INSTITUTE OF INFORMATION AND COMMUNICATION TECHNOLOGIES
BULGARIAN ACADEMY OF SCIENCES
CYBERNETICS AND INFORMATION TECHNOLOGIES • Volume **26**, No 1
Sofia • 2026 Print ISSN: 1311-9702; Online ISSN: 1314-4081
DOI: 10.2478/cait-2026-0006

Automated Bug Detection and Program Repair Using Deep Learning: A Comprehensive Review

Rawaa Hamza Ali[1], *Adala Mahdi Chyad*[2]

[1]*Department of Biology, College of Science, University of Misan, Maysan, Iraq*
[2]*College of Computer Science & Information Technology, University of Basrah, Iraq*
E-mails: *rawaaha@uomisan.edu.iq adala.gyad@uobasrah.edu.iq*

***Abstract*:** *Critical failures and financial losses are two of the biggest problems with bug detection. Traditional debugging methods don't work well as the problem gets more complicated. Large language models and deep learning have lately given promise to the automation of finding and fixing problems in software. These systems are better at finding defects and making patches than traditional methods because they learn syntactic and semantic patterns. This study presents a systematic review of benchmark datasets, detection algorithms, and repair frameworks published between 2018 and 2025. This article compares the creation of models based on graphs and tokens with that of transformer architectures and large language model -driven methodologies. It also talks about their pros and cons and how they are used in the real world. The paper also discusses unresolved challenges related to explainability, accuracy guarantees, and cross-project generalization. It also talks about scalability, validation, and evaluation metrics. It identifies research deficiencies and delineates prospective avenues for developing more reliable and robust software systems by integrating contemporary breakthroughs and offering a current summary of automated debugging research utilizing deep learning and large language model methodologies.*

***Keywords*:** *Program repair, Automated debugging, Large language models, Deep learning, Software engineering, Bug detection.*

1. Introduction

In industries with a lot of risks, including healthcare, transportation, finance, and defense, where mistakes can have serious consequences, software is very important. More intricate software systems are more likely to have bugs that cause problems, lose data, or put security at risk [1]. Despite significant advances in software engineering, substantial time and financial resources are still devoted to debugging and maintenance [2, 3]. Research shows that software bugs have significant global economic impacts, highlighting the need for robust automated debugging tools. As software projects get bigger and more involved, manual debugging gets harder and harder since there are more and more faults and various sources of error [4, 5]. To get

around these limits, Machine Learning (ML) methods, notably Deep Learning (DL), have been utilized to automate debugging tasks, including finding, locating, prioritizing, and fixing issues [2, 6]. Over the past few years, deep learning models have done an amazing job of learning semantic and syntactic patterns from execution traces, historical bug-fix data, and source code. Modern methods employ Graph Neural Network (GNN), transformer designs, and massive pre-trained language models. Older methods, on the other hand, used sequence and token-based models. Models like CodeBERT, CodeT5, and GPT-based solutions have made it easier to find and fix bugs in programs automatically. These models allow for contextual understanding of code, zero-shot and few-shot reasoning, and the automatic generation of patches [7]. In this way, automated bug detection and software repair can be viewed as two components of a unified process rather than separate tasks. Deep learning models are used to find code that is causing problems, which is the first stage in automated program repair. After that, candidate patches are made and tested based on previous problem fixes and test results. This integrated approach, depicted in Fig. 2, integrates detection, localization, repair, and validation in a continuous feedback loop that improves repair effectiveness and reliability over time. It fits with the primary idea of most contemporary deep learning and Large Language Model (LLM) based debugging solutions [1, 5]. Automated Program Repair (APR) is becoming more popular as a helpful tool for making debugging easier since it makes patches to fix bugs based on patterns acquired from existing program issues. Even while DL-based and LLM-based APR systems have done quite well on benchmark datasets like Defects4J, there are still certain concerns that need to be handled. Many strategies work well on certain benchmarks, but they typically have trouble applying to other datasets or tasks. Problems with reproducibility, overfitting to test suites, the lack of assurances for semantic accuracy, and the lack of emphasis on developers' confidence and explainability are other things that make it hard for people to use these technologies in the real world [5, 8]. Other investigations have looked at different aspects of automated bug finding and software repair on their own [9]. Recent progress has been made in huge language models, evaluation methods, and benchmark restrictions. However, there is currently no comprehensive and up-to-date synthesis that looks at deep learning-based issue detection and program repair at the same time [10, 11]. Moreover, contemporary assessments often emphasize performance improvements while inadequately addressing generalizability, repeatability, and human-centric considerations [12, 13]. To address these deficiencies, this study presents a literature review covering the years 2018-2025 that methodically analyses deep learning and LLM-based approaches for automated bug detection and software remediation. The study looks at model designs, benchmark datasets, validation frameworks, repair techniques, assessment metrics, and evaluation metrics. It also talks about present challenges and possible future research pathways. These questions are the starting point for the investigation:

- **RQ1.** What validation frameworks and assessment criteria are utilized to find out how well bug detection and repair methods are based on DL and LLM work, how well they can be used on a large scale, and how accurate they are?

- **RQ 2.** What are the most widely used benchmark datasets, and how do they stack up in terms of their pros and cons?
- **RQ3.** How have strategies for identifying defects in deep learning models evolved, transitioning from graph-based and token-based methods to transformers?
- **RQ 4.** How do large language models contribute to automated program repair, and how do they compare with classical and neural APR systems?
- **RQ 5.** How do the limitations and open problems of current methodologies affect their generalizability, explainability, and reliability?

Paper organization. The next parts of this article are in a logical order to talk about the study methodology, literature review, assessment, and key results. The Research Methodology is in Section 2. It talks about how to find relevant content and what makes something eligible for inclusion. In Section 3, we look at past surveys and benchmark datasets that are often used. Section 4 of the previous literature explains. Section 5 talks about how to measure things for evaluation and how to set up frameworks for validation. We talk about ways to automatically fix software. Section 6 and ways to find issues and repair that use deep learning. Section 7 Checking and Testing. Section 8 talks about important problems, limitations, and topics that the research doesn't solve. Section 9 sums up Future directions. The essay comes to an end section 10 discussion of research questions

Fig. 1 illustrates the general workflow of automated software bug detection and repair using deep learning and large language models, which frames the scope of this survey.

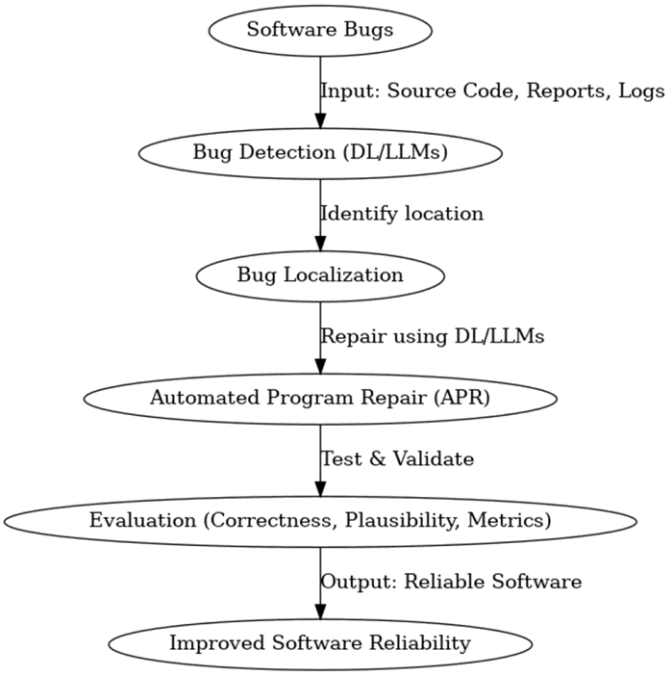

Fig. 1. General Workflow: Detect and repair software errors using DL/LLMs

2. Research methodology

The literature of DL and LLM-based methodologies for automated bug detection and APR is comprehensively examined using a systematic and structured approach to ensure repeatability, transparency, and extensive coverage. To provide a comprehensive synthesis of studies published from 2018 to 2025, the methodology is designed to address the research topics outlined in Section 1.

2.1. Literature search strategy

A thorough search of the literature was done utilizing well-known online libraries that are recognized for their work in AI and software engineering. The major sources were Elsevier ScienceDirect, MDPI, SpringerLink, ACM Digital Library, and IEEE Xplore. We also used Google Scholar to discover recent studies and early-access publications about deep learning and debugging methods that used LLM. We only looked for publications published between 2018 and 2025 since deep learning and large language models have had a major impact on research on automated bug finding and software repair during that time.

Search queries. Some of the words utilized in the literature search were "bug finding", "program repair", "deep learning", "big language models", along with "benchmark datasets". Here are a few examples of search phrases:

- "automated bug detection" in conjunction with "deep learning";
- "automated program repair" and "neural networks", respectively;
- "deep learning models" AND "repair of programs";
- "ManySStuBs4J" OR "Defects4J" OR "Bugs.jar";
- "software defects" and "transformer-based models".

The syntax of each digital library is changed to make it easier to answer these questions. We received roughly 300 publications on the first try.

2.2. Study selection criteria

The studies that were gathered had to meet precise inclusion and exclusion criteria to make sure they were relevant and of good quality.

Inclusion criteria

- Papers in academic publications, speeches at professional conferences, and survey findings.
- English-language books that came out between 2018 and 2025.
- Investigations into automated software repair, defect prediction, and the use of machine learning, deep learning, and massive language models for these objectives.
- Defects4J, Bugs.jar, Bears, and ManySStuBs4J are benchmark datasets utilized in research for empirical assessment reporting.

Exclusion criteria

Some examples of information that haven't been peer-reviewed are blog entries, technical reports, and tutorials.

- Research that doesn't include any learning-based parts and instead concentrates on static or dynamic analysis in the traditional sense.
- Researchers only maintained the most up-to-date and complete version of publications that had previously been published.
- Articles that don't do a good job of explaining the technique or giving an empirical assessment.

After going through these steps and getting rid of duplicates, about 130 articles were preserved for more in-depth study.

2.3. Data extraction and classification

Information, including publication metadata (authors, year, and venue) was carefully gathered for each research that was chosen.

- Areas of research (integrated pipelines, finding and fixing bugs, and finding and fixing bugs).
- The type of model, such as a hybrid, token, graph, transformer, LLM, or anything else.
- Data sets used as standards.
- Important parts include evaluation metrics and validation techniques.
- Recognized skills, strengths, and weaknesses.
- We utilized structured tables to arrange the data we got back so that we could easily compare different evaluation procedures, datasets, and techniques.

