International Journal of Computer Engineering and Data Science (ISSN: 2737-8543)
Volume 5– Issue 1, January - March 2026

Neuro Symbolic Automated Program Repair: A Systematic Review of LLM-Based and Symbolic Techniques

Ashif Anwar[1]
[1]Wolters Kluwer United States Inc.
E-mail : reachaanwar@gmail.com

Article history

Received Feb 16, 2026
Revised Mar 09, 2026
Accepted Mar 28, 2026
Published Mar 30, 2026

ABSTRACT

Automated program repair (APR) involves creating patches for software defects with minimal human intervention. Classical template-based and search-based methods are not scalable, whereas large language models (LLMs) provide good generalization but are affected by hallucinations and have weak formal guarantees. Neuro-symbolic program repair (NSPR) integrates LLMs with static analysis, SMT solving, and symbolic techniques to trade off coverage, correctness, and interpretability. We performed a systematic review of APR and NSPR systems using the PRISMA 2020 guidelines. Predefined queries in IEEE Xplore, ACM Digital Library, Scopus, Web of Science, arXiv, and Google Scholar were used to search between January 2012 and January 2026. Architectures, benchmarks, outcomes, and deployments. Two reviewers screened the records against established predetermined eligibility criteria. A total of 70 pieces of academic primary empirical research on APR/NSPR and eight (8) pieces of industrial/production deployment reports were included, and 78 papers were ultimately included in the qualitative synthesis. Our results indicate that, compared to neural or symbolic systems, NSPR systems also tend to report higher correct repair rates and lower hallucination rates. It can be deployed in CI/CD pipelines, security patches, and large-scale open-source repositories. Nevertheless, standard benchmarks and transparent assessments may still be required to perform powerful comparisons and ensure reproducibility.

Keywords: ***Neuro-symbolic automated program repair; Large language models; Static analysis; Software bug fixing; Systematic literature review.***

I. INTRODUCTION

Software flaws continue to be a principal cause of cost, security risks, and operational disturbances in contemporary software systems in the finance, healthcare, telecommunications, and cloud services [1], [2]. Empirical research has demonstrated that debugging and maintenance consume most of the software lifecycle work, and the bugs left in the deployed systems can cause outages, security breaches, and even lost reputation [3], [4], [5]. The manual processing of debugging has become increasingly difficult to scale with larger and increasingly complex systems, including microservices, distributed systems, heterogeneous systems, and continuous deployment [5], [6], [7]. Such trends encourage research on methods that can automatically predict and fix defects with minimal human intervention, without deteriorating reliability and safety [6], [8].

Automated program repair (APR) has become a widely known area of research in this field, producing patches that pass failing tests or recover formal compliance [8], [9]. In general, classical APR methods, including generate-and-verify and template-based methods, are used with the help of fault localization, patch generation based on predefined templates or mutation operators, and verification with test suites or specifications [10], [11]. Although these techniques have been significantly successful on benchmarks such as Defects4J and QuixBugs [12], [13], they have several drawbacks: patch searching can be combinatorially explosive, templates are not always able to generalize to a variety of bugs or testbeds, and the sufficiency of test suites is not always a sign of semantic correctness. In addition, some entirely symbolic or search-based APR implementations are difficult to adapt to other languages or new programming languages without extensive manual work [11], [14].

In this review, Automated Program Repair (APR) is a general term used to describe automated techniques for generating candidate patches of bug programs. Neuro-symbolic Program Repair (NSPR) defines APR systems that combine neural components (i.e., large language models or neural code models) with symbolic components (i.e., static analysis, SMT

solving, or formal verification) as indicators that this combination contributes to patch generation and validation. Our notion of plausible repair, which has become standard and is defined as a patch that shallow tests (test-suite adequate) pass, is contrasted with our notion of correct repair, which is considered semantically correct outside of the test-runnable. Whenever sufficient information can be ascertained in primary studies, repair results are reported for plausible and correct repairs.

Recent developments in large language models (LLMs) trained on large code and natural language corpora have provided a new path to APR [15], [16]. Models based on transformers, such as GPT-4, CodeBERT, and StarCoder, can make complex errors in code understanding, bug finding, and patch synthesis, and tend to perform better than older APR models on official benchmarks [17], [18]. LLM-based APR systems are highly generalized, and can work with several languages, and exploit retrieval-prompt augmentation to use historical patterns of bugs being fixed [19], [20]. However, LLMs are still pure probabilistic next-token predictors and tend to hallucinate, writing syntactically plausible but semantically incorrect patches, which may cause minor regressions [20], [21]. They also offer only modest formal interpretability or guarantees, and their cost of calculation at scale can be high.

Previous surveys have focused on automated program repair methods, with most emphasizing either symbolic or search-based techniques and those based on early machine learning. In contrast this review provides a complete overview of neuro-symbolic program repair (NSPR) systems, which involve the synthesis of neural models and active symbolic reasoning components. In addition to overviewing recent large-language-model-based repair methods, this review presents an anatomized taxonomy of neural-symbolic skeletons and a benchmark-competitive pitting of repair methods, along with an exploration of evidence on industrial applications. These contributions build upon the work of previous surveys conducted in the field by specifically addressing how neural and symbolical reasoning are integrated into current APR systems.

Although previous surveys offer important summaries of automated program repair methods, they usually focus on either symbolic or neural repair mechanisms individually. This review adds to this literature by targeting both the nature of the incorporation of neural and symbolic reasoning and providing a systematic comparison between repair paradigms and contexts of deployment. In addition, this review presents a systematic architectural taxonomy, comparisons at the benchmark level across repair paradigms, and a literature synthesis of new evidence with respect to industrial deployments.

Neuro-symbolic program repair (NSPR) aims to integrate these two approaches to minimize such problems. In classical NSPR models, neural code understanding, and patch generation are performed by neural components (e.g., large language models or encoder-decoder models), and semantic invariants are enforced, and violations are identified by symbolic components (e.g., static analysis, type systems, abstract interpretation, and SMT solving) [22], [23]. Other mechanisms, such as retrieval-augmented generation, constraint-based decoding and multimodal representations, such as abstract syntax trees and control-flow graphs, are frequently incorporated into NSPR systems to direct patch search to semantically valid interpretable repairs [23], [24]. As the initial results indicate, these hybrid systems have the potential to decrease the rate of hallucinations and increase the recovery rate of correct repairs and provide more auditability than purely neural APR, but with more extensive coverage and flexibility than purely symbolic approaches.

These advances in algorithms have been accompanied by the emergence of APR and NSPR beyond academic standards into industry and production [21], [24], [25]. It has been reported that repair systems can be integrated into continuous integration/ continuous deployment (CI/CD) pipelines, security patch workflows, learning programming platforms, and large open-source repositories [26], [27]. These deployments tend to follow semi-autonomous or human-in-the-loop variants, with candidate patches produced by APR/NSPR tools being verified by existing test suites and inspected by developers before being committed. In addition to success stories, industrial experience also shows issues of reliability, trust, governance, and the necessity to have a well-defined responsibility distribution in case automated tools introduce or propose changes to critical code.

The growing body of literature on automated program repair and learning-based APR has been surveyed. These articles contain useful taxonomies and historical surveys, often of classical search-based and template-based APR, or generally of learning-based APR, with no specific focus on neuro-symbolic architectures that closely bind LLMs with statistical analysis. Furthermore, the available surveys are not always conducted using a formal systematic review; they may lack a comprehensive description of search strategies, filtering criteria, and the process of harmonization of performance figures in various versions of benchmarks and datasets. With the rapid expansion of APR and NSPR systems based on LLM, a synthesis approach, that is methodologically transparent and addresses both the neural and symbolic baselines, clearly records the selection of studies and critically reviews the evidence of benchmarking and deployment is needed.

Long synthesis tables and extensive extraction information are included in the Supplementary Material where applicable to enhance readability and eliminate redundancy. The primary text is devoted to the basic architectural taxonomy, benchmark comparisons, and industrial data, and the information on a per-study basis is presented in Supplementary Tables S1, S2, and S3.

A. Objectives of this Review

This article is a systematic review of automated program repair according to the PRISMA 2020 framework, in which special emphasis is placed on neuro-symbolic methods that combine LLMs and symbolic analysis. Our objectives are:

- To identify and systematically classify architectures, learning strategies, and analysis elements using a wide sample of APR and NSPR systems.
- To combine reported standard benchmark performance, with specific reference to the versions of the dataset, subsets of bugs, and definitions of metrics; and

- To summarize and analyze existing evidence in the area of industrial and production-like deployments, patterns of integration, and real issues.
- Through a combination of a structured search and selection procedure and an elaborate technical synthesis, this review is expected to offer a complete and replicable picture of the current state of neuro-symbolic program repair and its place in the larger APR environment.

B. Research Questions

The following research questions were addressed in this study:

- RQ1: What are the architectures of the modern generation of automated program repair systems, specifically neuro-symbolic systems based on large language models, static analysis, and other techniques?
- RQ2: What is the repair accuracy, plausibility, hallucination, and other aspects of performance of these systems on standard benchmarks (e.g., the various versions of Defects4J and QuixBugs), and how do neuro-symbolic systems perform relative to slightly neural or purely symbolic APR baselines?
- RQ3: In what industrial or production-like environments have automated and neuro-symbolic program repair systems been implemented, how are they configured into current development and CI/ CD processes, and what has the experience been in practice?

