of different approaches. Table IV compares the approximate

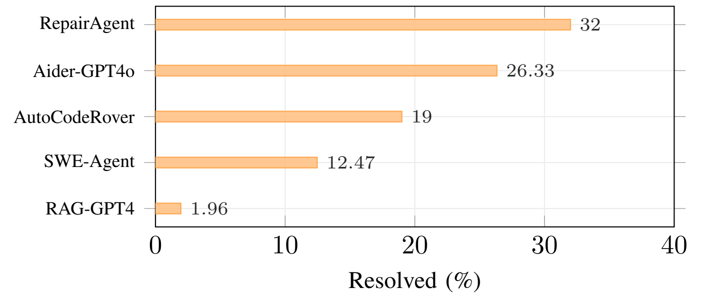

Fig. 7. Resolution rates on SWE-bench Lite (300 real-world GitHub issues).

computational requirements and costs across technique categories.

TABLE IV. COST AND RESOURCE COMPARISON ACROSS APR CATEGORIES.

Category	Training	Inference	Cost/Fix
Search-based	None	Hours	CPU only
Template	None	Minutes	CPU only
NMT-based	GPU days	Seconds	$100+ train
LLM (fine-tuned)	GPU weeks	Seconds	$1000+ train
LLM (API-based)	None	Seconds	$0.42–$5

The API-based LLM approach represents a fundamental shift in the economics of automated repair: by leveraging pre-trained models through APIs, organizations avoid the substantial upfront cost of training while achieving superior repair rates. However, this model introduces dependencies on external service providers and raises concerns about code confidentiality when proprietary code is sent to third-party APIs.

D. Bug Detection Performance

Table V summarizes the performance of key bug detection approaches.

TABLE V. BUG/VULNERABILITY DETECTION PERFORMANCE

Approach	Technique	F1 (%)	Dataset
DeepBugs	Name-based NN	89.0	JS bugs
Devign	GNN	60.1	C/C++ vulns
VulDeePecker	BiLSTM	85.6	NVD vulns
LineVul	Transformer	91.0	Big-Vul
CodeBERT	Pre-trained	62.1	Devign dataset

Fig. 8 provides a visual comparison of bug detection F1 scores.

VI. RESULTS AND DISCUSSION

A. RQ1: Comparative Effectiveness of APR Approaches

Our analysis reveals a clear progression in repair capability across the three generations of APR techniques. As shown in Table II and Fig. 6, LLM-based approaches (ChatRepair: 114 correct fixes) dramatically outperform both traditional methods (TBar: 43) and early learning-based methods (SequenceR: 14).

Key findings include:

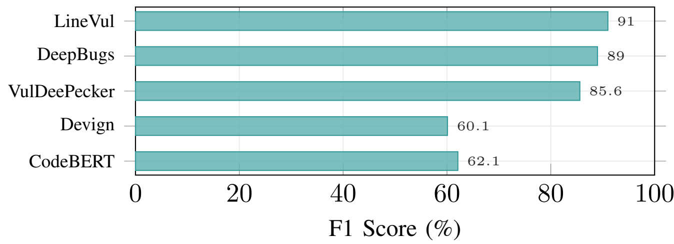

Fig. 8. F1 scores of AI-based bug/vulnerability detection approaches.

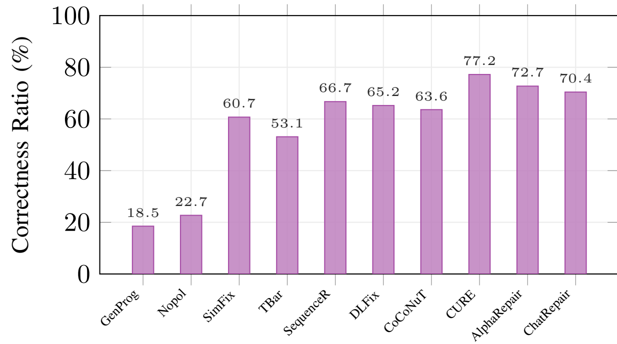

Fig. 9. Patch correctness ratio (correct÷plausible × 100) for each APR technique on Defects4J v1.2.