2.4. Synthesis and analysis approach

Qualitative synthesis was employed instead of a quantitative meta-analysis due to the utilization of diverse datasets, evaluation measures, and experimental circumstances across many studies. The main areas of research were:

- The creation of deep learning frameworks for spotting flaws.
- The good and bad things about APR systems that use LLM, neural networks, and templates.
- Common ways to monitor performance and the problems they cause.
- The capacity to extrapolate findings from alternative initiatives and databases.
- Things to think about while validating repeatability and putting people first.

This technique does more than just compare accuracy; it also shows trends in the research, places where performance is poor, and concerns that still need to be solved.

2.5. Methodological validity and limitations

There are still certain limits, even if the selected strategy provides extensive coverage and careful selection. The review may have been biassed by publication bias in favor of positive outcomes and the fact that some benchmarks, such Defects4J, are more common in the literature. A rigorous and fair examination of the issue is made easier by systematic analysis and clear selection criteria, which lower these risks.

3. Benchmark datasets and prior surveys

Datasets are the cornerstone of DL-based bug detection and repair. They provide training data, evaluation benchmarks, and reproducibility for comparative studies [14]. Table 1 summarizes widely adopted datasets. Even though they are popular, benchmark datasets for automated software repair and deep learning-based bug discovery include biases that make the stated performance appear different. The dataset you choose affects the sorts of faults you may fix, which in turn affects the model's general ability, evaluation validity, and repeatability. For this reason, it's important to grasp the pros and cons of each benchmark before you look at experimental data, so you don't make the wrong assumptions about how effectively DL and LLM approaches function.

3.1. Defects4J

Originally released in 2014 and later expanded to version 2.0, Defects4J has become the gold standard for evaluating APR systems. It provides hundreds of real, reproducible faults drawn from well-known open-source Java projects. Version 2.0 contains 835 bugs across 17 projects, each packaged with a test suite that fails on the bug version and passes on the corrected one [15, 16]. Because Defects4J only covers Java and a small number of bug kinds, supplementary datasets have been developed. To guarantee reproducibility, Defects4J includes at least one failed test for every defect. The fault triggering test is the name given to this unsuccessful test. Defects4J employs an automated phase to extract candidate fault-triggering tests from the system's bug fixes and buggy commits, as not all problems are guaranteed to have a fault-triggering test when they are first discovered. In particular, a fault-triggering test has to fail on the flawed commit and pass deterministically on the repaired commit. After that, each test that causes bugs and faults is personally reviewed to remove any unnecessary code modifications, such as adding new functionality. By default, Defects4J reproduces the flaws using developer-written tests that act as fault-triggering tests [15, 17].

3.2. Bugs.jar

To go beyond the scope of Defects4J, researchers introduced Bugs.jar. This dataset comprises 1,158 real bugs from eight large Apache projects, including Commons Math and Mahout. Each entry includes the buggy code, the fixed code, and the relevant test cases. What sets Bugs.jar apart is its complexity. The projects it draws from are larger and more sophisticated, meaning the bugs tend to be more challenging and the repair tasks more demanding. This makes it a useful test of scalability, assessing whether repair approaches can handle large real-world projects. The drawback is that these project types are hard to assess, as the compilation and execution of the project may be slow. Nevertheless, in combination with Defects4J, Bugs.jar provides researchers with a wider and more realistic picture of how their tools can work [18, 19]. Bugs.jar was constructed by systematically mining real bug-fix commits from large-scale open-source Java projects, ensuring that each bug instance includes the faulty version, the corresponding fix, and associated test cases.

This dataset aims to provide realistic and diverse bug scenarios that reflect practical software development environments. The representation of data, a few of its use cases, and an example of three of the use cases using Bugs.jar with three different tools, Ekstazi and JaCoCo. Although Defects4J and Bugs.jar work with curated repositories, The Bears dataset operates differently from traditional benchmark suites by mining bugs from Continuous Integration (CI) failures in GitHub-based Java projects. Each bug instance is associated with a failing build followed by a succeeding build, capturing regression-like faults that arise during active development. As a result, Bears reflects realistic CI-driven debugging scenarios, where patches are typically small and incremental, and build environments are preserved using Docker containers. Although the Bears dataset represents an independent benchmark, it is discussed alongside Bugs.jar due to their shared emphasis on real-world Java projects and commit-level bug-fix instances. The primary distinction lies in the data collection process: Bears focuses on CI build failures, whereas Bugs.jar is derived from manually curated bug-fix commits. While Bears covers a broader diversity of projects and is valuable for evaluating generalization across heterogeneous codebases, both datasets suffer from replication and execution challenges caused by incomplete configuration environments and missing dependencies [8, 20].

3.3. ManySStuBs4J

Nicknamed the “many stupid bugs” dataset, ManySStuBs4J was introduced in 2020 and has become a favorite for machine learning researchers. It contains more than 153,000 small bug-fix pairs mined from over a thousand Java projects. Each bug is minimal, often a one-line mistake such as swapping function arguments, using the wrong operator, or typing the wrong literal value [21]. ManySStuBs4J With an emphasis on single-statement modifications per release, the 10,231 and 63,923 single-statement bug cases in the ManySStuBs4J dataset were taken from 12,598 and 86,771 bug patches, respectively. The GitHub repository contains detailed information about the dataset, which is saved in JSON files. Annotations for the satisfied SStuB pattern, project name, Java file name, commit hash, bug start line, and Abstract Syntax Tree (AST) subtree position are included with every SStuB instance [8, 22, 23].The value of this dataset lies in its sheer size. With so many examples of common mistakes, it provides a rich training ground for neural models. Instead of relying on a handful of carefully chosen bugs, researchers can train on hundreds of thousands of small fixes, allowing their models to learn general bug-fixing patterns at scale. Unlike Defects4J or Bears, ManySStuBs4J doesn’t include test cases; it’s essentially a massive collection of “before and after” code snippets. As a result, it’s mainly used to train models or to evaluate patch generation in static terms, rather than through dynamic testing [21]. Together, these four datasets form the foundation of most modern work in DL-based bug detection and repair: Defects4J remains the benchmark standard; Bugs.jar extends the scope to larger, more complex projects; Bears captures the “real-time” debugging that happens in CI pipelines; ManySStuBs4J provides the scale needed for training data-hungry neural networks. Beyond these, researchers have also built specialized datasets such as QuixBugs for small algorithmic problems, Code flaws for C programming contest bugs [24], and

BugSwarm for paired failing/passing builds packaged in Docker containers. A recent survey even identified more than 130 datasets, covering areas from security vulnerabilities to performance defects. But in modern literature, Defects4J, Bugs.jar, Bears, and ManySStuBs4J remain the most widely used and influential [9].

Benchmark datasets are very important for APR for finding bugs using Deep Learning (DL) and LLM. They tell you how to train, test, and compare models. Widely used benchmarks like Defects4J, Bugs.jar, Bears, and ManySStuBs4J have proven very important in moving the field ahead by making it easier to do experiments over and over again and evaluate them in a consistent way. The dataset utilized has a big effect on performance claims, generalizability, and how reliable the outcomes of tests are. Defects4J is still the most used benchmark for APR research since it can be reproduced and has clear test suites. It gives you actual Java bugs and tests that always pass or fail. Even if they are good things, the fact that they are so common in literature creates a big bias in evaluations. Several DL and LLM repair algorithms perform well on Defects4J, but their performance drops sharply when evaluated on other benchmarks. This observation suggests that some models may capture test-suite or dataset-specific traits rather than learning repair processes that generalize to real-world scenarios. This could make them seem more efficient than they really are. To get over these limitations, Bugs.jar and Bears were included to make the projects more varied and realistic. To test scalability, issues.jar adds more complex Apache projects to the test area. Bears, on the other hand, gather problems that happen when continuous integration fails, which reflects how development is done now. On the other hand, both datasets are challenging to recreate because of challenges with managing dependencies and setting up the environment. This might stop large-scale comparative research. ManySStuBs4J gives you a big collection of small bug-fix pairings that were taken from thousands of Java projects. It focuses on scalability instead. This dataset can help a lot with common syntactic and semantic bug patterns and training neural models that need a lot of data. Because it doesn't have execution-based validation, it can't tell the difference between syntactically reasonable changes and actually proper changes that keep runtime behavior the same. This makes it less effective for checking semantic correctness.

Table 1. Widely used datasets for bug detection and repair

Dataset	Language	Size / Bugs	Key features	Reference
Defects4J	Java	835 bugs, 17 projects	Real, reproducible bugs with developer-written tests	[15, 16]
Bugs.jar	Java	1158 bugs, 8 projects	Industrial-scale projects (Apache)	[18, 19, 20]
Bears	Java	251 bugs, 72 projects	Derived from Continuous Integration / Continuous Deployment(CI/CD) failures on GitHub	[21]
ManySStuBs4J	Java	153,652 bug-fix pairs	"Stupid" small bugs, ideal for ML	[22]
Codeflaws	C	3,902 programs	Contest-based buggy/fixed C code	[25, 26, 27]
QuixBugs	Multiple	40 small programs	Algorithmic bugs across 10 languages	[18, 28, 29]
BugSwarm	Multiple	3091 artifacts	Docker-based failing/passing builds	[29]

Overall, these datasets present complementary strengths and weaknesses. While controlled benchmarks such as Defects4J support reproducibility, larger and more diverse datasets improve realism at the cost of increased complexity and reduced execution guarantees. Given this trade-off, it becomes essential to assess DL-based and LLM-based bug detection and repair techniques using varied benchmarks to ensure realistic and broadly applicable outcomes.

Fig. 2 shows a graph comparing the most popular datasets used in the field of Bug detection & Program repair in terms of the number of bugs or fix pairs they contain.

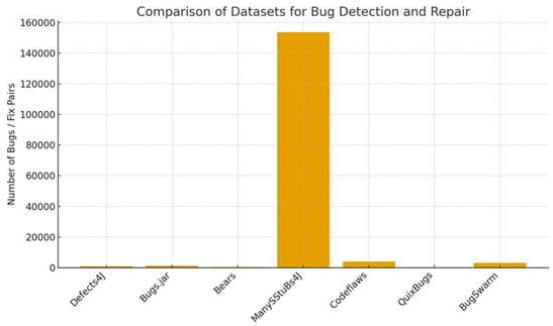

Fig. 2. Comparison of widely used datasets for bug detection and program repair

This figure illustrates the variation in dataset sizes commonly employed in DL-based and LLM-based bug detection and automated program repair research. ManySStuBs4J dominates with more than 153K bug-fix pairs, making it suitable for large-scale training, though it mostly contains simple, single-line bugs. Defects4J (835 bugs) and Bugs.jar (1158 bugs) remain benchmark standards due to their reproducibility and representation of industrial projects. Bears and BugSwarm capture real-world CI/CD failures and reproducible builds, reflecting practical debugging scenarios. Codeflaws and QuixBugs are smaller but valuable for language diversity and algorithmic problem testing. Overall, the figure highlights how dataset scale and characteristics influence the choice of benchmark for evaluating automated bug detection and repair systems. Below is a simplified textual description of the timeline:

- 2014 – Defects4J: The true beginning of the benchmark for evaluating APR systems;
- 2017 – Codeflaws: Added C language support for programming competitions;
- 2017 – QuixBugs: A small, multilingual, algorithmic resource;
- 2018 – Bugs.jar: Scales bugs for large Apache projects;
- 2019 – Bears: Collects bugs from GitHub associated with CI/CD reports;
- 2019 – BugSwarm: Provides reproducible bugs using Docker;
- 2020 – ManySStuBs4J: The largest source by volume (over 153,000 small bugs).