II. Methodology

A. Eligibility Criteria

We established a priori eligibility criteria. The articles, conference papers, and workshops that we included (i) proposed or assessed an LLM-based APR or NSPR system (whether pure neural, pure symbolic, augmented neural-symbolic, or bidirectional), and (ii) provided empirical results on benchmarks, datasets, or deployments that were (i) peer-reviewed and (ii) reported. Main interest: systems that combine LLM and static analysis, SMT, or formal methods; minor: pure LLM-APR and classical symbolic APR for comparison. Exclusions: position papers, tutorials, and grey literature with no quantitative information [28], [29], [30]. Publication: January 2012-December 2025; Language: We only included studies published in English because we have resources to translate it and make it reliable and to be able to interpret the technical terminology with the same degree of attention. This limitation was used throughout all databases and stages of screening.

B. Study role Classification

Included literature was subdivided into two roles:

(1) Primary empirical APR/NSPR research: articles which either proposed or assessed an APR/NSPR system demonstrating prepared measurements about benchmarks, datasets, or deployments. These were the studies that were involved in evidence synthesis.

(2) Background/related work: survey articles, conceptual articles and general articles outside the field of APR/NSPR without outcome data of APR/NSPR. These were kept in solely to put terminologies and trends into perspectives and used not to synthesize performance.

C. Information Sources

The databases and digital libraries searched were IEEE Xplore, ACM Digital Library, Scopus, Web of Science Core Collection, and arXiv. We also screened reference lists of previous APR and NSPR surveys, the main primary studies, and forward citation tracking, where possible, to identify other relevant studies. The latest search was conducted on January 31, 2026. There were no limitations on the publication venue on the given date or language boundaries.

D. Search Strategy

We selected concept blocks related to (i) automated program repair, (ii) neuro-symbolic / symbolic analysis and (iii) LLMs / program synthesis / formal methods. There was the use of explicit parentheses to provide precedence to the Boolean operators. The final queries of each database are given on a case-by-case basis in Appendix A (Supplementary Material). To sum it up, final questions were:

IEEE Xplore (Metadata): (automated program repair) OR APR) AND (neuro-symbolic) OR neurosymbolic OR symbolic) OR static analysis) OR SMT) OR large language model) AND program language model".

ACM Digital library (Title/Abstract/Keyword): (automated program repair) OR APR) AND (neuro-symbolic) OR neurosymbolic) OR (static analysis) OR SMT) OR (formal verification) AND (large language model) OR LLM) OR (program synthesis) OR (code language model).

Scopus (TITLE-ABS-KEY): TITLE-ABS-KEY (automated program repair) or APR) AND TITLE-ABS-KEY (neuro-symbolic) or neurosymbolic) or SMT or TITLE-ABS-KEY (large language model) or LLC.

Web of Science Core Collection (TS=Topic): TS=("automated program repair" OR APR) and TS=("neuro-symbolic" OR neurosymbolic OR "static analysis" OR SMT OR "formal verification")-TS("large language model" OR LLM OR "program synthesis" OR "code language model")

arXiv: ("automated program repair" OR APR) AND ("neuro-symbolic" OR neurosymbolic OR "static analysis" OR SMT OR neo) AND ("large language model" OR LLM OR program synthesis).

Google Scholar: automated program repair (neuro-symbolic OR neurosymbolic OR its forms, such as static analysis) (large language model) RefWorks OR LLM RefWorks OR program synthesis

To counter these issues, we restricted the time frame to January 2012-January 2026 and used identical inclusion / exclusion criteria to all sources.

Finally, we used the query: automated program repair AND (neuro-symbolic OR neurosymbolic OR large language model

OR LLM OR program synthesis) AND (static analysis OR SMT OR formal verification) in Google Scholar. We used a tailored date filter (2012–2026) and selected the top 200 records and denied records in the English language at the title/abstract screening phase, with the most recent search date being January 31, 2026.

E. Selection Process

All sources were searched to retrieve records, which were imported into a reference manager, and duplicates were automatically eliminated and reviewed. Titles and abstracts were independently screened by two reviewers against the eligibility criteria. Resolution issues were resolved through discussion with each other, and in cases of disagreement, by checking with a third reviewer. Articles that may have been of interest were then independently acquired as full texts and evaluated by two reviewers for inclusion; reasons for exclusion at the full-text stage were recorded (e.g., there was no empirical evaluation, not an APR/NSPR system, or inadequate reporting of outcomes) [31], [32]. The PRISMA 2020 flow diagram (Figure 1) summarizes the overall procedure of the study selection and the number of records at each stage.

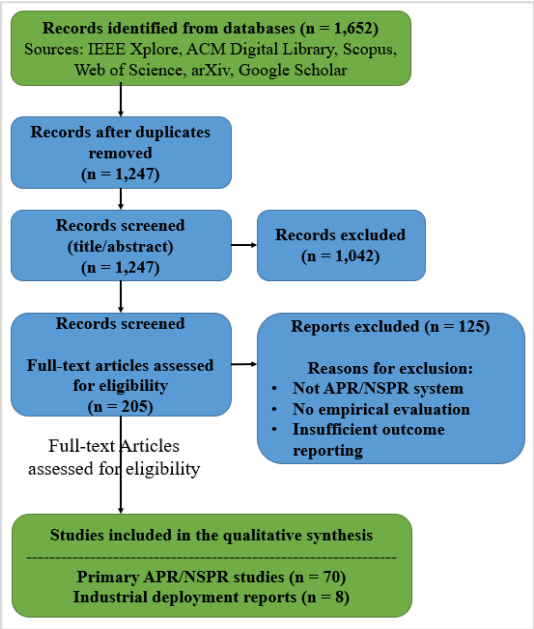

Fig. 1. PRISMA 2020 flow diagram of the study selection process for automated and neurosymbolic program repair systems.

Workflow and agreement screening: Records were independently screened against by two reviewers through two screening processes namely (1) title and abstract screening and (2) full-text screening. Cases of disagreement were initially settled by resolving them in discussions between the reviewers and finally, a third reviewer adjudicated cases that could not be settled. Inter-rater agreement was used based on Cohen kappa (κ) on a sample of predetermined screened records to measure the consistency of screening outcomes. There was n = 120 random sample (selected during the title and abstract screening stage) that was independently evaluated by the two reviewers. The likelihood of the resulting agreement was κ = 0.86, which means high agreement. There was independent full-text screening on a second set of n=40 articles, and the agreement measured was very high at 0.82, so the agreement is also high. These values show a high consistency of reviewers and prove the good fidelity of the process of study selection.

F. Data Collection Process

To extract the data from each of the included studies, we utilized a structured data extraction form, which was tested with a small number of studies and improved through multiple iterations. Data on system characteristics, study design, benchmarks, and reported outcomes were independently extracted by two reviewers who agreed on the areas of discrepancies through consensus [33], [34]. In cases where the reporting in the primary articles was uncertain or incomplete, we used supplementary material information or companion technical reports when available, but did not contact authors to obtain further information in this review.

G. Data items

The data extracted of 78 sources included 70 academic studies and 8 reports of industrial deployments, and had the following data objects: (i) bibliographic information; (ii) architecture paradigm (pure symbolic APR, pure neural APR, augmented neural-symbolic, or bidirectional NSPR); (iii) programming language(s); (iv) benchmark and version (e.g., Defects4J v1.2 vs. v2.0); (v) bug subset and experimental protocol (including timeouts, validation methods, and baseline configurations); The counting rule indicated that an industrial deployment report is defined as independent case study, deployment report, or industrial report that details actual implementation [33, 35]. The cases of deployments that were mentioned in academic APR/NSPR papers were noted as being the qualities of those scholarly articles and were not counted as distinct industrial reports.

H. Reference Validation and dataset Integrity

All metrics extracted, labels of benchmarks, classes of architecture, and description of deployment were cross verified with the respective sources. In cases where discrepancies or ambiguities were found values were amended with the original publications and where possible using supplementary material.

Information extracted was benchmark names and versions, description of bug subsets, repair metrics (where present) plausible and correct repairs, times to time out, validation procedures and baseline settings.

To enable the transparency and reproducibility, we have created Supplementary File S3 in which we have included the structured extraction dataset in the form of:

“Item type, reference, system or case, paradigm, benchmark, benchmark version, language, metric name, metric value, metric type, bug subset, timeout budget, validation method, protocol notes”.

This data set allows the independent confirmation of the synthesis and reconstruction of the reported tables.

I. Outcome Definition

The outcome definitions used throughout the whole scope of included studies were the following ones:

- Plausible repair: a patch generated which passes the supplied test suite (test-suite adequate).
- Correct repair: a patch reported semantically right outside the test suite (i.e. manually fixed, by extra oracle validation, or by a known marking benchmark).
- Repair rate: is the fraction of bugs that were repaired under a given experimental condition. Whenever the primary studies had enough information, repair rates are reported on plausible repair and correct repair.