- LLMs fix 2.6× as many bugs as the best template-based approach on Defects4J (114 vs. 43 correct fixes, a 165% increase), demonstrating the power of large-scale pre-training on diverse code corpora.
- The overfitting problem persists: Even ChatRepair produces 48 plausible but incorrect patches (162 plausible – 114 correct), highlighting that test suite adequacy remains a fundamental bottleneck. This connects to Rice's theorem in the theory of computation—determining semantic correctness from a finite test suite is undecidable in general.
- Cost-effectiveness: ChatRepair's $0.42 per fix represents a paradigm shift in repair economics, making LLM-based repair practical for industrial adoption compared to the estimated $50–$150 cost of manual developer debugging per bug.
- Complementary strengths: Template-based methods (TBar) achieve a higher correct-to-plausible ratio (53%) compared to LLM-based methods (ChatRepair: 70%), suggesting that template methods, while fixing fewer bugs, produce more precise patches within their scope.

Fig. 9 visualizes the patch correctness ratio (correct/plausible) for each technique, revealing how precision varies independently of volume.

An important observation is the diminishing returns within each category. While the jump from Wave 1 (GenProg: 5 fixes) to Wave 2 (CURE: 44 fixes) is an 8.8× increase in correct fixes, the jump from Wave 2 to Wave 3 (ChatRepair: 114 fixes) is only 2.6×, so the rate of improvement is slowing. This suggests that benchmark saturation may be approaching, and future progress may require fundamentally different evaluation methodologies beyond Defects4J.

B. RQ2: Strengths and Limitations of Detection Approaches

Bug detection approaches show complementary strengths, as visualized in Fig. 8. Our analysis identifies a clear trade-off between specialization and generalization:

- Name-based approaches (DeepBugs, F1: 89%) excel at catching surface-level mistakes such as swapped function arguments and incorrect operators. Their key advantage is speed—they operate on identifier embeddings without expensive program analysis. However, they fundamentally cannot detect semantic bugs that do not manifest in naming patterns.
- Sequence-based approaches (VulDeePecker, F1: 85.6%) use bidirectional LSTMs on code slices to capture sequential dependencies. They handle common vulnerability patterns (buffer overflows, use-after-free) well but struggle with bugs that require understanding global program state or inter-procedural data flow.
- GNN-based approaches (Devign, F1: 60.1%) capture deeper program semantics through graph representations of code. While theoretically more expressive, their lower F1 scores reflect challenges with imbalanced datasets (vulnerable functions are rare in practice) and the difficulty of constructing accurate program dependency graphs at scale.
- Transformer-based approaches (LineVul, F1: 91%) achieve the highest detection rates by leveraging pre-trained representations. They provide line-level localization capabilities that are valuable for developer workflows. Their main limitation is the reliance on large pre-training datasets and the potential for distribution shift when applied to programming languages or frameworks not well-represented in the training data.

Fig. 10 plots the maximum number of correct fixes achieved by the best technique in each year, illustrating the accelerating progress driven by each technological wave.

A notable trend is the convergence toward pre-trained models as a foundation for detection tasks. CodeBERT's relatively lower F1 score (62.1%) on the Devign dataset demonstrates that general pre-training alone is insufficient—task-specific fine-tuning with appropriate training objectives (as in LineVul) remains essential for competitive performance.

C. RQ3: Repository-Level Agentic Repair on SWE-bench

As Table III and Fig. 7 shows, agency—not merely model scale—drives repository-level performance. A retrieval-augmented prompting baseline (RAG + GPT-4) resolves only 1.96% of SWE-bench Lite instances, whereas wrapping comparable models in agentic scaffolding that can read files, run tests, and iterate lifts resolution dramatically: SWE-Agent (12.47%), AutoCodeRover (19.00%), Aider + GPT-4o (26.33%), and RepairAgent (32.00%). This is a 16.3× jump

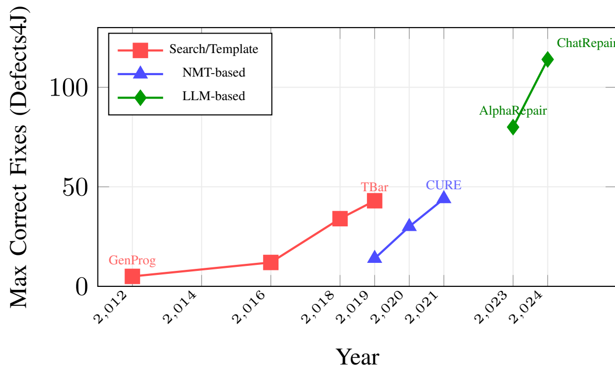

Fig. 10. Progress in maximum correct fixes on Defects4J over time, by technique category.