It can be visualized as a timeline from 2014 to 2020, and along the line the previous points appear in order of years, as shown in Fig. 3.

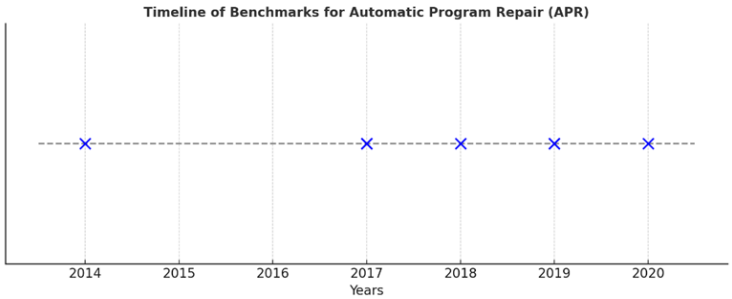

Fig. 3. Timeline of benchmarks used in APR evaluation
between 2014 and 2020, showing the gradual expansion from Defects4J to ManySStuBs4J

4. Literature reviews

In recent years, several studies have proposed automated bug triage systems to recommend appropriate developers for specific bug reports. These studies have employed various techniques to develop their proposed models. Some of these studies have either not used machine learning and Natural Language Processing (NLP) techniques or have used them minimally. In contrast, other studies have relied heavily on various machine learning and NLP techniques to develop their models. A h m e d and Q a i s e r [30], this paper proposes a Convolutional Neural Network (CNN) – Long Short-Term Memory (LSTM) model-based innovative method for classifying software bugs. When processing text data from the bug repositories, LSTM and CNN work together for extracting spatial patterns as well as sequential dependencies. The model's ability for accurately classify problems into different categories is demonstrated by experimental findings on datasets from open-source projects. Through addressing problems with bug prioritization and triage, the approach hopes to increase software stability and developer efficiency. The study's limitations include: the model's use in projects with noisy or missing bug descriptions may be limited by its reliance on text data. K h u r m a et al. [31] presented Island-Based Moth Flame Optimization (IsBMFO), dividing the population into sub-populations (islands) for independent evaluation, using IsBMFO for feature selection, Bayesian, k nearest neighbors, and Support Vector Machine (SVM) for classification. IsBMFO and SVM showed the best performance. R a h i m et al. [32] proposed a model for predicting defects in software, including three aspects: data pre-processing for filtering the noise and normalization of data, correlation-based feature extraction, and finally, the classification is done by using Naïve Bayes (NB) and Logistic Regression (LR) algorithms [15]. The accuracy result obtained was about 98.7% for NB, and it succeeded in reducing maintenance cost and complexity of code while detecting the software defects early. U d d i n et al. [33] proposed a novel model, Software Defect Prediction – BiLSTM + BERT (SDP-BB), which can address the deficiencies of using existing methods. SDP-BB employed Bi-directional Long Short-Term Memory (BiLSTM) networks and BERT-based semantic features to make up for deficiencies in the manual code feature models. Contrary to previous models, SDP-BB used semantic and context information from source code. The BiLSTM had taken the

context information from BERT by using embedded token vectors, and the attention mechanism highlighted important features. The method used data augmentation for augmented training. Competitive results on ten open-source projects, when compared with the state-of-the-art models, showed that our model outperformed its peers with regard to fault prediction as indicated by F1-score.

Table 2. Comparative summary of representative deep learning and LLM-based approaches for bug detection and automated program repair

Year	Authors	Contribution	Critical notes	Aspect	Dataset
2020	Ahmed and Qaiser [30]	Combining CNN and LSTM to extract spatial and sequential patterns from bug reports, improving bug classification accuracy	Evaluated only on bug report texts; lacks comparison with other DL methods	Bug triage using deep learning	Bug reports from Google Chrome, Mozilla Core, and Firefox projects
2021	Khurma et al. [31]	Introduced IsBMFO to select features; divides the population into islands and applies SVM for classification, achieving the best performance	Limited to static datasets; lacks cross-language evaluation	Feature selection and defect prediction	21 public datasets from the NASA MDP and PROMISE repositories
2021	Rahim et al. [32]	Proposed a framework with preprocessing, correlation-based feature selection, and classification using Naïve Bayes and logistic regression; reported ~98.7 % accuracy	High accuracy, partly due to a small, balanced dataset; not compared against modern DL models	Traditional ML for defect prediction	The PROMISE dataset was used for evaluation
2022	Uddin et al. [33]	Proposed SDP-BB, which leverages BilSTM and BERT-based semantic features with attention and data augmentation to predict defects more accurately than baselines	Performance depends on TF-IDF weights; it lacks quantitative accuracy figures and a large-scale evaluation	Defect prediction using semantic embeddings	Ten open-source projects
2022	Zaidi, Woo and Lee [34]	Developed a bug triage system based on a graph neural network over a heterogeneous graph of words and bug reports; used similarity metrics to weight edges	Precise numbers are not reported; applicability to other languages remains unclear	Bug triage with GNNs	Open-source datasets including Eclipse Platform, Eclipse JDT, and Mozilla Firefox (with variants requiring 0/5/10/20 reports per developer)
2024	Lin et al. [35]	The research proposes a new system called Mulpor for fixing software errors (APR)	Mulpor does not perform bug detection independently but relies on other pre-existing tools to locate suspicious lines	Automated program repair	Entire Defects4J dataset (835 bugs across 17 Java projects)
2024	Xia and Zhang [36]	Introduced a conversational LLM-based system that fixed 162 of 337 bugs in Defects4J for USD 0.42 per bug	Results restricted to Defects4J; cross-dataset performance unknown	Automated program repair with LLMs	Defects4J Versions 1.2 and 2.0
2025	Chen [37]	A system called SynergyBug was designed to integrate error detection and repair into a single system using BERT + GPT-3	Data bias, difficulty in verifying systems, need for powerful resources (GPU/TPU)	Integrating Bug detection and repair	Natural language reports

This study promoted the development of a software defect prediction methodology. Z a i d i, W o o and L e e [34] proposed a bug triage model using the Graph Neural Network (GNN) and a heterogeneous network. They utilized TF-IDF to weigh the connections between words and bug reports, similarity measures for word-to-word connections, and a simple 2-layer GNN for model learning. This was a system that suggested ten developers address the newly reported bugs. In their award-winning paper L i n et al. [35] introduce Mulpor, a novel approach for automated program repair (APR) that leverages multi-granularity patch generation to address a key limitation in existing systems: the tendency to operate at a fixed level of granularity (e.g., statement-level or token-level only). Instead, Mulpor dynamically selects the appropriate granularity-token, expression, or statement based on the nature of the bug, thereby enhancing both flexibility and repair accuracy. An interesting study in 2024, shown by X i a and Z h a n g [36], presented ChatRepair, which is considered a conversational large language model-based system for automated program repair on Java projects such as Defects4J. Contrary to the traditional generate-and-validate model, ChatRepair incorporates interactive feedback loops, in which failing test output execution and incorrect patches are fed back to the model to instruct the guide of next patch generation, whereas plausible patches undergo iterative refinement. This technique fixed 162 out of 337 Defects4J bugs in the real world (Version 1.2 and Version 2.0) at an overall price of $0.42 on average per bug, thus showing the efficiency and state-of-the-art repair ability as well. The primary contribution is in demonstrating that conversational LLMs can diversify the patch generation and overcome repetitive failures, although they are limited by dependence on test coverage, prompt design, and compatibility with multi-location or logic-heavy bugs. Through the use of a tool suite, dynamic prompting, and a finite-state controller, C h e n [37] propose a novel system called SynergyBug, which leverages deep learning to perform automated bug detection and repair. The system integrates BERT for contextual analysis and GPT-3 for generating repair solutions. Table 2 shows a comparison of previous studies.

The data suggests that LLM-based methods outperform traditional approaches but tend to specialize in Defects4J. While ChatRepair achieves high repair counts on Defects4J, cross-benchmark experiments show that approximately 21% of the 2141 bugs have been fixed through 11 APR tools over multiple datasets[7]. Such a difference indicates overfitting as well as underscores the requirement for broader benchmarks. The data evidence that LLM-based techniques are better than classic ones, but potentially focus on Defects4J. Although ChatRepair achieves high counts of repair on Defects4J, the results of cross-benchmarks are that only 21% of 2141 bugs were repaired with 11 APR tools on multiple datasets [7]. This discrepancy highlights overfitting and emphasizes the importance of broader benchmarking.

Evaluation metrics and validation frameworks. To evaluate DL and LLM based methods for automated bug identification and APR, you need to use measures that go beyond simple performance indicators. Learning-based systems can make syntactically acceptable patches and pass test suites, but they might not necessarily be semantically right or be able to generalize to a wide range of projects [38, 39, 40]. This is different from how software engineering is usually done. So, evaluation tools

are quite important for figuring out how reliable and effective these procedures [40, 41]. F1-score, Accuracy, Precision, and Recall are some of the most common metrics used to find bugs. When there are a lot of bad instances in a dataset, accuracy, which measures the percentage of correct predictions, might be misleading. Precision is very important in bug localization and repair procedures since it shows how likely it is that a predicted fault or provided fix is correct [42]. However, high precision alone is insufficient if the system fails to identify a substantial number of actual bugs. Recall addresses this limitation by measuring the proportion of real defects that are successfully detected, while the F1-score provides a balanced view by combining precision and recall [40]. More steps are needed to check how well automatic program fixes work. The Patch Success Rate is a standard way to quantify how many of the patches that were made pass all of the tests [39]. This metric does not guarantee semantic correctness because many legitimate patches might overfit the test suite; nonetheless, it does provide a realistic indicator of repair efficacy. The need for more strict validation standards is evidenced by the distinction between correct patches and believable patches in several studies [40]. Validation frameworks have a big effect on evaluation findings, maybe even more than the choice of measures. Most APR systems need test suites to be validated; however, these suites have problems with coverage and the quality of tests. Models that are only evaluated on benchmarks may work well in the lab, but they won't work in the real world. Consequently, essential methodologies for assessing generalizability and robustness encompass cross-project evaluation and multi-benchmark testing [39, 41, 43]. There exists a significant yet unexamined facet of validation pertaining to people. Even when automated tools get strong quantitative results, developer studies show that trust and productivity may go down because of inaccurate or hard-to-explain repair suggestions. It is becoming more and more clear that usability studies, explainability approaches, and developer involvement must all be part of evaluation pipelines if they are to be accepted in the real world [38, 42, 43].