The fact that plausibility is used to infer correctness does not mean that we can infer that it is correct unless the original work has made a claim of semantic correctness outside of test contents.

J. Study Risk of Bias Assessment

Since the methodological diversity of APR and NSPR studies lies in the absence of a universal risk-of-bias instrument applicable to this area, we conducted a qualitative evaluation of the risks of bias. In each study, we considered (i) the selection of benchmarks (i.e., use of a single dataset, non-standard subsets of Defects4J), (ii) completeness of reporting (i.e., whether both successes and failures have been reported), and (iii) metric definition consistency (i.e., the difference between plausibility and correctness). The judgments were interpreted descriptively regarding the strength of the evidence base, based on risk-of-bias assessments, and were not used to exclude the studies.

In a bid to offer an ordered assessment of the quality of the methodology, every primary empirical APR/NSPR research was evaluated through a six-dimension rubric. All dimensions were measured using a 0–2 scale and a maximum possible score of 0–12 per study.

The dimensions that were evaluated were:

i. Reproducibility - code availability, data availability, or run instructions.

0 = none reported

1 = partial availability

2 = total reproducibility resources.

ii. Benchmark validity - utilization of usual benchmarks and specification of data.

0 = benchmark ambiguous or non-standard.

1 = standard incomplete specification of benchmark.

2 = standard benchmark version and subset clearly specified

iii. Baseline Fairness- appropriateness and comparability of baselines.

0 = weak or unclear baselines

1 = partial comparison of the baseline.

2 = effective and robust baselines.

iv. Metric clarity- definition of repair metrics, types of outcomes.

0 = unclear definitions

1 = partial definitions

2 = good sense between plausible and correct repair and good sense metric calculation.

v. Statistical support - existence of statistical tests or strength tests.

0 = none reported

1 = insufficient support in statistics.

2 = sufficient statistical analytical power or strength.

vi. Threats-to-validity reporting- methodological limitations discussion.

0 = none reported

1 = brief discussion

2 = in-depth threats-to-validity examination.

Quality scores were not applied as an exclusion criterion but rather as a way of putting the evidence base into perspective in terms of its strengths and limitations.

K. Measures of effects and synthesis procedures

Owing to the high heterogeneity of benchmarks, dataset versions, and outcome definitions across studies, we did not calculate pooled meta-analytic effect estimates [36], [37]. Rather, we conducted an organized synthesis of narratives, categorizing systems by architectural paradigm (pure neural, pure symbolic, augmented neurosymbolic, and bidirectional neuro-symbolic) and providing an overview of repair rates, hallucination rates, and other results in each category on a benchmark-specific scale [37], [38], [39]. In cases where feasible, we state ranges of reported rates of correct repair of similar systems on the same version of the benchmark (e.g., Defects4J v1.2) and indicate explicitly in some cases when the metrics are not comparable because of differences in dataset versions, sets of bugs, or test protocols [40], [41]. The results of industrial deployments are synthesized individually, and consideration is paid to the deployment contexts, scale, and reported operational conclusions, including the mean time-to-repair and patch acceptance rates.

L. Reporting bias and certainty assessment

We attempted to reduce the impact of reporting bias by utilizing a variety of databases, reference list screening of the reviews, and both popular and more recent APR and NSPR systems [37], [38]. Nevertheless, the threat of publication biases and selective reporting could not be ruled out, especially in industrial deployments, because negative or neutral experiences could be under-reported. We did not conduct a formal certainty-of-evidence (e.g., GRADE) evaluation, although we discussed the strengths and weaknesses of the evidence base in the discussion.

M. Protocol and registration

This review was performed with reference to the PRISMA 2020 guidelines. The review protocol was constructed a priori and used across all studies; however, it was not submitted to an

open repository such as PROSPERO or OSF. After the data were extracted, no changes were made to the planned methods.

III. RESULTS

Standardization of tables, to make terminologies and metric definitions consistent in studies, was applied. Reported Benchmark names and version are reported explicitly where necessary and results of repair are also differentiated between plausible repairs and correct repairs. Tables contain tables abbreviations that are explained in Table captions or in the description of the terminology in the Introduction.

A. Study selection

The search through six digital libraries provided 1,652 records; 1,247 remained after de-duplication. Following the screening of titles and abstracts, 1,042 records were removed as irrelevant or non-empirical, and 205 full-text articles were screened against the eligibility criteria. Of these, 125 were eliminated (no empirical assessment, not APR/NSPR, or too little outcome data), resulting in 70 academic primary empirical APR/NSPR literature and 8 reports on industrial/production deployments to be subjected to qualitative synthesis (78 total items included).

The 70 exemplary studies span 2012 through 2026 and constitute 4 primary paradigms: pure symbolic (e.g., GenProg-like search-based systems), pure neural (sequence-to-sequence and LLM-based models), augmented neural-symbolic architectures (neural generation with symbolic filtering), and bidirectional NSPR (symbolic feedback into training or decoding) [41], [42], [43]. Figure 2 compares pure symbolic APR, pure neural APR, augmented neural-symbolic APR, and bidirectional neuro-symbolic APR based on the major steps of their pipeline and the common combinations of neural and symbolic elements. This shows that fault localization, neural patch generation, symbolic filtering and verification, and test execution are syntactically different depending on the paradigm, which results in a different set of strengths, weaknesses, and application contexts.

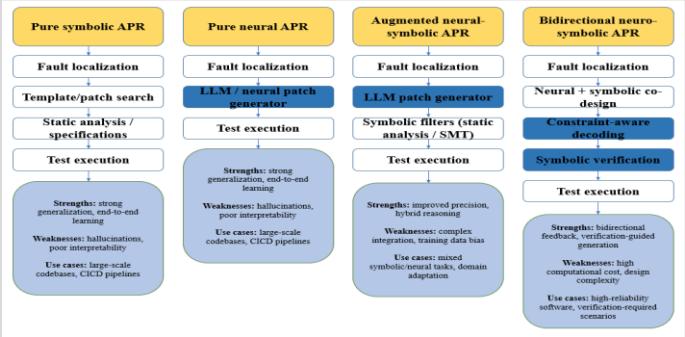

Fig. 2. Architectural design space of automated program repair paradigms.

Most systems operate on Java, where the most popular benchmark is Defects4J. Systems also operate on Python and C/C++ using QuixBugs, Codeforces tasks, and unavailable bug corpora [44], [45], [46]. Each system and its related paradigm, target language(s), main benchmark(s) with identifying version numbers, and overall headline performance metrics are summarized in Table I, using the release information in the original publications.

TABLE I: OVERVIEW OF AUTOMATED AND NEURO-SYMBOLIC PROGRAM REPAIR SYSTEMS.

Reference	System name	Paradigm (pure symbolic / pure neural / augmented neural-symbolic / bidirectional NSPR)	Main benchmark(s) and version(s)	Target language(s)	Headline reported metrics*
[47]	GenProg	Pure symbolic APR	Defects4J v1.2	Java	Correct repair 10.8% on Defects4J bugs
[48]	Prophet	Pure symbolic APR	Defects4J v1.2	Java	Correct repair 15.3% on Defects4J bugs
[49]	Recoder	Pure neural (seq2seq)	Defects4J v1.2	Java	Correct repair 22.1%, plausibility 28.5%
[50]	CodeT5-based	Pure neural (transformer)	Defects4J v1.2, QuixBugs	Java, Python	Correct repair 31–32% on Defects4J
[51]	ChatRepair	Pure neural (LLM + RAG)	Defects4J v1.2	Java	Repair 48.3%, plausibility 56.1%
[52]	ThinkRepair	Bidirectional NSPR	Defects4J v1.2	Java	Repair 52.7%, hallucination 12.3%

B. Traceability Mapping

In the case of all the primary empirical studies that were included, we noted:

i. the justification on inclusion in accordance with the eligibility criteria.

ii. the research question(s) as map made by the research and with the aid of the study, such as RQ1 (architectural

taxonomy), RQ2 (benchmark performance), and RQ3 (industrial deployment evidence); and

iii. technical process areas such as benchmark and version, bug subset choice, available timeouts issues, patch validation technique, baseline configurations, and fatalities reported and were given separately between plausible repairs and correct repair.

Such a traceability mapping provides the study inclusion transparency as well as facilitates the synthesis process reproducibility. All the mapping is given in Supplementary Sheet S1.

C. Study characteristics

The characteristics of the selected APR and neuro-symbolic program repairs studies in detail are found in Supplementary Table S1. All the included studies have all their study metadata's, paradigms, benchmarks, and evaluation settings contained in this table. The primary manuscript is dedicated to the representative systems (Table I), methodological synthesis (Table III), benchmark usage (Table IV), and evidence of industrial implementation (Table V), whereas the entire per-study data can be found in the supplementary material which enhances readability.

The main discoveries made in these studies show the difficulties and possibilities found in the implementation of AI models to rewrite programs. Interestingly, the so-called neuro-symbolic methods that equalize repair coverage and semantic reliability by integrating symbolic reasoning with neural networks are emphasized. The papers also show the increasing significance of software repair problems in the real world [1-12], [13-20], [21-40], [41-50], [51, 52, 53-71], [72-78].