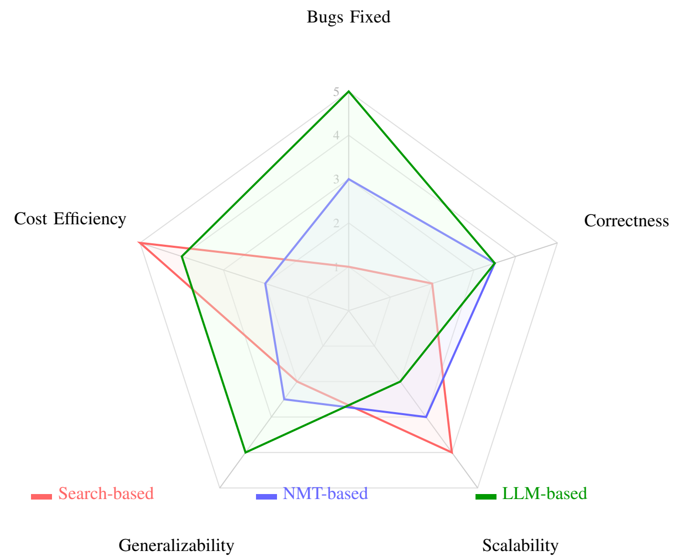

Fig. 11. Multi-dimensional comparison of APR categories across five quality attributes (1=lowest, 5=highest).

from the passive baseline to the best agent, confirming that the ability to interact with the environment—rather than emit a single patch—is the dominant factor at the repository level. This directly mirrors the conversational feedback loop formalized in Algorithm 1.

At the same time, even the best agents lag far behind isolated-bug performance. ChatRepair resolves 33.8% (114/337) of Defects4J bugs it attempts, yet the best agent on SWE-bench Lite reaches only 32%—and does so on issues that are pre-filtered to be "Lite". The gap underscores that fault localization across an unfamiliar codebase, not patch synthesis, is now the binding constraint. We, therefore, treat SWE-bench resolution rate, rather than Defects4J fix count, as the more faithful proxy for practical maintenance capability; the open problems this exposes are analyzed in RQ4.

D. RQ4: Open Challenges

Fig. 11 provides a multi-dimensional comparison of the three APR categories across five key quality attributes, scored on a normalized 1–5 scale based on our analysis.

Despite significant progress, several fundamental challenges remain that represent important directions for the research community:

- Patch Correctness Validation: Distinguishing correct patches from plausible but incorrect ones remains fundamentally difficult. The ratio of plausible-to-correct patches across all techniques averages approximately 1.6:1, meaning nearly 40% of generated patches that pass all tests are semantically incorrect. Without formal verification or significantly more comprehensive test suites, this gap is difficult to close. Emerging approaches include using multiple diverse test suites, semantic equivalence checking, and leveraging LLMs as patch validators.
- Multi-file and Multi-hunk Repair: Most approaches target single-line or single-function fixes within a single file. SWE-bench results (Table III) show that even the best agent-based systems resolve only 32% of real-world issues, many of which require coordinated changes across multiple files. This represents a fundamental challenge because the search space grows combinatorially with the number of edit locations.
- Computational Cost and Accessibility: Training large code models requires significant GPU resources (e.g., StarCoder [32] was trained on 512 GPUs), creating a barrier for academic research groups. While API-based approaches democratize access, they introduce latency, cost, and privacy concerns. Edge deployment of smaller, specialized models remains an active research area.
- Benchmark Quality and Representativeness: Recent studies have identified significant issues with vulnerability detection benchmarks, including mislabeled samples (up to 30% label noise), high duplication rates, and bias toward specific vulnerability types. These issues inflate reported performance and may not reflect real-world detection capability.
- Hallucination and Semantic Correctness: LLMs can generate syntactically valid and test-passing patches that introduce subtle semantic errors—for instance, changing an error message string that coincidentally makes a string-comparison test pass while masking the underlying bug. These "hallucinated" patches are particularly dangerous because they erode developer trust in automated repair systems.
- Explainability: Current AI repair systems operate as black boxes, generating patches without explaining the reasoning behind the fix. For developers to trust and adopt these tools, future systems must provide explanations of why a bug occurred and how the proposed patch addresses the root cause.