Deep learning: A radical transformation in bug detection and resolution. Deep learning, a type of ML, has transformed several areas of AI, including computer vision, NLP, and autonomous systems, to name a few. Compared to traditional methods, DL is applied in software engineering for automated bug discovery, localization and fixing, achieving much higher accuracy and efficiency. The following are available with DL models: [9, 44]. Learning complex patterns from codebases: By learning from labeled data in large datasets, DL models can even spot bugs that have become defects but never seen before. Cross-language generalizability: DL could be cross-linguistically as well as cross-coding-style adaptable, dissimilar to the rule-based systems. Minimizing human intervention: Automated bug detection reduces human intervention in the process of debugging, and the developer can prioritize important tasks. Autonomous bug prediction and correction: Some of the models are capable of producing bug fixes and also suggest possible solutions based on past data. Application of DL to bug detection is in various forms like:

- Token-based analysis for source code and the application of CNNs [45].
- RNN and LSTM to learn sequence dependencies in code.

- Transformer-based models (e.g., CodeBERT, GPT-4) for contextual understanding of programming languages [46, 47].

DL techniques have transformed automated bug detection by allowing models to infer code structures and meanings from historical data, reducing the need for manual rule crafting [48]. These approaches have demonstrated improved scalability on large codebases; however, their effectiveness remains strongly influenced by code representation, model architecture, and evaluation settings.

Models and architectures of bug detection using deep learning. Automatic bug detection is an important component in contemporary software testing and quality assurance, and part of the effort to maintain reliable, secure, and effective software systems. With its scalability, intelligence, and high precision, DL has dramatically transformed software engineering, in particular, automated bug detection [48, 49]. Unlike traditional rule-based and heuristic techniques, DL models can automatically learn from large amounts of source code, execution traces, and debug logs to detect software defects and vulnerabilities [27, 29]. This section explores the various deep learning architectures, their particular uses in software debugging, and how deep learning is changing bug discovery. The traditional machine learning models ignore the contextual information and deep semantic features of the programs. Therefore, recent models. employed deep learning algorithms such as DBN, CNN, RNN, and LSTM to exploit prominent semantics and contextual information while making software defect predictions [33, 50]. Fig. 4 shows the classification of models of bug detection using deep learning.

Conventional bug detection techniques. Effective bug detection requires an appropriate numerical representation of source code, typically in the form of embeddings. Depending on the task, representations may vary in granularity, ranging from token-level embeddings for code completion to method-level or file-level embeddings for defect prediction [45]. Traditional approaches often rely on hand-crafted features derived from code metrics such as size, complexity, and change history [51]. However, these features frequently fail to capture program semantics, as structurally similar code fragments may implement different functionalities. Sequence-based representations using characters or tokens [52] and tree-based representations using Abstract Syntax Trees (ASTs) [53] partially address this limitation, but handcrafted feature methods generally remain insufficient for semantic understanding [51]. Consequently, modern approaches favor DL models that automatically learn structural and semantic features directly from code [54, 55].

Token-based models. Token-based models represent the earliest DL approaches for bug detection, treating code as sequences of lexical tokens and applying CNNs or RNNs to identify anomalies. A representative example is DeepBugs [56], which detects errors such as incorrect operators or swapped function arguments by learning embeddings of identifier names. These models are computationally efficient and require minimal preprocessing but struggle to capture deeper semantic relationships such as dataflow or control-flow dependencies. As a result, they have largely been surpassed by graph- and transformer-based methods, though they remain valuable for baseline analysis [57, 58].

Graph-based models. Graph Neural Networks (GNNs) operate on structured representations such as ASTs, Control-Flow Graphs (CFGs), and Data-Flow Graphs (DFGs), enabling models to reason about program semantics and intraprocedural dependencies [59]. Systems such as Devign demonstrated substantial improvements in accuracy and F1-score over prior methods, achieving strong performance on real-world vulnerabilities [60, 61]. Similarly, DeepWukong applied deep GNNs for vulnerability detection with higher accuracy than token-based models [62]. Despite their effectiveness, graph-based approaches incur high computational cost and scalability challenges due to graph construction and processing overhead, limiting their applicability to large industrial codebases [63].

Transformer-based models. Transformer architectures have become dominant in code intelligence due to their ability to capture long-range dependencies through self-attention. Pre-trained models such as CodeBERT, GraphCodeBERT, and CodeT5 have achieved state-of-the-art results in bug detection and localization tasks [64-66, 68]. For example, LineVul significantly outperformed baseline models in accuracy and inspection effort, demonstrating the effectiveness of transformer-based representations [63]. While transformers offer strong scalability and generalization, especially when pre-trained on large code corpora, they require substantial computational resources and may overfit benchmark datasets, raising concerns about real-world robustness [69].

Large language models and hybrid approaches. Large Language Models (LLMs), including Codex, GPT-3.5, and GPT-4, represent a recent paradigm shift in bug detection [70]. These models support zero-shot and few-shot detection across multiple languages and frameworks without task-specific fine-tuning. For instance, CrossProbe demonstrated cross-framework bug detection by transferring knowledge between deep learning libraries [5]. Hybrid systems such as SynergyBug [38] further combine LLM reasoning with retrieval-based mechanisms, integrating semantic understanding with concrete bug-fix examples. Despite their adaptability and generalization capabilities, LLM-based approaches often suffer from high false-positive rates and limited explainability, highlighting the need for further research on trust and interpretability to support industrial adoption.

Model architectures. DL architectures like Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTM) networks, which are sequence models [32, 48], were the first to try to solve bug detection and defect prediction concerns. These models are good at finding patterns in certain areas, but they don't function well when it comes to long-range linkages and scalability. CNNs were able to make things more efficient by finding patterns in individual tokens, but they couldn't properly show the structure of programs on a global scale [52]. Graph Neural Networks (GNNs) are a big step forward since they can work directly with Application State Trees (ASTs) and program graphs. Global Neural Networks (GNNs) significantly improve the precision of defect localization and classification by enabling relational reasoning regarding program structure through the dissemination of information across nodes and edges [35, 63, 64, 66, 67]. GNN-based methods might not work well with huge software systems since they might not be able to handle the size, and sometimes need complicated preprocessing steps.

Recently, the transformer-based design has become the norm for finding faults. Using self-attention processes, transformers may understand contextual links and long-range relationships without having to develop graphs. Two pre-trained code models, CodeBERT and CodeT5, have pushed this idea even further by making it easier to learn from huge code collections, which has resulted in big performance improvements with sparse labelled data [9, 68, 71].

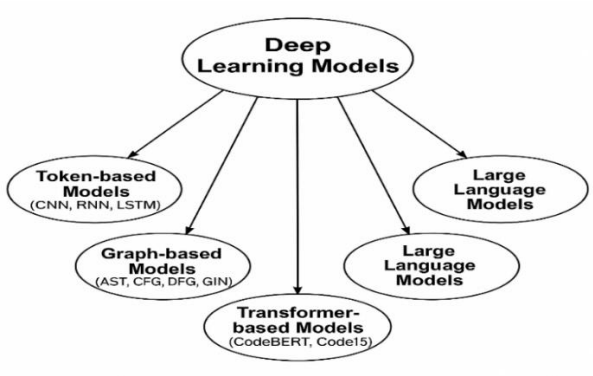

Fig. 4. Taxonomy of Models of bug detection

Code representation for bug detection. Most of the early deep learning bug detection systems employed token-based representations, which meant that the source code was broken down into a sequence of lexical tokens. These kinds of representations are easy to comprehend and apply in different languages, but they don't always work well for modelling complicated dependencies or interactions across control flows because they don't take into account the semantic and structural links that are common in programming languages [32, 33, 56, 60]. Later studies developed hybrid program graphs, ASTs, and CFGs as structure-aware representations to address these deficiencies. By keeping the code's logical and structural qualities intact, these representations make it easier for the model to examine and understand how programs work. Combining deep learning models with graph-based representations has shown to work better at finding difficulties with data dependence, control flow, and conditional logic [35, 63, 64, 66]. The taxonomy does not fully encompass new hybrid, agent-based, and multi-modal methodologies, but it does include the most important representation and modelling paradigms. These methods are interesting ways to study in the future.

Bug localization and granularity. There are several levels of detail for bug detection systems, such as files, methods, and lines. Coarse-grained detection makes processes simpler to follow and models easier to grasp. However, fine-grained localization may be more useful for developers since it shows exactly where issues occur. As a result, some DL-based approaches integrate coarse detection with more precise localization algorithms using hierarchical or multi-stage pipelines [47, 49, 65, 69].

Limitations and open challenges. Even though there have been big improvements, there are still problems with DL-based bug detection. Because broken

instances are generally just a small part of the codebase, they can throw off model predictions [36, 48], which is why dataset imbalance is such a big problem. It is hard to apply what you learnt about how well a model worked in one project or area to another, especially when you used the same criteria for training and testing [97]. There have been recent attempts to create debugging solutions that are explainable and focused on people [40, 93, 96]. This is because the lack of explainability in many deep learning models makes developers less confident and makes it harder for them to be used in industrial settings.

Types of bugs detectable by DL and LLMs. In addition to architectural advances, DL-based systems have been evaluated across a wide spectrum of bug types. These range from simple lexical anomalies to complex concurrency and security vulnerabilities. Table 3 summarizes categories of errors that modern DL and LLM-driven systems are capable of detecting. Fig. 5 shows the classification of software errors.