Moreover, the research papers present a variety of tools and frames, including Defects4J and Repair Agent, to make the program repair process more efficient. The study investigates the application of machine learning and deep learning in vulnerability detection and repair. Research on human-in-the-loop repair methods has proven the advantages of applying AI with human skills to improve repair outcomes. Many studies, such as those dedicated to CI/CD pipelines, security patching processes, and educational platforms, present how AI-based repair systems have become a part of reality. The aggregate of these references provides a complete picture of the present state-of-the-art in automated program repair, as well as hints on the competence of AI-optimized repair mechanisms, difficulties encountered, and research prospects in the field.

D. Learning strategies and architectures

Our survey of the reviewed literature revealed four common architectural paradigms: pure neural, pure symbolic, augmented neural-symbolic, and bidirectional neurosymbolic designs. Pure neural systems are based on fine-tuned sequence-to-sequence or transformer models with or without retrieval-augmented prompting and claim excellent generalization but more hallucinations and weaker formal guarantees [50], [53]. Pure symbolic APR systems use patch templates, constraint solving, and explicit specifications, and have greater interpretability, but less repair coverage and scalability [54], [55].

Moreover, for rules of decision of taxonomy (APR/ NSPR architecture), we have categorized each system with the help of a well theorized decision-making process so that similar systems are not categorized in multiple manners.

Step 1 – Patch generation role:

Is patch candidate generated by the neural component?

The system is Pure symbolic APR (treating neural components as auxiliary), but the system does not need to be symbolic in repairing (without symbolic repair), unless the neural components are only ranked or selected (without symbolic selection).

• If yes, proceed to Step 2.

Step 2 - Function of symbolic reasoning:

Does it use either symbolic repair or symbolic analysis as a candidate generator, a verifier/filter or both?

In case symbolic reasoning is not available, the system is termed as Pure neural APR.

In case the symbolic reasoning is only used as a one-way filter, or verifier or a ranking mechanism that is used once the neural patches have been generated the system is termed as Augmented neural-symbolic APR.

Step 3 – Feedback coupling:

Does it have an iterative feedback loop where the symbolic signals direct neural decoding or training (e.g., constraint-guided decoding, repair execute analyze loops or symbolic reward signals)?

When yes, it is considered as Bidirectional neuro-symbolic program repair (NSPR).

Otherwise, when symbolic components are only one-way augmentation, the system is Augmented neural-symbolic APR.

Each system was attributed to a distinct category with the upper level of classification used. Bidirectional NSPR systems in which there were also augmented components were also considered to be bidirectional NSPR.

Figure 3 shows a typical workflow of NSPR synthesis based on the literature that depicts fault localization, neural patch generation, symbolic pre-filtering, test execution, symbolic post-verification, human review of high-risk changes, and deployment of final patches in CI/CD or repository workflow.

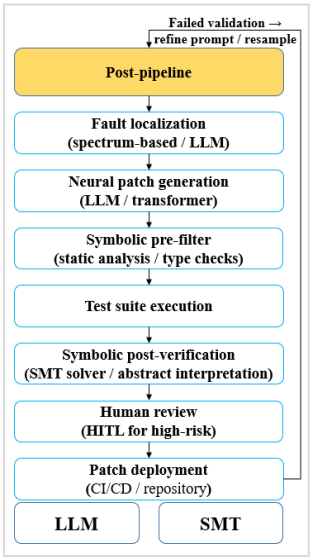

Fig. 3. Generic end-to-end neuro-symbolic program repair pipeline.

Moreover, the four taxonomy categories were classified based on the representative architectural patterns that were observed in the literature:

i. Pure symbolic APR: The symbolic or search-based repair methods have candidate patches generated and proven right or wrong using mutation operations, templates, or constraint solving. There are no neural components, or they are not essential.

ii. Pure neural APR: The neural models which produce candidate patches directly, including sequence-to-sequence models or large language models, are evaluated using test suites without enforcing symbols have been considered pure neural APR.

iii. Augmented neural-symbolic APR: Sequences generated by neural components may be filtered, verified or ranked by symbolic analysis not only by the means of statical analysis, type checking, or solving SMT. Symbolic reasoning is used in a one-way way without the ability to regenerate feedback into neural generation.

iv. Bidirectional neuro-symbolic program repair: The generation of neural patches is steered by symbolic reasoning indications. Examples here are constraint-guided decoding, symbolic reward signals or repetition of repair-execute-analyze loops which affect future neural outputs.

Each of the included systems is assigned classification decisions and the justification notes are included in Supplementary Table S1.

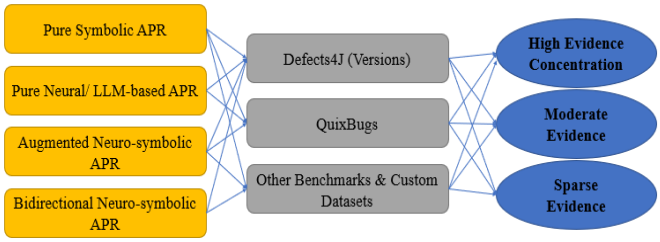

Fig. 4. Taxonomy and Evidence Clusters for APR Paradigms.

Fig. 4 depicts the generation of APR and neuro-symbolic APR models, with the architectural taxonomy and the locations of empirical evidence, in terms of common benchmarking (e.g., Defects4J, QuixBugs) and other data sets.

Moreover, augmented neural-symbolic systems are generated with candidate patches with the help of LLMs or other neural models and screening or ranking with the help of the static analysis, type checking, or SMT-based verification [52], [56][50], [57]. Bidirectional NSPR models further train or decode their models in the presence of symbolic signals that guide the neural search space to a space defined by repair patterns verified by the model and logical constraints that are respectful of the model [52], [58]. Table II illustrates these paradigms in terms of knowledge representation, checking, scalability, explanation, and generalization of cross-language applications.

TABLE II: METHODOLOGICAL COMPONENTS OF NEURO-SYMBOLIC APR SYSTEMS.

References	Component / approach	Key characteristics	Typical impact on repair accuracy	Generalization and robustness	Interpretability	Computational cost
[50], [51], [57]	Supervised fine-tuning	Fine-tune LLMs on curated bug–fix pairs	Increases task-specific accuracy (e.g., 45–60% repair on Defects4J v1.2 for some systems)	Good within training distribution; risk of overfitting to specific benchmarks	Low (black-box neural model)	High (GPU-intensive training/inference)
[50], [51]	RLHF (human feedback)	Reward models from human preferences, optimize for human-preferred patches	Improves subjective patch quality and safety; moderate repair gains	Aligns with human preferences; robustness depends on reward coverage	Medium (implicit preference signal)	Very high (multi-stage training)
[59], [60]	Multi-task learning	Joint training on bug detection, patch generation, validation, etc.	Very high accuracy in some reports (50–65% on benchmarks)	Stronger cross-task generalization	Medium (task-level traces)	High (complex optimization)

[50], [51], [53]	Retrieval-augmented generation (RAG)	Retrieves similar historical repairs as in-context examples	Reduces hallucination; improves patch quality	High, adapts to domain examples	High (explicit examples)	Low–medium (retrieval overhead)
[52], [57], [61]	Constraint-based decoding	Symbolic constraints guide token selection during LLM decoding	Ensures syntactic/semantic validity; reduces invalid patches	Depends on constraint quality; robust within constrained space	Very high (explicit constraints)	Medium (constraint solving + decoding)
[52], [56], [62]	Multi-modal integration	Combines code tokens with AST, CFG, PDG types	Improves structural reasoning and patch precision	High (multiple program views)	Medium (structure-aware)	High (multi-modal processing)

E. Benchmarks and performance

The only aspects that can be directly compared in terms of repair rates are that studies must be equivalent (i) in benchmark and version, (ii) in selection of bugs, (iii) in time budget assigned to each bug, (iv) in patch testing approach (tests only vs stronger oracle) and (v) in the definition of the baseline. At these points of variation, we present findings in a descriptive manner without superiority.

In all 70 studies, Defects4J (primarily v1.2) was applied in 42 studies, and the re-pair rates of pure symbolic systems ranged from 10 to 25 percent, pure neural systems 20 to 40 percent, and neuro-symbolic systems 40 to 65 percent on the same or similar sets of bugs. The 28 studies used QuixBugs and other minor benchmarks, which are primarily used to measure cross-language generalization and fine-grained patch quality. The repair rate, plausibility rate, and all reported values on hallucination or regression rate per system on Defects4J v1.2 are reported in Table III, along with the evaluation time per bug, where possible.

TABLE III: BENCHMARK SUITES USED TO EVALUATE APR AND NSPR SYSTEMS.