VII. Threats to Validity

A. Selection Bias

Our survey covers 32 papers selected from top-tier venues with citation thresholds. This may exclude recent high-quality work from workshops or lower-tier venues, and may over-represent techniques evaluated on popular benchmarks (Defects4J, SWE-bench), while under-representing approaches targeting other languages or domains.

B. Benchmark Limitations

Our comparative analysis relies on results reported in the original papers, which may use different experimental configurations, hardware, and versions of shared benchmarks. Direct comparisons across papers should be interpreted with caution, as subtle differences in fault localization, patch validation, and timeout settings can significantly affect reported numbers.

C. Recency Bias

The rapid pace of LLM development means that some results may already be superseded by the time of publication. We mitigated this by including papers up to 2025, but the field continues to evolve rapidly.

D. Generalizability

Most surveyed techniques are evaluated on Java (Defects4J) or Python (SWE-bench) codebases. Their effectiveness on other programming languages, paradigms (functional, concurrent), or application domains (embedded, real-time) remains largely unvalidated.

VIII. Conclusion and Future Work

This study presented a comprehensive survey and comparative analysis of AI techniques for software error detection and repair. We systematically reviewed 32 key works spanning from 2012 to 2025, categorizing them across three generations of automated program repair (search-based, learning-based, and LLM-based) and multiple categories of bug detection (name-based, sequence-based, graph-based, and transformer-based). Our evaluation used established benchmarks, including Defects4J and SWE-bench.

Our key findings can be summarized as follows:

- LLM-based APR dominates: ChatRepair achieves 114 correct fixes on Defects4J v1.2, fixing 2.6× as many bugs as the best template-based method (TBar: 43) and over 20× as many as early search-based methods (GenProg: 5).
- Real-world gaps remain: SWE-bench performance (best: 32%) reveals that isolated benchmark success does not translate directly to practical software maintenance capability.
- Detection approaches are complementary: No single detection technique dominates across all vulnerability types. Transformer-based methods (LineVul: 91% F1) lead on curated datasets, but specialized approaches remain valuable for specific bug categories.
- Economics favor LLMs: API-based repair at $0.42 per fix fundamentally changes the cost-benefit analysis of automated repair adoption.
- Correctness validation remains the core challenge: Approximately 40% of test-passing patches are semantically incorrect across all technique categories.

A. Future Directions

Based on our analysis, we identify the following promising research directions:

- Agent-based repair systems: Autonomous agents (e.g., RepairAgent [20]) that combine planning, tool use, and iterative refinement represent the most promising direction for complex, multi-file repairs. These systems can navigate codebases, read documentation, run tests, and refine patches iteratively—mimicking the workflow of experienced developers.
- Hybrid detection-repair pipelines: Current systems typically treat detection and repair as separate tasks. End-to-end systems that first localize bugs using GNN-based detection and then apply targeted LLM-based repair could achieve higher accuracy by leveraging the complementary strengths of both approaches.
- Formal verification integration: Connecting LLM-generated patches with lightweight formal methods (e.g., bounded model checking, symbolic execution) could address the correctness validation challenge. A practical approach might use formal methods to filter obviously incorrect patches before test execution.
- Multilingual and cross-platform repair: Most current techniques focus on Java (Defects4J) and Python (SWE-bench). Extending AI repair to languages with different paradigms (Rust, Haskell, Go) and domains (embedded systems, mobile applications) requires new benchmarks and potentially new architectures.
- Developer-in-the-loop systems: Interactive repair systems that present ranked candidate patches with explanations, allowing developers to guide the repair process. This approach combines AI efficiency with human judgment and builds developer trust through transparency.
- Continuous learning from developer feedback: Repair systems that learn from accepted and rejected patches in production environments could continuously improve their accuracy for organization-specific codebases and coding conventions.

B. Implications for Practice

For software development organizations considering AI-based error detection and repair tools, our analysis suggests a practical adoption strategy: begin with LLM-based API tools for simple, single-file bug fixes where the cost-benefit ratio is most favorable; complement with specialized detection tools (e.g., LineVul for vulnerability scanning) in CI/CD pipelines; and invest in comprehensive test suites as the most impactful enabler for all automated repair approaches. As agent-based systems mature, they will increasingly handle the complex multi-file issues that currently require human intervention.