Table 3. Types of Bugs detectable by DL and LLM-based systems

Bug category	Example errors	Detection systems
Lexical/Syntactic	Typos, swapped args, wrong ops	DeepBugs [58], ManySStuBs4J [20]
Semantic/Logic	Off-by-one, wrong conditionals	SequenceR [72], CoCoNuT [74]
Concurrency	Deadlocks, data races	Devign [62], DeepWukong [64]
Security vulnerabilities	Buffer overflow, Use After Free (UAF), SQLInjection (SQLi)	LineVul [63], Devign [62]
Configuration	Build misconfigs, dep errors	Bears dataset [20, 29]
Domain-specific	ML bugs, Web API misuse	CrossProbe [5]

The classifications presented in this section are not intended to represent an exhaustive taxonomy of all bug detection techniques. Instead, they are derived through a structured synthesis of dominant paradigms reported in recent literature, aiming to capture the most influential and widely adopted approaches in deep learning-based bug detection. Token-based, graph-based, transformer-based, and LLM-driven models collectively reflect the evolution of the field; however, emerging directions such as agent-based debugging, execution-aware detection, and multi-modal representations are not yet fully integrated into current classification schemes. While the proposed categorization effectively highlights methodological trends and comparative strengths, it remains limited by the scope of existing benchmarks and evaluation practices. Future classifications could be enriched by incorporating execution traces, reinforcement learning signals, and human-in-the-loop feedback, which would enable a more comprehensive understanding of real-world debugging scenarios and improve generalization beyond controlled datasets.

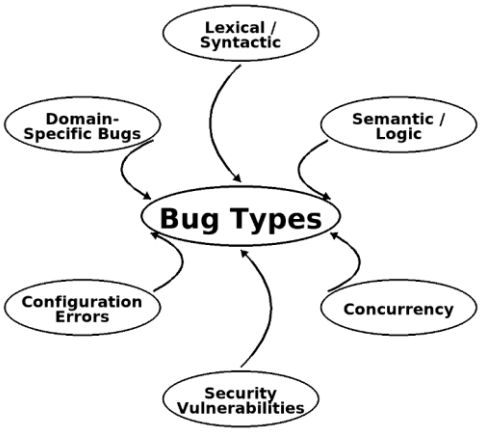

Fig. 5. Taxonomy of bug types handled by DL-based detection systems

Automated program repair using deep learning. APR seeks to generate patches that correct defective software with minimal human intervention. APR has traditionally relied on heuristics (such as genetic programming) or template-based systems [72]. Fig. 6 illustrates the most important models used in software bug repair. With the emergence of DL and LLM programs, APR has shifted toward learning-based approaches that can be generalized to various types of bugs.

6.7. Repair paradigms

Template-Based APR. Predefined transformation patterns fix known bug classes. TBar, with 104 templates, performs strongly on Defects4J but cannot handle novel errors and may fix as few as 10-15% of bugs on other datasets [73]. Template-based repair applies predefined transformation rules to fix known bug categories. TBar [76]. For example, it revisits template-based APR with a curated set of fix templates, achieving strong results on Defects4J. The strength of template-based APR lies in explainability: developers can easily interpret generated patches. However, its limitations are significant; it can only repair defects covered by existing templates, making it brittle when faced with novel or domain-specific bugs.

Neural Sequence-to-Sequence Repair. Seq2Seq models treat repair as translation. SequenceR was trained on 35,578 examples and evaluated on 75 one-line Defects4J bugs. It generated patches for 58 bugs, with 53 having at least one compilable patch, 19 yielding at least one plausible patch, and only 14 were correctly fixed [70]. Of the 2321 generated patches, 761 compiled. In total, only 61 passed the tests, and finally, only 18 were correct ≈, yielding an approximate 30 % correctness rate [70], indicating that passing the test suites does not ensure semantic correctness. Subsequently, CoCoNuT [74] enhanced this by combining several Seq2Seq models that consider various contexts. Seq2Seq repair can handle bug classes other than template coverage, such as logic errors (off-by-one errors, wrong conditionals) and missing API calls. However, one common issue with these tools is the synthesis of plausible yet incorrect patches – those that pass past tests but not necessarily ones that address the true fault [75, 76].

6.8. LLM-Based APR

LLMs revolutionize APR. ChatRepair uses conversational prompts and test feedback to fix 162 of 337 Defects4J bugs at USD 0.42 [77] each. AprMcts combines GPT-3.5 with a Monte-Carlo tree search, fixing 201 of 835 bugs and adding 30-37 extra fixes for other models while halving cost [78]. RepairAgent employs an autonomous agent to plan, generate, and validate patches, fixing 164 bugs and adding 39 new fixes [79]. Despite these advances, performance drops sharply on other datasets, and tests must be comprehensive. Large Language Models such as Codex [80] and GPT-4 [81] have redefined the scope of APR. Unlike earlier neural systems requiring fine-tuning, LLMs demonstrate zero-shot repair [82], generating patches for unfamiliar bug types without retraining. Studies show that Codex and GPT-4 outperform state-of-the-art APR tools on Defects4J and Bugs.jar, both in plausible and correct patch rates.

Retrieval-augmented repair. Retrieval-augmented systems enhance APR by grounding LLM patch generation in historical bug–fix examples. Tensor Guard [83] exemplifies this approach, retrieving domain-specific bug–fix pairs for ML frameworks before prompting GPT-based repair. This improves accuracy, decreases hallucinations, and enhances trust in generated patches.

6.9. Hybrid approaches

Hybrid systems combine templates, LLMs, and symbolic analysis. GIANTREPAIR employs LLM-generated patch skeletons refined in a context-sensitive manner. It increased the number of correct fixes from 43 to 53 on Defects4J V1. 2 (≈23 %) and 45-53 on V2. 0 (≈17 %). Advances of single LLMs were as high as 31% [85]. Yet, it needs precise fault localization and has so far been evaluated only on Defects4J. Hybrid repair systems combine more than one paradigm. RepairGPT adopts template rules, neural Seq2Seq models, and symbolic reasoning. This multi-pronged strategy leads to resilience across multiple bug classes. Nevertheless, it is computationally expensive and hard to implement hybrid systems [85].

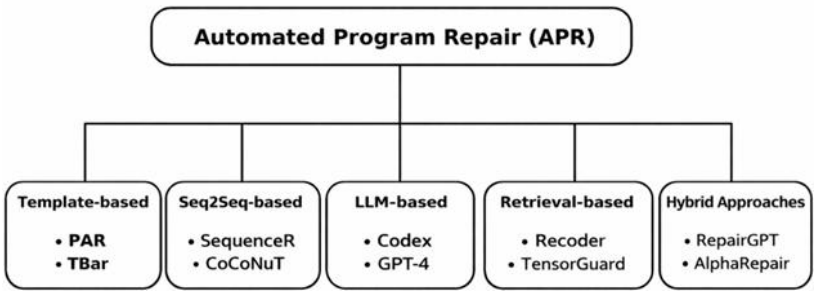

Fig. 6. Models used to repair software bugs

6.10. Types of Bugs repairable by DL and LLMs

There is a considerable overlap between the APR scope and the categories of bug detection, although practical success varies by bug type. Table 4 types of Bugs repairable by DL and LLM.

Table 4. Types of Bugs repairable by DL-based and LLM-based APR systems

Bug category	Example fixes	APR systems
Lexical/Syntactic	Correct swapped args, fix typos, replace wrong operators	SequenceR [72], Codex [36]
Semantic/Logic	Fix off-by-one errors, repair wrong conditionals, and insert missing returns	CoCoNuT [74], GPT-4 [81]
Concurrency	Limited (deadlocks, race conditions remain unsolved)	Hybrid attempts [85]
Configuration	Resolve build errors, dependency mismatches	Bears [20], BugSwarm [29] + LLM repair
Domain-specific	Fix incorrect API calls in ML frameworks	CrossProbe [5]

Fig. 7 shows the standard workflow of automatic program repair techniques. It starts with bug detection, where the defective code is discovered using ML or DL models. From this, it proceeds to bug localization, further pinpointing where the defect occurs. Second, patch generation is the process of making a candidate fix by rule-based, neural-based, or LLM-based methods. A validation step that applies the patch to the program's test suite and verifies its correctness before integration.

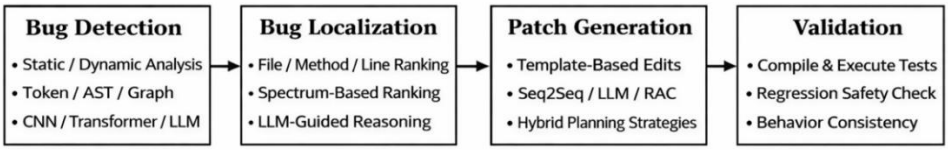

Fig. 7. Workflow automated program repair

The repair paradigms discussed in this section reflect the most representative approaches reported in contemporary APR research, including template-based systems, neural sequence-to-sequence models, LLM-based repair, and hybrid frameworks. These categories are synthesized from prior studies and surveys rather than proposed as a definitive or complete taxonomy. Although they capture the primary directions of APR development, they do not fully encompass recent advances such as autonomous repair agents, retrieval-augmented reasoning, and multi-location patch synthesis. Moreover, current classifications remain closely tied to benchmark-centric evaluation, particularly Defects4J, which may obscure limitations related to scalability, semantic correctness, and cross-project robustness. Future extensions of APR classifications should therefore emphasize hybrid and agent-based systems that integrate program analysis, execution feedback, and explainable decision-making to better reflect realistic software development and maintenance workflows.

Evaluation and Validation. Evaluation and validation constitute fundamental components in assessing the effectiveness, robustness, and generalizability of DL-based and LLM-based approaches for bug detection and automated program repair. In the context of this survey, evaluation does not refer to the validation of the authors' own experimental results, but rather to a systematic analysis of the evaluation methodologies, metrics, and validation practices reported across the literature. This section synthesizes how existing studies assess detection accuracy,

repair correctness, generalization across projects, and human usability, highlighting both commonly adopted practices and their inherent limitations [36]. Fig. 8 shows the evaluation pipeline based deep learning.

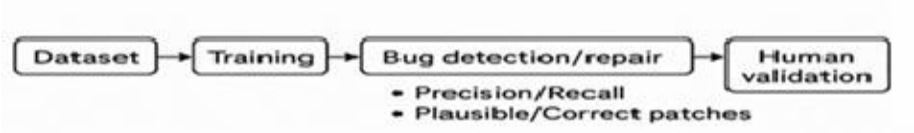

Fig. 8. Evaluation pipeline

7.1. Bug detection evaluation

Bug detection systems are typically evaluated using standard machine learning metrics. Precision, recall, and F1-score measure overall detection quality. Mean Average Precision (MAP) and Top-K recall are used in ranking scenarios, where the goal is to prioritize the most suspicious lines or functions. Performance is measured by precision, recall, and F1; ranking tasks use MAP or Top-K recall. LineVul illustrates progress: Top-10 accuracy of 0.65, median Inspected Functions Average (IFA) of 1 (versus 3-4 for baselines), Effort@ 20% Recall of 0.75, and Recall@ 1% Lines Of Code (LOC) of 0.24 [63] evaluates vulnerability detection at line granularity, reporting Top-1, Top-5, and Top-10 accuracy. Graph-based systems such as Devign [62] report AUC to demonstrate ranking capability.