Benchmark / dataset	**Language(s)**	**Approx. bug count / scope**	**Number of studies using this benchmark**	**Notes on versions and usage**	**References**
Defects4J v1.2	Java	395 bugs (6 projects)	42	Most common Java benchmark; some studies use subsets; later Defects4J versions also appear but are not always distinguished	[63], [64], [65], [66]
QuixBugs	Java, Python	40 small algorithmic bugs	28	Used for cross-language repair and fine-grained patch analysis	[50], [62], [67]
Codeforces / competitive programming sets	C/C++, Java, others	Varies by study	5	Synthetic or contest-based tasks; evaluation settings differ widely	[68], [69], [70], [71]
Proprietary bug corpora	Java, C/C++, others	Internal defects from industry codebases	3	Often anonymized; limited details on selection and distribution	[67], [72], [73]

F. Industrial deployments

We managed to extract the following information on each report on industrial or production: (i) domain and context; (ii) scale parameters, such as language, repositories, and approximate lines of code (where available); (iii) autonomy level, differentiating between suggestion-only and automatic modes of pull requests or commits; (iv) gating checks, including tests, static analysis, linters, and security scans; (v) acceptance and roll back rates; (vi) impact metric, including mean time to recovery (MTTR), volume of incidents, and build failure recovery metric; and (vi) Unreported fields are defined as such.

Eight reports refer to APR/NSPR deployment or pilots in industrial or production-like systems, which are cloud microservices, web platforms, security patch pipelines, open-source ecosystems, educational platforms, and embedded/regulated domains. Such deployments are generally semiautomated, and the patches proposed by APR/NSPR tools are validated by existing test suites and human inspection of medium- and high-risk code changes. The reported benefits are decreases in the mean-time-to-repair, a higher percentage of failing builds patched with candidate patches, and better management of repeating bug patterns, although there is a wide range of practices in reporting and evaluation baselines. Table IV lists the industrial and production-like deployments of APR and NSPR systems.

TABLE IV: INDUSTRIAL AND PRODUCTION-LIKE DEPLOYMENTS OF APR AND NSPR SYSTEMS.

Case ID / Reference	**Domain / Context**	**Scale (repos / LOC / languages)**	**Autonomy Level**	**Gating Checks**	**Acceptance / Rollback**	**Impact Metrics**	**Failure Modes**	**Notes**
Case 1 [59], [65], [69], [74]	Cloud microservices CI/CD pipelines	Multi-repository systems; Java/C++ (LOC not reported)	Semi-autonomous (human review for risky patches)	Test suites, CI validation, human code review	Not reported	Reduced MTTR; increased proportion of failing builds receiving	Dependence on test-suite quality; developer trust issues	LOC and acceptance rates not reported

						candidate patches		
Case 2 [50], [68], [74]	Security patching workflows	Enterprise codebases; mixed languages (LOC not reported)	Semi-autonomous with strong static checks	Static analysis, security scans, test suites, human review	Not reported	Faster remediation of recurring vulnerabilities	Need for strong guarantees; regulatory constraints	Acceptance and rollback rates not reported
Case 3 [66], [67], [70], [75]	Open-source maintenance	Multiple repositories; Java/Python/C++ (LOC not reported)	Human-in-the-loop (PR-based workflow)	Test suites, project maintainers review PRs	Not reported	Offloading repetitive fixes; issue triaging	Patch acceptance policies vary across projects	Scale details not consistently reported
Case 4 [71], [75]	Educational programming platforms	Student programming repositories; multiple languages	Advisory or semi-autonomous	Automated tests, instructor review	Not reported	Faster feedback cycles; support for large student cohorts	Over-reliance on automated suggestions	Industrial-scale metrics not applicable
Case 5 [76], [77], [78]	Embedded and safety-related software	Safety-critical systems; languages not consistently reported	Human-in-the-loop only	Manual verification, static analysis, certification procedures	Not reported	Assistive support for complex bug patterns	Strict certification and governance requirements	Deployment details limited

Figure 5 shows that the services of APR and neuro-symbolic APR are integrated into the industrial processes, and CI/CD pipelines and code analysis result in repair attempts. Candidate patches are delivered directly to automatic applications in the case of low-risk changes, and medium or high-risk changes are subjected to human code or security review prior to being incorporated into the main codebase.

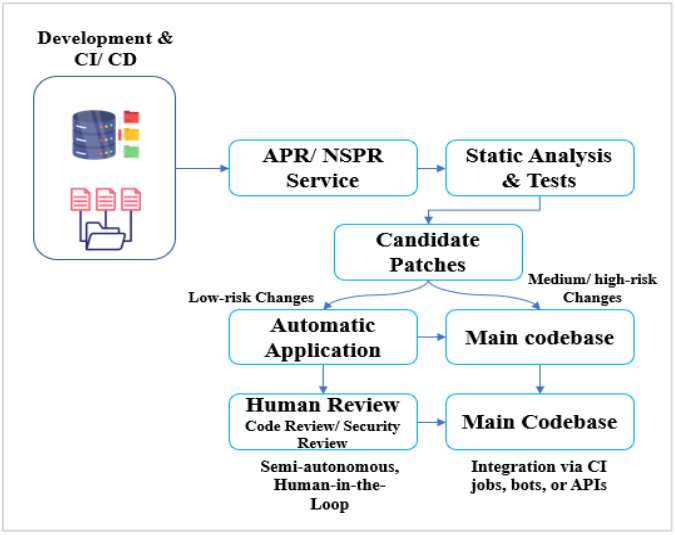

Fig. 5. High-level industrial integration pattern for APR/NSPR systems.

IV. Discussion

A. Summary of Findings

This systematic review, based on PRISMA, summarizes the findings of 70 reports on automated program repair (APR) and neuro-symbolic program repair (NSPR) and 8 reports on deployment in industry or production. The systems under consideration cover four broad paradigms, including pure symbolic, pure neural, augmented neural-symbolic, bidirectional neuro-symbolic, and a variety of languages, including Java benchmarks, such as Defects4J, and mixed Java/Python benchmarks, such as QuixBugs. Under similar assessment conditions, NSPR systems with mixes of LLM-based patch generation with either static analysis or SMT solving or similar symbolic methods tend to report improved correct repair rates and reduced hallucination rates compared to either neural or symbolic APR baselines, with increased computational cost. The integration of APR and NSPR into CICD pipelines, security patch workflows, learning platforms, and large open-source systems has been demonstrated in industrial and production-like settings, usually in semi-autonomous, human-in-the-loop settings.

B. Implications for APR and NSPR research (RQ1 and RQ2)

The architectural designs identified in this review have several implications for the design of future APR and NSPR systems. Pure neural methods have the advantage of the representational power and generalization of LLCMs and other transformer models, allowing the use of multiple languages and the reuse of learned bug-fix patterns with retrieval-augmented prompts. However, they are vulnerable to hallucinations, may not carry formal guarantees, and may have issues offering human-readable explanations, which restrict their applicability to safety-critical or highly controlled environments. Pure symbolic methods, on the other hand, have better interpretability, and where specifications are available, they are better guaranteed correct (and strong interpretability), but are limited by the completeness of templates, manual heuristics, and scalability problems.

According to empirical evidence, augmented neural-symbolic and bidirectional NSPR architectures present an optimistic trade-off for overlaying symbolic reasoning on or between neural patch generation. Pre-filters can be applied to eliminate infeasible candidate patches, in-generation constraints can be applied to divert decoding to semantically invalid tokens, or post-verification can be performed to eliminate hallucinated patches that only pass existing tests. Bidirectional designs take this a step further and inject symbolic information into training or decoding, such as encoding symbolic repair templates as supervision cues, or using solver feedback to bias the search space to verified patterns. In the range of studies that have evaluated Defects4J v1.2 with similar time budgets, varying time budgets, and

methods of patch validation, augmented or bidirectional NSPR systems tend to report higher plausible and/or correct repair rates than the mentioned baselines, but these results are dependent on the bug subset use, time budgets, and patch validation.

Concurrently, this review identifies critical methodological overtures when people interpret the reported performance. The fact that the versions of datasets used, bugs chosen, and evaluation protocols are revealed implies that repair rates cannot necessarily be compared directly between systems, even when they seem to use using the same benchmark. Some of the studies do not draw the distinction between plausibility (test-passing) and correctness, and some of the reported repair rates can contain overfitting patches. More conservative systems that are willing to filter candidates with symbolic checks may appear to have less apparent coverage but much higher semantic reliability. To do so in the future, it will be necessary to report on benchmark versions and bug subsets and on metric definitions with more accuracy and explicitly separate plausibility, correctness, and hallucination to support robust cross-study comparisons and meta-analysis.

C. Implications for industrial deployment and practice (RQ3)

Despite the encouraging outcomes of APR and NSPR systems in controlled laboratory experiments, their practical use is limited by practical factors in the industry. The deployments that are reported are usually in semi-autonomous or human-in-the-loop mode just since it is very necessary to carefully test the deployment before the generated patches are assured to be implemented into production systems. In addition, other typical operational constraints are reliance on high quality test suites, variability in patch acceptance policies through projects, and trust in automatically generated change by developers. The cases of failure that have been reported in industrial research would involve patches that fail incomplete test suites, patches that do not adhere to project-specific coding conventions, and performance overheads due to large-scale repairs search. It is indicated by such results that the existing APR and NSPR technologies are most applicable as developer-assistive devices more than entirely autonomous repair systems, especially operating in the safety-critical or massive production processes.