REFERENCES

[1] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 54–72, 2012.

[2] M. Chen, J. Tworek, H. Jun, Q. Yuan *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[3] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "Tbar: Revisiting template-based automated program repair," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2019, pp. 31–42.

[4] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 2016, pp. 298–312.

[5] J. Xuan, M. Martinez, F. Demarco, M. Clement, S. Marcote, T. Durieux, D. Le Berre, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in java programs," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 34–55, 2017.

[6] Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1943–1959, 2019.

[7] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "Coconut: Combining context-aware neural translation models using ensemble for program repair," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2020, pp. 101–114.

[8] N. Jiang, T. Lutellier, and L. Tan, "Cure: Code-aware neural machine translation for automatic program repair," in *Proceedings of the 43rd International Conference on Software Engineering (ICSE)*. IEEE/ACM, 2021, pp. 1161–1173.

[9] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*. ACL, 2020, pp. 1536–1547.

[10] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair via conversation: Fixing 162 out of 337 bugs for $0.42 each using chatgpt," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2024, pp. 1–13.

[11] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun, "Deep learning for just-in-time defect prediction," *IEEE Transactions on Software Engineering*, 2015.

[12] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 32, 2019.

[13] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, "Graphcodebert: Pre-training code representations with data flow," in *Proceedings of the 9th International Conference on Learning Representations (ICLR)*, 2021.

[14] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Proceedings of the 38th International Conference on Software Engineering (ICSE)*. ACM, 2016, pp. 694–705.

[15] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2018, pp. 298–309.

[16] Y. Li, S. Wang, and T. N. Nguyen, "Dlfix: Context-based code transformation learning for automated program repair," in *Proceedings of the 42nd International Conference on Software Engineering (ICSE)*. ACM, 2020, pp. 602–614.

[17] J. Bader, A. Scott, M. Pradel, and S. Chandra, "Getafix: Learning to fix bugs automatically," in *Proceedings of the ACM on Programming Languages (OOPSLA)*, vol. 3. ACM, 2019, pp. 1–27.

[18] C. S. Xia and L. Zhang, "Less training, more repairing please: Revisiting automated program repair via zero-shot learning," in *Proceedings of the 45th International Conference on Software Engineering (ICSE)*. IEEE/ACM, 2023, pp. 1–13.

[19] X. Chen, M. Lin, N. Schärli, and D. Zhou, "Teaching large language models to self-debug," in *Proceedings of the 12th International Conference on Learning Representations (ICLR)*, 2024.

[20] I. Bouzenia, P. Devanbu, and M. Pradel, "Repairagent: An autonomous, llm-based agent for program repair," in *Proceedings of the 47th International Conference on Software Engineering (ICSE)*. IEEE/ACM, 2025.

[21] M. Fu, C. Tantithamthavorn, T. Le, V. Nguyen, and D. Phung, "Vulrepair: A t5-based automated software vulnerability repair," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2022, pp. 935–947.

[22] H. Ye, M. Martinez, and M. Monperrus, "Neural program repair with execution-based backpropagation," in *Proceedings of the 44th International Conference on Software Engineering (ICSE)*. ACM, 2022, pp. 1–12.

[23] M. Pradel and K. Sen, "Deepbugs: A learning approach to name-based bug detection," *Proceedings of the ACM on Programming Languages (OOPSLA)*, vol. 2, pp. 1–25, 2018.

[24] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2018.

[25] M. Fu and C. Tantithamthavorn, "Linevul: A transformer-based line-level vulnerability prediction," in *Proceedings of the 19th International Conference on Mining Software Repositories (MSR)*. ACM, 2022, pp. 608–620.

[26] J. Nam and S. Kim, "Transfer defect learning," in *Proceedings of the 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 382–391.

[27] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE Transactions on Software Engineering*, vol. 45, no. 1, pp. 34–67, 2019.

[28] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," *ACM Transactions on Software Engineering and Methodology*, 2024.

[29] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 30, 2017.

[30] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 23rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2014, pp. 437–440.

[31] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan, "Swe-bench: Can language models resolve real-world github issues?" in *Proceedings of the 12th International Conference on Learning Representations (ICLR)*, 2024.

[32] R. Li, L. B. Allal, Y. Zi *et al.*, "Starcoder: may the source be with you!" *Transactions on Machine Learning Research*, 2023.