7.2. Repair evaluation

APR research distinguishes plausible patches (passing tests) from correct patches (semantically equivalent). SEQUENCER demonstrates that only ≈30% of plausible patches are truly correct [70]. Tools like DiffTGen generate additional tests to filter overfitting patches. Machine learning classifiers can predict patch correctness without executing tests [86]. In APR, evaluation differentiates between plausible patches (those that pass all available test cases) and correct patches (those semantically equivalent to developer fixes). This distinction is of high importance since various systems achieve low rates of correctness, and high plausible rates [75].

7.3. Generalization and cross-project testing

Generalization is still poor: fixing tools that repair ≈47% of Defects4J bugs can repair only 10-30% of other benchmarks [7]. A reproducibility study showed that 62.6% of BugSwarm artefacts within 13 months broke, and after dependency isolation, the reproducibility reached more than 95% [87]. These results underline the importance of an isolated environment and multiple benchmarks. Generalization is still a weakness in many DL models. Tufano et al. [85] discovered that models learnt on Defects4J did not generalize to Bugs.jar or Bears. Cross-project evaluation demonstrates benchmark overfitting and thus a lack of real-world adoption that motivates the need for broader training corpora and more diverse benchmarks [80].

Human-centered validation. Developers benefit from tools driven by AI. Vaithilingam, Zhang and Glassman [87] carried out user studies highlighting that productivity is boosted by correct AI suggestions; on the other hand, incorrect ones harm as well as mislead efficiency. Thus, evaluation should

incorporate human factors, such as trust calibration, usability studies, and explainability metrics.

Challenges, limitations, and open issues. DL-based and LLM-based methods have made a lot of progress in automatically finding bugs and fixing software, but there are still certain basic challenges that need to be solved. These limitations underscore significant domains for more research and constrain their practical implementation. We are mostly concerned with the bias and representativeness of the dataset. The results from many well-known benchmarks only work for the Java ecosystem and a small number of open-source projects. This means they don't work for other projects, domains, or programming styles. It has been previously demonstrated that models trained on homogeneous datasets exhibit commendable performance in benchmarks but significantly worse results when used in disparate projects [17, 20-22]. Recent studies have reinforced the notion that dataset bias complicates equitable technique comparison by diminishing their external validity and reproducibility [89, 91]. Problems with test suite appropriateness and evaluation bias are also big problems. When all the tests in a test suite pass, patches are deemed real. This is a standard way for APR systems to check patches. But as a number of real-world investigations have shown, largely because of test-suite overfitting, a significant number of these patches are plausible but semantically erroneous [40, 44]. Both learning-based and LLM-driven repair systems frequently acknowledge this deficiency, which continues to be a significant obstacle to efficient assessment [92, 93, 97].

Computational costs and scalability problems make practical adoption much harder. It takes a lot of CPU effort to train and operate transformer-based models, whereas graph-based models that employ ASTs or program graphs usually need extensive preprocessing steps. Even if pre-trained models fix the problem of not having enough data, there are still problems with using them in continuous integration or large-scale settings because of delays and a lack of resources [61, 70, 94]. These limits become extremely clear when you work on more than one project or in more than one language [95]. People don't trust developers and can't explain things, which is a related and developing problem. Many DL-based and LLM-based systems don't make it clear why they find particular problems or come up with specific remedies. Even when performance metrics are stated as high, human-centered research indicates that engineers are less inclined to utilize automated debugging tools if the outcomes lack transparency or justification [40, 93]. Recent opinions on reliable software engineering using LLMs [96] say that explainability is no longer considered a desirable but obligatory requirement for industry acceptance. Lastly, there are still concerns that need to be solved about how to make benchmarking consistent and repeatable. It is challenging to derive reliable findings from disparate studies owing to discrepancies in dataset versions, experimental methodologies, and evaluation standards. Ongoing efforts to provide uniform standards and reproducible procedures have highlighted persistent inconsistencies in experimental execution and data reporting [84]. Improvements in finding and fixing defects using DL and LLM might be little steps instead of big steps if methods aren't clearer and more consistent [15, 92].

Summary of open issues. To advance the field, it is imperative to address the following unresolved issues highlighted by the aforementioned challenges: Some important goals are to build scalable and resource-efficient architectures, add explainability and human-centered validation to debugging tools, make reproducibility better by using standardized benchmarks and clear reporting, make datasets more diverse and representative to improve generalization, and create evaluation frameworks that take semantic correctness into account and lower test-suite overfitting. To transition DL-based and LLM-based bug detection and repair systems from reliable research environments to practical software development scenarios, it is essential to tackle these outstanding issues.

Conclusion and future directions. This study looked closely at automated bug finding and software repair methods that use DL and LLMs. The paper highlighted substantial progress achieved in the past and the persistent challenges that impede practical implementation, through an exhaustive examination of contemporary research trends, benchmark datasets, model architectures, and evaluation methodologies. The study clearly indicated a shift from sequence-driven and token-based models to structure-aware representations and transformer-based architecture. Pre-trained code models and graph neural networks have substantially improved the success rates of repairs and the accuracy of detections. However, the effectiveness of these models is still extremely dependent on the dataset's characteristics, the assessment techniques used, and the models' ability to generalize. For example, true semantic correctness and cross-project resilience are hidden by the widespread use of test suites that are just for benchmarks and datasets that are all the same. This study has shown that new ideas for conversational repair, agent-based debugging, zero-shot or few-shot learning, and LLM-based techniques are becoming more popular. Even though these technologies have promising performance, other issues need to be carefully thought about before they are used in software development processes. These include the cost of computing, the capacity to repeat results, the ability to explain results, and the dependability of technologies. Future research should concentrate on diverse and representative benchmarks, assessment systems that directly address semantic accuracy and generalization, and scalable solutions suitable for industrial applications. If automated debugging systems want developers to trust and use them, they need to use human-centered validation and eXplainable Artificial Intelligence (XAI) methodologies. A more practical and reliable way to fix programs automatically could be available with a hybrid approach that combines program analysis with DL and LLM-based models. In short, DL-based and LLM-based approaches have changed how bugs are found and fixed automatically, but their long-term impact will depend on how well they can solve basic issues with evaluation, generalization, and usability. This project will combine existing information and identify significant outstanding topics to lay the framework for future breakthroughs in intelligent software maintenance.

Discussion of research questions. This section summarizes and consolidates the answers to the research questions posed in Section 1, providing a unified perspective on the main findings of this survey.

RQ1. Current criteria for evaluating DL and LLM bug detection and repair systems are helpful, but there isn't a single score that encompasses semantic correctness, generalizability, and practical usability well enough. For this reason, a thorough and trustworthy review can only be achieved by combining metrics focused on precision with repair success measurements, cross-project validation, and human-centered evaluation.

RQ2. Defects4J, Bugs.jar, Bears, and ManySStuBs4J are the most significant benchmarks for automatic program repair and defect discovery based on DL and LLM. Defects4J does provide for controlled evaluation and repeatability, but it tends to generalize and doesn't have a lot of variety. ManySStuBs4J have scales that are good for training, but it doesn't have dynamic validation. Bugs.jar and Bears, on the other hand, make projects more realistic and varied, but they also make configuration more complicated. These trade-offs show how important it is to use several benchmarks to get reliable and useful findings. Benchmark dataset study shows that common datasets provide uniform evaluation settings, but their restricted range of projects and lack of diversity make it hard to generalize. To accurately assess learning-based methodologies for bug discovery and remediation, it is essential to establish more representative and cross-project standards.

RQ3. Deep learning has come a long way since token-based and sequence models. Now, it uses structure-aware graph models and transformer-based architectures to find bugs. There is a need for hybrid representations and stricter testing practices since no one technique entirely covers generalization, scalability, and explainability, even though each paradigm has its own strengths.

RQ4. The results show that huge language models have substantially expanded the options for automated program repair. Some of these options are conversational repair, zero-shot patch creation, and agent-based debugging workflows.

RQ5. They still have a long way to go before they can be really useful because of problems like high computational costs, lack of explainability, and the chance of making fixes that are plausible but semantically incorrect. Hybrid approaches are needed to fix these problems and make sure that the remedy is reliable. These methods should mix program analysis with execution feedback and techniques that can be understood.

References

1. Fang, C., Q. Zhang, W. Sun, Y. Ma, Z. Chen. Survey of Learning-Based Automated Program Repair. – ACM Trans. Software Eng. Methodol., Vol. **33**, 2023, No 2, pp. 1-69.
2. Ye, H., M. Monperrus. Iter: Iterative Neural Repair for Multi-Location Patches. – In: Proc. of 46th IEEE/ACM Int. Conf. on Software Engineering (ICSE'24), 2024, pp. 1-13.
3. Kim, Y., Y. Park, S. Han, J. Yi. Enhancing the Efficiency of Automated Program Repair via Greybox Analysis. – In: Proc. of 39th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE'24), 2024, pp. 1719-1731.
4. Chen, X., D. Zhang, Y. Zhao, Z. Cui, C. Ni. Software Defect Number Prediction: Unsupervised vs Supervised Methods. – Inf. Software Technol., Vol. **106**, 2019, pp. 161-181.
5. Durieux, T., F. Madeiral, M. Martinez, R. Abreu. Empirical Review of Java Program Repair Tools: A Large-Scale Experiment on 2141 Bugs and 23,551 Repair Attempts. – In: Proc. of ESEC/FSE'2019, 2019, pp. 302-313.