The 8 industrial and production-like reports examined in this study indicate that the implementation of APR and NSPR in real-life software engineering contexts can be achieved, but only with due consideration to integration with the current processes, tools, and risk management. APR/NSPR systems can serve as first responders in CI/CD pipelines because they produce candidate patches of failing builds, which are verified by existing test suites, and for non-trivial changes, they are entered into human review. NSPR components are employed to propose or partially automate the correction of known vulnerability patterns under stringent conditions of both static analysis and verification, with the ultimate control of human security engineers. In the open-source and educational domains, tools like APR can be used to provide faster feedback on beginner-level bugs and to triage large amounts of recurring problems.

These deployments also highlight the fact that in practice most organizations implement semi-autonomous or human-in-the-middle modes as opposed to a fully autonomous repair, particularly for high-impact or safety-critical systems. The choice of locations on which automated patch implementation should be applied or where to review manually is usually influenced by the criticality of the defects, sensitivity of the codebase, strength of the test suite and regulatory limitations. The literature raises issues of trust and explainability: developers might be unwilling to deploy black-box patches generated by LLM-based systems without an apparent justification or evidence of why they were generated, and auditors may insist on clear documentation of the manner in which a patch was generated and testing was performed in broader areas (such as finance, aviation, or healthcare). The formal constraints and justifications that are easy to read and understand in NSPR can be used to manage these issues, but without further complexity and computation.

The findings imply several practical lessons for practitioners. First, the usefulness of APR/NSPR tools is highly dependent on the quality and coverage of the available tests and static analysis; without the power of oracles, even advanced NSPR systems cannot ensure that accepted patches are semantically correct. Second, gradual adoption plans, such as beginning with low-risk areas or non-essential services, and gradually extending the coverage as one gets more confident and the tooling gets more ready, seem more practical than immediate and widespread adoption. Third, organizations are encouraged to view at APR/NSPR as part of a wider socio-technical system, where the developers, reviewers, platform owners, and governance bodies have clear roles and responsibilities and not as self-governing actors.

D. Methodological and Reporting Limitations of the Evidence base

The synthesized evidence used in this review has a few limitations that limit the strength of the conclusions drawn. First, the 70 major studies are highly heterogeneous both in terms of the choice of benchmark, version of the dataset, subset of bugs, and evaluation protocol; many studies report their results on a single benchmark, without necessarily specifying the exact configuration they used. These discrepancies render the calculation of meaningful pooled effect estimates unattainable and an accountable narrative synthesis that highlights ranges and patterns as opposed to significant ranks. Second, critical metrics such as repair rate, plausibility, and hallucination are defined and measure different things, and not all studies distinguish between patches that pass tests and those that are semantically correct by ground-truth fixes or manual inspection.

For critical appraisal of included studies, the quality of methodology of the involved APR and NSPR studies differs significantly. Although several more modern studies offer implementations that can be reproduced and offer well-defined benchmarks, older research tends to have less detailed reporting of experimental settings like a time limit budget and a subset of bugs. Also, comparison at the baseline is not always consistent and, in some studies, there is evaluation against

outdated and weak baselines. In addition, a significant part of the literature only claims plausible repairs which are made and not published as semantically correct based on independently testing their semantic adequacy in test-suites. This interferes with the possession of repair rates reported, and makes it difficult to make a direct comparison between techniques. Moreover, these shortcomings emphasize the significance of there being designation of experimental procedures and better reporting conducts in the upcoming APR and NSPR studies.

Third, the number of reports on industrial and production-like deployments is comparatively low, a significant part of them is anonymized, and most of the reports include only a general overview of benefits and difficulties. Although these reports are invaluable for gaining insight into real-world applicability, they rarely reveal raw figures, specific failure scenarios, and long-term operational data in the long run, and bad or neutral experiences may be underreported. Fourth, the search was limited to English-language publications and sampled digital libraries, potentially missing pertinent work in other languages or non-indexed publications (even though snowballing and reference screening were used). Lastly, a formal certainty-of-evidence framework, like GRADE, was not used in this review, partly due to the fact that there is no standardized risk-of-bias tool designed to address APR and NSPR studies, but risk of bias has been evaluated using qualitative methods, i.e., benchmark selection, completeness of metrics used, and clarity of reporting.

E. Quality and bias Trends

According to the trends of the quality and bias among the primary empirical studies involved, a number of common approach weaknesses were identified. Such problems as not fully reported experimental settings, insufficient access to source code or data to reproduce, and unstable specification of time out budgets and bug selection were considered to be common. There was a high number of studies that merely gave plausible repairs but have not tested the semantic correctness independently by using test-suites that are adequate.

The results of the per-study quality assessment are shown in Supplementary Table S2 and the overall trends are summarized in percentages across the studies included. These tendencies prove that, even though the APR/NSPR literature is quickly developing, the reporting practices are still heterogeneous and, in many cases, may not allow making the direct comparison between the systems.

F. Directions for future research

This synthesis identifies some potential avenues for future APR and NSPR studies. At the algorithmic level, more extensive use of symbolic reasoning in the decoding stage of LLMs, such as using constraints to select tokens or search using a solver or multimodal representations of programs should be more successful in reducing hallucinations and increasing explainability, especially for complex, multi-line, or architectural bugs. It will need improved abstractions of cross-language reasoning and more efficient symbolic procedures to scale NSPR to large and heterogeneous codebases, as well as to handle other emerging languages. A more systematic study of human factors in NSPR, such as the interaction between developers and proposed patches, the establishment or destruction of trust, and whether explanation interfaces can render symbolic justifications applicable in practice, is also needed.

From an assessment perspective, the discipline could use common benchmark suites with well-versioned datasets, standard bug subsets, and deeper ground truth annotations, capable of supporting more refined metrics, such as repair minimality, readability, and long-term stability. Publicly available leaderboards and reproducible and open-source implementations of APR/NSPR systems would also assist in minimizing the differences between reported and independently reproducible performance. Finally, the industrial case studies with more transparent deployments, clear analyses of failure, and detailed governance models should be more common to learn how NSPR can match regulatory and ethical standards in safety-critical and high-impact environments.

G. Overall assessment

Collectively, the papers analyzed in this study suggest that neuro-symbolic automated program repair is a viable research path that can significantly enhance repair accuracy and alleviate hallucinations when compared to purely neural or purely symbolic baselines, particularly on popular benchmarks. Simultaneously, the existing evidence base is still disjointed, with inconsistent benchmarks, measure types, and reporting habits reducing the power of comparative statements. Further development of the evaluation methods, the extension of the transparent reporting of industrial operations, and further improvement of the integration of neural and symbolic methods will also be the key to the appearance of NSPR as an effective part of contemporary software engineering practice.

V. CONCLUSION

This review of the literature involved 70 systematic automated program repair (APR) studies and 8 industrial or production-like deployments as part of an effort to elucidate how systems are built, tested, and put into practice in the modern era. The evidence demonstrates a visible shift in the direction of the classical template-based and strictly symbolic models to the hybrid ones that integrate large language models with static analysis, solving SMT, etc. to balance between the coverage of repair and semantic and interpretable results. Augmented neural-symbolic and bidirectional NSPR systems tend to have higher correct repair rates and significantly lower hallucination rates on popular benchmarks such as Defects4J and QuixBugs, but at a higher computational and engineering cost.

According to industrial reports, APR and NSPR can be fully deployed into CI/CD pipelines, security patch processes, educational systems, and open-source maintenance processes, most commonly in semi-autonomous, human-in-the-loop systems, where developers still have ultimate control over accepting patches. Simultaneously, the existing evidence base has weaknesses in terms of non-uniform benchmarks, inconsistencies in reporting data version and measures, and

scarce and, in some cases, anonymized deployment data, which limit the quality of comparative data claims. To progress in the field in the future, we should focus on standardized, versioned benchmarks, more articulate definitions of the distinctions between plausibility, correctness, and hallucination, and more open industrial case studies that reveal both success cases and failure modes. Overall, neuro-symbolic program repair seems to be potentially beneficial and a more realistic future of AI-assisted software engineering; however, it can only be successfully applied to safety-critical and high-impact fields with better empirical support, better evaluation practices, better governance, and human-in-the-loop oversight.

Author Contributions: Conceptualization, A.A.; methodology, A.A.; validation, A.A.; formal analysis, A.A.; investigation, A.A.; writing—original draft preparation, A.A.; writing—review and editing, A.A.; visualization, A.A.; supervision. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Informed Consent Statement: Informed consent was obtained from all subjects involved in the study.

Data Availability Statement: The original contributions presented in this study are included in the article. Further inquiries can be directed to the corresponding author.

Conflicts of Interest: The authors declare no conflict of interest.

Acknowledgments: All Figures/ Tables are created by the authors; no GenAI was used. The authors have reviewed and edited the output and take full responsibility for the content of this publication."

REFERENCES

[1] Campos, Viola, et al. "Exploring Generalizable Automated Program Repair with Large Language Models." ArXiv.org, 3 June 2025, https://doi.org/10.48550/arxiv.2506.03283. Accessed 25 Feb. 2026.

[2] E. Dehaerne, B. Dey, S. Halder, S. De Gendt, and W. Meert, "Code Generation Using Machine Learning: A Systematic Review," *IEEE Access*, vol. 10, pp. 82434–82455, 2022, doi: 10.1109/ACCESS.2022.3196347.

[3] I. Bouzenia, P. Devanbu, and M. Pradel, "RepairAgent: An Autonomous, LLM-Based Agent for Program Repair," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, IEEE, Apr. 2025, pp. 2188–2200. doi: 10.1109/ICSE55347.2025.00157.

[4] Chandra Maddila, "Agentic Program Repair from Test Failures at Scale: A Neuro-symbolic approach with static analysis and test execution feedback," *ArXiv*, 2025.

[5] B. ; C. Z. ; L. F. ; L. X. B. D. ; Z. L. ; B. T. F. ; L. Y. ; T. H. Yang, "A Survey of LLM Based Automated Program Repair," *ArXiv*, 2025.

[6] B. Liang, Y. Wang, and C. Tong, "AI Reasoning in Deep Learning Era: From Symbolic AI to Neural–Symbolic AI," *Mathematics*, vol. 13, no. 11, p. 1707, May 2025, doi: 10.3390/math13111707.

[7] F. Zubair, M. Al-Hitmi, and C. Catal, "The use of large language models for program repair," *Comput. Stand. Interfaces*, vol. 93, p. 103951, Apr. 2025, doi: 10.1016/j.csi.2024.103951.

[8] Bui, Quang-Cuong, et al. "APR4Vul: An Empirical Study of Automatic Program Repair Techniques on Real-World Java Vulnerabilities." Empirical Software Engineering, vol. 29, no. 1, 6 Dec. 2023, https://doi.org/10.1007/s10664-023-10415-7. Accessed 20 Feb. 2024.

[9] Md. S. H. Shaon and M. S. Akter, "Modern Approaches to Software Vulnerability Detection: A Survey of Machine Learning, Deep Learning, and Large Language Models," *Electronics (Basel).*, vol. 14, no. 22, p. 4449, Nov. 2025, doi: 10.3390/electronics14224449.

[10] C. Woesle, L. Fischer-Brandies, and R. Buettner, "A Systematic Literature Review of Hallucinations in Large Language Models," *IEEE Access*, vol. 13, pp. 148231–148253, 2025, doi: 10.1109/ACCESS.2025.3601206.

[11] S. Banitaan, M. Daoud, H. Alquran, and M. Akour, "Foundation Models in Software Engineering: A Taxonomy, Systematic Review, and In-Depth Analysis of Testing Support," *Information*, vol. 17, no. 1, p. 73, Jan. 2026, doi: 10.3390/info17010073.

[12] 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 2014 International Symposium on Software Testing and Analysis*, New York, NY, USA: ACM, Jul. 2014, pp. 437–440. doi: 10.1145/2610384.2628055.

[13] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama, "QuixBugs: a multi-lingual program repair benchmark set based on the quixey challenge," in *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, New York, NY, USA: ACM, Oct. 2017, pp. 55–56. doi: 10.1145/3135932.3135941.

[14] Dikici, Sena, and Turgay Tugay Bilgin. "Advancements in Automated Program Repair: A Comprehensive Review." Knowledge and Information Systems, 6 Mar. 2025, https://doi.org/10.1007/s10115-025-02383-9.

[15] F. Corradini, M. Leonesi, and M. Piangerelli, "State of the Art and Future Directions of Small Language Models: A Systematic Review," *Big Data and Cognitive Computing*, vol. 9, no. 7, p. 189, Jul. 2025, doi: 10.3390/bdcc9070189.

[16] A. Faraji and N. Pombo, "AI-Driven Software Test Automation: An AI4SE-Oriented Survey of Techniques, Tools, and Challenges," *IEEE Access*, vol. 13, pp. 183296–183313, 2025, doi: 10.1109/ACCESS.2025.3623944.

[17] T. Saber and N. Tao, "Review and Mapping of Search-Based Approaches for Program Synthesis," *Information*, vol. 16, no. 5, p. 401, May 2025, doi: 10.3390/info16050401.

[18] M. Alharbi and M. Alshayeb, "Automatic Code Generation Techniques: A Systematic Literature Review," *Automated Software Engineering*, vol. 33, no. 1, p. 4, May 2026, doi: 10.1007/s10515-025-00551-3.

[19] L. B. Germano, R. R. Goldschmidt, R. C. Noya, and J. C. Duarte, "A Systematic Review on Detection, Repair, and Explanation of Vulnerabilities in Source Code Using Large Language Models," *IEEE Access*, vol. 13, pp. 192263–192293, 2025, doi: 10.1109/ACCESS.2025.3631363.

[20] B. C. , & R. W. Colelough, "Neuro-Symbolic AI in 2024: A Systematic Review," *ArXiv (Cornell University)*, 2025.

[21] S. Parihar, Z. Dadachanji, P. K. Singh, R. Das, A. Karkare, and A. Bhattacharya, "Automatic Grading and Feedback using Program Repair for Introductory Programming Courses," in *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*, New York, NY, USA: ACM, Jun. 2017, pp. 92–97. doi: 10.1145/3059009.3059026.

[22] J. C. Paiva, J. P. Leal, and Á. Figueira, "Incremental Repair Feedback on Automated Assessment of Programming Assignments," *Electronics (Basel).*, vol. 14, no. 4, p. 819, Feb. 2025, doi: 10.3390/electronics14040819.

[23] D. Choi and E. Lee, "Automated Feedback Generation for Programming Assignments Through Diversification," in *2025 IEEE/ACM 37th International Conference on Software Engineering Education and Training (CSEE&T)*, IEEE, Apr. 2025, pp. 230–241. doi: 10.1109/CSEET66350.2025.00030.

[24] Y. Yang *et al.*, "Patch Generation in APR: A Survey from the Perspectives of Utilizing LLMs and Using APR-Specific Information," *ACM Transactions on Software Engineering and Methodology*, Aug. 2025, doi: 10.1145/3764584.

[25] C. Geethal, M. Böhme, and V.-T. Pham, "Human-in-the-Loop Automatic Program Repair," *IEEE Transactions on Software Engineering*, vol. 49, no. 10, pp. 4526–4549, Oct. 2023, doi: 10.1109/TSE.2023.3305052.

[26] K. Etemadi, N. Tarighat, S. Yadav, M. Martinez, and M. Monperrus, "Estimating the potential of program repair search spaces with commit analysis," *Journal of Systems and Software*, vol. 188, p. 111263, Jun. 2022, doi: 10.1016/j.jss.2022.111263.

[27] F. N. Meem, J. Smith, and B. Johnson, "Exploring Experiences with Automated Program Repair in Practice," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, New York, NY, USA: ACM, Apr. 2024, pp. 1–11. doi: 10.1145/3597503.3639182.

[28] R. Majumdar and J.-F. Raskin, "Symbolic Model Checking in Non-Boolean Domains," in *Handbook of Model Checking*, Cham: Springer International Publishing, 2018, pp. 1111–1147. doi: 10.1007/978-3-319-10575-8_31.

[29] S. Hao, X. Shi, H. Liu, Y. Yin, and X. Chen, "Template-guided interpretable reasoning with execution feedback for LLM-based program repair," *Inf. Softw. Technol.*, vol. 193, p. 108058, May 2026, doi: 10.1016/j.infsof.2026.108058.

[30] F. Li, J. Jiang, J. Sun, and H. Zhang, "Hybrid Automated Program Repair by Combining Large Language Models and Program Analysis," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 7, pp. 1–28, Sep. 2025, doi: 10.1145/3715004.

[31] N. Jiang, K. Liu, T. Lutellier, and L. Tan, "Impact of Code Language Models on Automated Program Repair," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, IEEE, May 2023, pp. 1430–1442. doi: 10.1109/ICSE48619.2023.00125.

[32] W. Zhong, C. Li, J. Ge, and B. Luo, "Neural Program Repair : Systems, Challenges and Solutions," in *Proceedings of the 13th Asia-Pacific Symposium on Internetware*, New York, NY, USA: ACM, Jun. 2022, pp. 96–106. doi: 10.1145/3545258.3545268.

[33] C. Ni, X. Yin, X. Li, X. Xu, and Z. Yu, "Abundant Modalities Offer More Nutrients: Multi-Modal-Based Function-Level Vulnerability Detection," *ACM Transactions on Software Engineering and Methodology*, vol. 35, no. 2, pp. 1–31, Feb. 2026, doi: 10.1145/3731557.

[34] R. and P. S. and K. A. and S. S. Gupta, "DeepFix: fixing common C language errors by deep learning," *AAAI'17: Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, pp. 1345–1351, 2017.

[35] T. Le-Cong, D. Nguyen, B. Le, and T. Murray, "Towards Reliable Evaluation of Neural Program Repair with Natural Robustness Testing," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 7, pp. 1–44, Sep. 2025, doi: 10.1145/3716167.

[36] A. T. C. C. D. C. Chandra, "Agentic Program Repair from Test Failures at Scale: A Neuro-symbolic approach with static analysis and test execution feedback," *ArXiv*, 2025.