6. R a n a, M. R. R., A. N a w a z, T. A l i, A. S. A l a t t a s, D. S. A b d E l m i n a a m. Sentiment Analysis of Product Reviews Using Transformer-Enhanced 1D-CNN and BiLSTM. – Cybernetics and Information Technologies, Vol. **24**, 2024, No 3, pp. 450-465.
7. F e n g, Z., D. G u o, D. T a n g, N. D u a n, X. F e n g, M. Z h o u, L. S h o u. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. – In: Findings of the Association for Computational Linguistics (EMNLP'2020), 2020, pp. 1536-1547.
8. J a b r w, S. K., Q. I. S a r h a n. A Systematic Survey on Large Language Models for Code Generation. – ARO – The Scientific Journal of Koya University, Vol. **13**, 2025, No 2, pp. 83-99.
9. D i k i c i, S., T. T. B i l g i n. Advancements in Automated Program Repair: A Comprehensive Review. – Knowledge and Information Systems, Vol. **67**, 2025, No 6, pp. 4737-4783.
10. Y i n, X., C. N i, S. W a n g, Z. L i, L. Z e n g, X. Y a n g. ThinkRepair: Self-Directed Automated Program Repair. – In: Proc. of 33rd ACM SIGSOFT Int. Symp. on Software Testing and Analysis (ISSTA'24), 2024, pp. 1274-1286.
11. K o n g, J., X. X i e, S. L i u. Demystifying Memorization in LLM-Based Program Repair via a General Hypothesis Testing Framework. – Proc. ACM Software Eng., Vol. **2**, 2025, No FSE, pp. 2712-2734.
12. G u a n, H., G. B a i, Y. L i u. CrossProbe: LLM-Empowered Cross-Project Bug Detection for Deep Learning Frameworks. – Proc. ACM Software Eng., Vol. **2**, 2025, No ISSTA, pp. 2430-2452.
13. F a r i d, A. B., E. M. F a t h y, A. S. E l d i n, L. A. A b d-E l m e g i d. Software Defect Prediction Using a Hybrid Model (CBIL) of CNN and Bi-LSTM. – PeerJ Comput. Sci., Vol. **7**, 2021, p. e739.
14. X u, W., C. H u a n g, S. G a o, S. S h a n g. LLM-Based Agents for Tool Learning: A Survey. – Data Sci. Eng., 2025, pp. 1-31.
15. S a a v e d r a, N., A. S i l v a, M. M o n p e r r u s. GitBug-Actions: Building Reproducible Bug-Fix Benchmarks with GitHub Actions. – In: Proc. ICSE Companion, 2024, pp. 1-5.
16. Y a n g, D., et al. Where Were the Repair Ingredients for Defects4J Bugs? Exploring the Impact of Repair Ingredient Retrieval on the Performance of 24 Program Repair Systems. – Empir. Software Eng., Vol. **26**, 2021, No 6, p. 122.
17. R a f i, M. N., A. R. C h e n, T.-H. P. C h e n, S. W a n g. Revisiting Defects4J for Fault Localization in Diverse Development Scenarios. – In: Proc. of 22nd IEEE/ACM Int. Conf. on Mining Software Repositories (MSR'25), 2025, pp. 63-75.
18. L o p e z-D u r a n, N., D. R o m e r o-O r g a n v i d e z, F. L. C r u z, D. B e n a v i d e s. Software Bug Report Dataset from Eclipse Projects. – Data in Brief, Vol. **62**, 2025, 112016.
19. F e r e n c, R., Z. T ó t h, G. L a d á n y i, I. S i k e t, T. G y i m ó t h y. A Public Unified Bug Dataset for Java and Its Assessment Regarding Metrics and Bug Prediction. – Software Qual. J., Vol. **28**, 2020, pp. 1447-1506.
20. S a h a, R. K., Y. L y u, W. L a m, H. Y o s h i d a, M. R. P r a s a d. Bugs.jar: A Large-Scale, Diverse Dataset of Real-World Java Bugs. – In: Proc. of 15th Int. Conf. on Mining Software Repositories (MSR'18), 2018, pp. 10-13.
21. M a d e i r a l, F., S. U r l i, M. M a i a, M. M o n p e r r u s. Bears: An Extensible Java Bug Benchmark for Automatic Program Repair Studies. – In: Proc. of 26th IEEE Int. Conf. on Software Analysis, Evolution and Reengineering (SANER'19), 2019, pp. 468-478.
22. K a r a m p a t s i s, R.-M., C. S u t t o n. How Often Do Single-Statement Bugs Occur? The Many SStuBs4J Dataset. – In: Proc. of 17th Int. Conf. on Mining Software Repositories (MSR'20), 2020, pp. 573-577.
23. A b d o l l a h p o u r, M. M., M. A s h t i a n i, F. B a k h s h i. Automatic Software Code Repair Using Deep Learning Techniques. – Software Qual. J., Vol. **32**, 2024, No 2, pp. 361-390.
24. L i, X., D. L i, M. Z h a o, W. E. W o n g, H. L i. Learning-Based Patch Overfitting Detection: A Survey. – J. Internet Technol., Vol. **26**, 2025, No 1, pp. 53-64
25. L e, X. B. D., F. T h u n g, D. L o, C. L e G o u e s. Overfitting in Semantics-Based Automated Program Repair. – Empirical Software Engineering, Vol. **23**, 2018, No 5, pp. 2941-2973.
26. L i u, K., et al. TrickyBugs: A Dataset of Corner-Case Bugs in Plausible Programs. – In: Proc. of 21st Int. Conf. on Mining Software Repositories (MSR'24), 2024, pp. 113-117.

27. T a n, S. H., J. Y i, S. M e c h t a e v, A. R o y c h o u d h u r y. Codeflaws: A Programming Competition Benchmark for Evaluating Automated Program Repair Tools. – In: Proc. of ICSE-Companion, 2017, pp. 180-182.
28. L i n, D., J. K o p p e l, A. C h e n, A. S o l a r-L e z a m a. QuixBugs: A Multi-Lingual Program Repair Benchmark Set Based on the Quixey Challenge. – In: Proc. of SPLASH Companion, 2017, pp. 55-56.
29. Y e, H., M. M a r t i n e z, T. D u r i e u x, M. M o n p e r r u s. A Comprehensive Study of Automatic Program Repair on the QuixBugs Benchmark. – J. Syst. Softw., Vol. **171**, 2021, 110825.
30. A h m e d, A., H. Q a i s e r. Bug Classification Using CNN-LSTM in Open-Source Software Systems. – Computer Science Review, Vol. **37**, Article 2020, 100234.
31. K h u r m a, R. A., H. A l s a w a l q a h, I. A l j a r a h, M. A. E l a z i z, R. D a m a š e v i č i u s. An Enhanced Evolutionary Software Defect Prediction Method Using Island Moth Flame Optimization. – Mathematics, Vol. **9**, 2021, No 15, 1722.
32. R a h i m, A., Z. H a y a t, M. A b b a s, A. R a h i m, M. A. R a h i m. Software Defect Prediction with Naïve Bayes Classifier. – In: Proc. of Int. Bhurban Conf. on Applied Sciences and Technologies (IBCAST'21), 2021, pp. 293-297.
33. U d d i n, M. N., B. L i, Z. A l i, P. K e f a l a s, I. K h a n, I. Z a d a. Software Defect Prediction Employing BiLSTM and BERT-Based Semantic Feature. – Software Comput., Vol. **26**, 2022, No 16, pp. 7877-7891.
34. Z a i d i, S. F. A., H. W o o, C.-G. L e e. A Graph Convolution Network-Based Bug Triage System to Learn Heterogeneous Graph Representation of Bug Reports. – IEEE Access, Vol. **10**, 2022, pp. 20677-20689.
35. L i n, B., R. W a n g, Z. S h e n, J. J i a n g, L. L i, X. G u. One Size Does Not Fit All: Multi-Granularity Patch Generation for Better Automated Program Repair. – In: Proc. of 46th International Conference on Software Engineering (ICSE'24), IEEE/ACM, 2024, pp. 1546-1557. DOI: 10.1145/3597503.3623406.
36. X i a, C. S., L. Z h a n g. Automated Program Repair via Conversation: Fixing 162 out of 337 Bugs for $0.42 Each Using ChatGPT. – In: Proc. of ISSTA'24, 2024, pp. 819-831.
37. C h e n, H. SynergyBug: A Deep Learning Approach to Autonomous Debugging and Code Remediation. – Scientific Reports, Vol. **15**, 2025, 24888. DOI: 10.1038/s41598-025-08226-5.
38. B o u z e n i a, I., P. D e v a n b u, M. P r a d e l. RepairAgent: An Autonomous LLM-Based Agent for Program Repair. – Proceedings of the ACM on Software Engineering, Vol. **2**, 2025, No ICSE, pp. 1-15.
39. A l-B a t a i n e h, O. I., L. M o o n e n, L. V i d z i u n a s. Extending the Range of Bugs that Automated Program Repair Can Handle. – J. Syst. Software, Vol. **209**, 2024, 111918.
40. L i u, K., et al. A Critical Review on the Evaluation of Automated Program Repair Systems. – J. Syst. Software, Vol. **171**, 2021, 110817.
41. O u y a n g, Y., J. Y a n g, L. Z h a n g. An Empirical Study on the Suitability of Test-Based Patch Acceptance Criteria. – ACM Trans. Software Eng. Methodol., Vol. **34**, 2025, No 4, pp. 1-28.
42. O y o-I t a, E., E. A. E d i m, A. O. O t i k o, D. E. I z u k i. Improving Model Performance for Software Defect Detection and Prediction Using an Ensemble Method and Cross-Validation Techniques. – Int. J. Sci. Res. Archive, Vol. **12**, 2024, No 2, 10.30574.
43. K a n g, S., J. W a n g, T. Z h a n g, G. M e n g, Z. W a n g. AutoSD: Explainable Automated Debugging with Large Language Models. – Empirical Software Engineering, Vol. **30**, 2025, No 2, pp. 1-32.
44. B e l l o, R.-W., S. J. T o b i. Software Bugs: Detection, Analysis, and Fixing. – Analysis and Fixing, 2023.
45. N i u, F., C. L i, K. L i u, X. X i a, D. L o. When Deep Learning Meets IR-Based Bug Localization: A Survey. – ACM Comput. Surv., Vol. **57**, 2025, No 11, pp. 1-41.
46. A k i m o v a, E. N., et al. A Survey on Software Defect Prediction Using Deep Learning. – Mathematics, Vol. **9**, 2021, No 11, 1180.
47. Z h a n g, H., Z. L i, J. L i, Z. J i n, G. L i. WELL: Applying Bug Detectors to Bug Localization via Weakly Supervised Learning. – J. Software: Evol. Process, Vol. **36**, 2024, No 9, e2669.
48. V i s w a n a d h a p a l l i, V. Automated Bug Detection and Resolution Using Deep Learning: A New Paradigm in Software Engineering. – Int. J. Eng. Comput. Sci., Vol. **13**, 2024, No 4.

49. P r a s a d, R. D., M. S r i v e n k a t e s h. Evaluating RNNs and Transformers for Code-Related Tasks, Including Bug Detection, Code Completion, and Summarization. – J. Theor. Appl. Inf. Technol., Vol. **102**, 2024, No 21.
50. K u k k a r, A., R. M o h a n a, A. N a y y a r, J. K i m, B.-G. K a n g, N. C h i l a m k u r t i. A Novel Deep-Learning-Based Bug Severity Classification Technique Using CNN and Random Forest with Boosting. – Sensors, Vol. **19**, 2019, No 13, 2964.
51. H o f f m a n, I., N. B r o o k s. Automated Bug Detection and Correction in Software Development Using Machine Learning. – Int. J. Adv. Comput. Theory Eng., Vol. **12**, 2023, No 1, pp. 15-21.
52. K e s a v a n, E. Software Bug Prediction Using Machine Learning Algorithms: An Empirical Study on Code Quality and Reliability. – Int. J. Innovations Sci. Eng. Manag., 2025, pp. 377-381.
53. A l b a t t a h, W., M. A l z a h r a n i. Software Defect Prediction Based on Machine Learning and Deep Learning Techniques: An Empirical Approach. – AI, Vol. **5**, 2024, No 4, pp. 1743-1758.
54. P a r v a t h y, R., M. T h u s h a r a. AST-Based and Token-Based Neural Networks for Source Code Classification: A Comparative Performance Analysis. – In: Proc. of 15th Int. Conf. on Computing Communication and Networking Technologies (ICCCNT'24), 2024, pp. 1-7.
55. Z h a n g, X., M. G u o, Z. C h e n, Y. Z h a n g, T. J u, B. X i a o. Exploring Extended Abstract Syntax Tree Encoding for Enhancing Code Vulnerability Detection. – SSRN 4951333.
56. L i, Z., D. Z o u, S. X u, H. J i n, Y. Z h u, Z. C h e n. SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities. – IEEE Trans. Dependable Secure Comput., Vol. **19**, 2021, No 4, pp. 2244-2258.
57. Z a i d i, S. M. H., M. K h a n, M. L a t i f, A. A k h t a r, S. K h a n. Use of Deep Learning in Early Software Bug Detection. – Mehran Univ. Res. J. Eng. Technol., Vol. **44**, 2025, No 3, pp. 141-151.
58. P r a d e l, M., K. S e n. DeepBugs: A Learning Approach to Name-Based Bug Detection. – Proc. ACM Program. Lang., Vol. **2**, 2018, No OOPSLA, pp. 1-25.
59. Y a n g, Y., X. X i a, D. L o, J. G r u n d y. A Survey on Deep Learning for Software Engineering. – ACM Comput. Surv., Vol. **54**, 2022, No 10s, pp. 1-73.
60. C a s e y, B., J. C. S a n t o s, G. P e r r y. A Survey of Source Code Representations for ML-Based Cybersecurity Tasks. – ACM Comput. Surv., Vol. **57**, 2025, No 8, pp. 1-41.
61. Z h a n g, J., et al. Detecting Condition-Related Bugs with a Control-Flow Graph Neural Network. – In: Proc. of ISSTA'23, 2023, pp. 1370-1382.
62. Z h o u, Y., S. L i u, J. S i o w, X. D u, Y. L i u. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. – NeurIPS, Vol. **32**, 2019.
63. F u, M., C. T a n t i t h a m t h a v o r n. LineVul: A Transformer-Based Line-Level Vulnerability Prediction. – In: Proc. of 19th Int. Conf. on Mining Software Repositories (MSR'22), 2022, pp. 608-620.
64. C h e n g, X., H. W a n g, J. H u a, G. X u, Y. S u i. DeepWukong: Statically Detecting Software Vulnerabilities Using a Deep Graph Neural Network. – ACM Trans. Softw. Eng. Methodol. (TOSEM), Vol. **30**, 2021, No 3, pp. 1-33.
65. D e K r a k e r, W., H. V r a n k e n, A. H o m m e r s o m. MultiGLICE: Combining Graph Neural Networks and Program Slicing for Multiclass Software Vulnerability Detection. – Computers, Vol. **14**, 2025, No 3, p. 98.
66. P a n, C., M. L u, B. X u. An Empirical Study on Software Defect Prediction Using the CodeBERT Model. – Appl. Sci., Vol. **11**, 2021, No 11, 4793.
67. L i, Y., et al. A Knowledge-Enhanced Large Language Model for Bug Localization. – Proc. ACM Softw. Eng., Vol. **2**, 2025, No FSE, pp. 1914-1936.
68. W a n g, Y., W. W a n g, S. J o t y, S. C. H o i. CodeT5: Identifier-Aware Unified Pre-Trained Encoder-Decoder Models for Code Understanding and Generation. – arXiv:2109.00859, 2021.
69. H u a n g, K., J. Z h a n g, X. B a o, X. W a n g, Y. L i u. Comprehensive Fine-Tuning of Large Language Models of Code for Automated Program Repair. – IEEE Trans. Software Eng., 2025.
70. V o k h r a n o v, I., B. B u l a k h. Transformer-Based Models Application for Bug Detection in Source Code. – Technol. Audit Prod. Reserves, Vol. **5**, 2024, No 2(79), pp. 6-15.
71. M o e n k s, N., P. P e n a v a, R. B u e t t n e r. A Systematic Literature Review of Large Language Model Applications in Industry. – IEEE Access, 2025.

72. Chen, Z., S. Kommrusch, M. Tufano, L.-N. Pouchet, D., Poshyvanyk, M., Monperrus. Sequencer: Sequence-to-Sequence Learning for End-to-End Program Repair. – IEEE Trans. Software Eng., Vol. **47**, 2019, No 9, pp. 1943-1959.
73. Vassilev, M., V. Vassilev, A. Penev. IDD – A Platform Enabling Differential Debugging. – Cybernetics and Information Technologies, Vol. **20**, 2020, No 1, pp. 29-43.
74. Lutellier, T., H. V. Pham, L. Pang, Y. Li, M. Wei, L. Tan. COCONUT: Combining Context-Aware Neural Translation Models Using an Ensemble for Program Repair. – In: Proc. of ISSTA'20, 2020, pp. 101-114.
75. Li, Y., S. Wang, T. N. Nguyen. DEAR: A Novel Deep Learning-Based Approach for Automated Program Repair. – In: Proc. of 44th Int. Conf. on Software Engineering (ICSE'22), 2022, pp. 511-523.
76. Liu, K., A. Koyuncu, D. Kim, T. F. Bissyandé. TBar: Revisiting Template-Based Automated Program Repair. – In: Proc. of ISSTA'19, 2019, pp. 31-42.
77. Huang, S., X. Zhou, S. Chin. Application of seq2seq Models on Code Correction. – Front. Artif. Intell., Vol. **4**, 2021, 590215.
78. DMonperrus, M. Sequencer: Sequence-to-Sequence Learning for End-to-End Program Repair. – IEEE Transactions on Software Engineering, Vol. **47**, 2019, No 9, pp. 1943-1959.
79. Hossain, S. B., et al. A Deep Dive into Large Language Models for Automated Bug Localization and Repair. – Proc. ACM Software Eng., Vol. **1**, 2024, No FSE, pp. 1471-1493.
80. Xia, C. S., L. Zhang. Less Training, More Repairing Please: Revisiting Automated Program Repair via Zero-Shot Learning. – In: Proc. of ESEC/FSE'22, 2022, pp. 959-971.
81. Li, F., J. Jiang, J. Sun, H. Zhang. Hybrid Automated Program Repair by Combining Large Language Models and Program Analysis. – ACM Trans. Software Eng. Methodol., Vol. **34**, 2025, No 7, pp. 1-28.
82. Jiang, N., T. Lutellier, L. Tan. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. – In: Proc. of ICSE'21, 2021, pp. 1161-1173.
83. Ye, H., M. Martinez, M. Monperrus. Automated Patch Assessment for Program Repair at Scale. – Empir. Software Eng., Vol. **26**, 2021, No 2, p. 20.
84. Zhu, H.-N., C. Rubio-González. On the Reproducibility of Software Defect Datasets. – In: Proc. of ICSE'23, 2023, pp. 2324-2335.
85. Tufano, M., C. Watson, G. Bavota, M. di Penta, M. White, D. Poshyvanyk. An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation. – ACM Trans. Software Eng. Methodol., Vol. **28**, 2019, No 4, pp. 1-29.
86. Renzullo, J., P. Reiter, W. Weimer, S. Forrest. Automated Program Repair: Emerging Trends Pose and Expose Problems for Benchmarks. – ACM Comput. Surv., Vol. **57**, 2025, No 8, pp. 1-18.
87. Vaithilingam, P., T. Zhang, E. L. Glassman. Expectation vs Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. – In: Proc. of CHI EA, 2022, pp. 1-7.
88. Ouyang, Y., J. Yang, L. Zhang. Benchmarking Automated Program Repair: An Extensive Study on Both Real-World and Artificial Bugs. – In: Proc. of ISSTA'24, 2024, pp. 440-452.
89. Ahmed, I., et al. Artificial Intelligence for Software Engineering: The Journey so far and the Road Ahead. – ACM Trans. Softw. Eng. Methodol., Vol. **34**, 2025, No 5, pp. 1-27.
90. Khati, D., Y. Liu, D. N. Palacio, Y. Zhang, D. Poshyvanyk. Mapping the Trust Terrain: LLMs in Software Engineering – Insights and Perspectives. – ACM Trans. Software Eng. Methodol., 2025.
91. Yang, D., Y. Lei, X. Mao, Y. Qi, X. Yi. Seeing the Whole Elephant: Systematically Understanding and Uncovering Evaluation Biases in Automated Program Repair. – ACM Trans. Software Eng. Methodol., Vol. **32**, 2023, No 3, pp. 1-37.
92. Bui, Q.-C., R. Paramitha, D.-L. Vu, F. Massacci, R. Scandariato. APR4Vul: An Empirical Study of Automatic Program Repair Techniques on Real-World Java Vulnerabilities. – Empir. Software Eng., Vol. **29**, 2024, No 1, 18.
93. Eladawy, H., C. Le Goues, Y. Brun. Automated Program Repair – What Is It Good for? Not Absolutely Nothing! – In: Proc. of ICSE'24, 2024, pp. 1-13.
94. Xu, P., B. Kuang, M. Su, A. Fu. Survey of Large-Language-Model-Based Automated Program Repair. – J. Comput. Res. Dev., Vol. **62**, 2025, No 8, pp. 2040-2057.

95. O s h i, H., J. C. S a n c h e z, S. G u l w a n i, V. L e, G. V e r b r u g g e n, I. R a d i č e k. Repair is Nearly Generation: Multilingual Program Repair with LLMs. – In: Proc. AAAI, Vol. **37**, 2023, No 4, pp. 5131-5140.
96. S o t t o-M a y o r, B., M. K a l e c h. A Survey on Transfer Learning for Cross-Project Defect Prediction. – IEEE Access, Vol. **12**, 2024, pp. 93398-93425.
97. Z e m i n, L., et al. An Empirical Study on the Suitability of Test-Based Patch Acceptance Criteria. – ACM Trans. Software Eng. Methodol., Vol. **34**, 2025, No 3, pp. 1-20.

Received: 11.11.2025, First revision: 26.12.2025, Second revision: 15.01.2026, Accepted: 20.01.2026