[37] M. M. Ruslan Idelfonso Magana Vsevolodovna, "Enhancing Large Language Models through Neuro-Symbolic Integration and Ontological Reasoning," *ArXiv*, 2025.

[38] F. L. X. Q. H. Z. J. J. Jiajun Sun, "Empirical Evaluation of Large Language Models in Automated Program Repair," *ArXiv*, 2025.

[39] Y. Chen *et al.*, "When Large Language Models Confront Repository-Level Automatic Program Repair: How Well They Done?," in *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, New York, NY, USA: ACM, Apr. 2024, pp. 459–471. doi: 10.1145/3639478.3647633.

[40] W. , J. C. , S. K. , S. M. , J. P. , T. P. , F. S. David, "User-Centric Deployment of Automated Program Repair at Bloomberg," *ArXiv*, 2023.

[41] A. Marginean *et al.*, "SapFix: Automated End-to-End Repair at Scale," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, IEEE, May 2019, pp. 269–278. doi: 10.1109/ICSE-SEIP.2019.00039.

[42] H. Eladawy, C. Le Goues, and Y. Brun, "Automated Program Repair, What Is It Good For? Not Absolutely Nothing!," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, New York, NY, USA: ACM, Apr. 2024, pp. 1–13. doi: 10.1145/3597503.3639095.

[43] K. Noda, Y. Nemoto, K. Hotta, H. Tanida, and S. Kikuchi, "Experience Report: How Effective is Automated Program Repair for Industrial Software?," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE, Feb. 2020, pp. 612–616. doi: 10.1109/SANER48275.2020.9054829.

[44] K. Liu *et al.*, "On the efficiency of test suite based program repair," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, New York, NY, USA: ACM, Jun. 2020, pp. 615–627. doi: 10.1145/3377811.3380338.

[45] D. Yang, Y. Qi, and X. Mao, "An Empirical Study on the Usage of Fault Localization in Automated Program Repair," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, Sep. 2017, pp. 504–508. doi: 10.1109/ICSME.2017.37.

[46] Q. Zhang, C. Fang, T. Zhang, B. Yu, W. Sun, and Z. Chen, "Gamma: Revisiting Template-Based Automated Program Repair Via Mask Prediction," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, Sep. 2023, pp. 535–547. doi: 10.1109/ASE56229.2023.00063.

[47] 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, Jan. 2012, doi: 10.1109/TSE.2011.104.

[48] F. Long and M. Rinard, "An analysis of the search spaces for generate and validate patch generation systems," in *Proceedings of the 38th International Conference on Software Engineering*, New York, NY, USA: ACM, May 2016, pp. 702–713. doi: 10.1145/2884781.2884872.

[49] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *2013 35th International Conference on Software Engineering (ICSE)*, IEEE, May 2013, pp. 802–811. doi: 10.1109/ICSE.2013.6606626.

[50] Y. Wang, W. Wang, S. Joty, and S. C. H. Hoi, "CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, Stroudsburg, PA, USA: Association for Computational Linguistics, 2021, pp. 8696–8708. doi: 10.18653/v1/2021.emnlp-main.685.

[51] Y. Tang, "Large Language Models Meet Automated Program Repair: Innovations, Challenges and Solutions," *Applied and Computational Engineering*, vol. 117, no. 1, pp. 22–30, Dec. 2024, doi: 10.54254/2755-2721/2024.18303.

[52] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix," in *Proceedings of the 38th International Conference on Software Engineering*, New York, NY, USA: ACM, May 2016, pp. 691–701. doi: 10.1145/2884781.2884807.

[53] Q. Zhu *et al.*, "A syntax-guided edit decoder for neural program repair," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, New York, NY, USA: ACM, Aug. 2021, pp. 341–353. doi: 10.1145/3468264.3468544.

[54] M. Martinez and M. Monperrus, "ASTOR: a program repair library for Java (demo)," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, New York, NY, USA: ACM, Jul. 2016, pp. 441–444. doi: 10.1145/2931037.2948705.

[55] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair," in *Proceedings of the 36th International Conference on Software Engineering*, New York, NY, USA: ACM, May 2014, pp. 254–265. doi: 10.1145/2568225.2568254.

[56] Y. Xiong *et al.*, "Precise Condition Synthesis for Program Repair," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, IEEE, May 2017, pp. 416–426. doi: 10.1109/ICSE.2017.45.

[57] Z. Fan, X. Gao, M. Mirchev, A. Roychoudhury, and S. H. Tan, "Automated Repair of Programs from Large Language Models," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, IEEE, May 2023, pp. 1469–1481. doi: 10.1109/ICSE48619.2023.00128.

[58] X.-B. D. Le, D.-H. Chu, D. Lo, C. Le Goues, and W. Visser, "S3: syntax- and semantic-guided repair synthesis via programming by examples," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, New York, NY, USA: ACM, Aug. 2017, pp. 593–604. doi: 10.1145/3106237.3106309.

[59] Q. Zhang, C. Fang, Y. Ma, W. Sun, and Z. Chen, "A Survey of Learning-based Automated Program Repair," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 2, pp. 1–69, Feb. 2024, doi: 10.1145/3631974.

[60] Y. , Z. C. , F. L. , X. B. Boyang, "A Survey of LLM-based Automated Program Repair: Taxonomies, Design Paradigms, and Applications," *ArXiv*, 2025.

[61] L. Chen, Y. Pei, and C. A. Furia, "Contract-based program repair without the contracts," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, Oct. 2017, pp. 637–647. doi: 10.1109/ASE.2017.8115674.

[62] Y. W. D. C. T. W. R. S. C. C. J. Rohan Mukherjee, "Neural Program Generation Modulo Static Analysis," *ArXiv*, 2021.

[63] F. Long and M. Rinard, "Staged program repair with condition synthesis," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, New York, NY, USA: ACM, Aug. 2015, pp. 166–178. doi: 10.1145/2786805.2786811.

[64] C. Le Goues *et al.*, "The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs," *IEEE Transactions on Software Engineering*, vol. 41, no. 12, pp. 1236–1256, Dec. 2015, doi: 10.1109/TSE.2015.2454513.

[65] 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.

[66] C. Le Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Commun. ACM*, vol. 62, no. 12, pp. 56–65, Nov. 2019, doi: 10.1145/3318162.

[67] M. Monperrus, "Automatic Software Repair," *ACM Comput. Surv.*, vol. 51, no. 1, pp. 1–24, Jan. 2019, doi: 10.1145/3105906.

[68] M. Bohme, V.-T. Pham, and A. Roychoudhury, "Coverage-Based Greybox Fuzzing as Markov Chain," *IEEE Transactions on Software Engineering*, vol. 45, no. 5, pp. 489–506, May 2019, doi: 10.1109/TSE.2017.2785841.

[69] M. Motwani, S. Sankaranarayanan, R. Just, and Y. Brun, "Do automated program repair techniques repair hard and important bugs?," *Empir. Softw. Eng.*, vol. 23, no. 5, pp. 2901–2947, Oct. 2018, doi: 10.1007/s10664-017-9550-0.

[70] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation," *ACM Transactions on Software Engineering and Methodology*, vol. 28, no. 4, pp. 1–29, Oct. 2019, doi: 10.1145/3340544.

[71] C. Zhang and J. Chen, "Fuzzing Methods Recommendation Based on Feature Vectors," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, Nov. 2021, pp. 1079–1081. doi: 10.1109/ASE51524.2021.9678630.

[72] J. Imtiaz, M. Z. Iqbal, and M. U. khan, "An automated model-based approach to repair test suites of evolving web applications," *Journal of Systems and Software*, vol. 171, p. 110841, Jan. 2021, doi: 10.1016/j.jss.2020.110841.

[73] Z. Chen, S. J. 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*, pp. 1–1, 2021, doi: 10.1109/TSE.2019.2940179.

[74] Z. X. S. Y. H. S. X. L. Z. Y. Y. Z. Kai Huang, "A Survey on Automated Program Repair Techniques," *arXiv* , 2023.

[75] C. S. Xia, Y. Wei, and L. Zhang, "Automated Program Repair in the Era of Large Pre-trained Language Models," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, IEEE, May 2023, pp. 1482–1494. doi: 10.1109/ICSE48619.2023.00129.

[76] S. Hao, X. Shi, and H. Liu, "Exploring the Potential of Pre-Trained Language Models of Code for Automated Program Repair," *Electronics (Basel).*, vol. 13, no. 7, p. 1200, Mar. 2024, doi: 10.3390/electronics13071200.

[77] H. Joshi, J. Cambronero Sanchez, S. Gulwani, V. Le, G. Verbruggen, and I. Radiček, "Repair Is Nearly Generation: Multilingual Program Repair with LLMs," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 37, no. 4, pp. 5131–5140, Jun. 2023, doi: 10.1609/aaai.v37i4.25642.

[78] S. Bhatia, P. Kohli, and R. Singh, "Neuro-symbolic program corrector for introductory programming assignments," in *Proceedings of the 40th International Conference on Software Engineering*, New York, NY, USA: ACM, May 2018, pp. 60–70. doi: 10.1145/3180155.3180219.