A Survey of Fault Localization, Program Repair, and Deep Learning in Software Debugging

www.surveyx.cn

Abstract

This survey paper examines the intersection of fault localization, program repair, and deep learning within software debugging, emphasizing the transformative role of artificial intelligence in modern software development. Automated Program Repair (APR) has been pivotal in reducing costs and developer effort by leveraging AI-driven techniques to identify and fix software bugs, thereby enhancing software reliability and maintainability. Deep learning models, particularly Large Language Models (LLMs), have significantly advanced APR by generating contextually relevant code patches, addressing complex bugs, and improving fault localization. Despite these advancements, traditional debugging methods face limitations, such as reliance on static analysis and the out-of-vocabulary problem, which deep learning techniques aim to overcome. The survey organizes its exploration into traditional and emerging methodologies, highlighting deep learning's impact on fault localization and APR, including enhancements in patch generation and evaluation. Challenges such as scalability, adaptability, and the need for comprehensive datasets persist, underscoring the necessity for innovative solutions. Future research directions include expanding training datasets, integrating AI techniques into continuous integration pipelines, and improving model generalization across diverse software environments. As deep learning continues to evolve, its integration into software debugging promises to offer sophisticated solutions to contemporary software development challenges, enhancing the accuracy, efficiency, and scalability of debugging processes.

1 Introduction

1.1 Significance of Automated Program Repair

Automated Program Repair (APR) has emerged as a crucial component of contemporary software development, significantly reducing costs and developer effort through data-driven, deep learning approaches [1]. The synergy between APR and AI techniques, particularly deep learning, has transformed bug identification and correction, alleviating the manual burdens associated with traditional debugging [2]. Deep learning models can autonomously generate bug-fixing patches, thereby enhancing software reliability and maintainability.

APR effectively addresses variable misuse bugs, often stemming from code copying errors and context misinterpretation [3]. By employing AI techniques, APR rectifies these issues, thereby improving code quality. Notably, APR has evolved to incorporate code review comments, exemplified by methods like Review4Repair, which enhance fix suggestions by leveraging insights typically overlooked in conventional approaches [4].

Recent advancements include the utilization of Large Language Models (LLMs) for generating precise and contextually relevant code patches [5]. These models tackle challenges related to dependent changes across multiple statements, as demonstrated by DEAR, which seeks to overcome the limitations of existing deep learning-based APR systems [6].

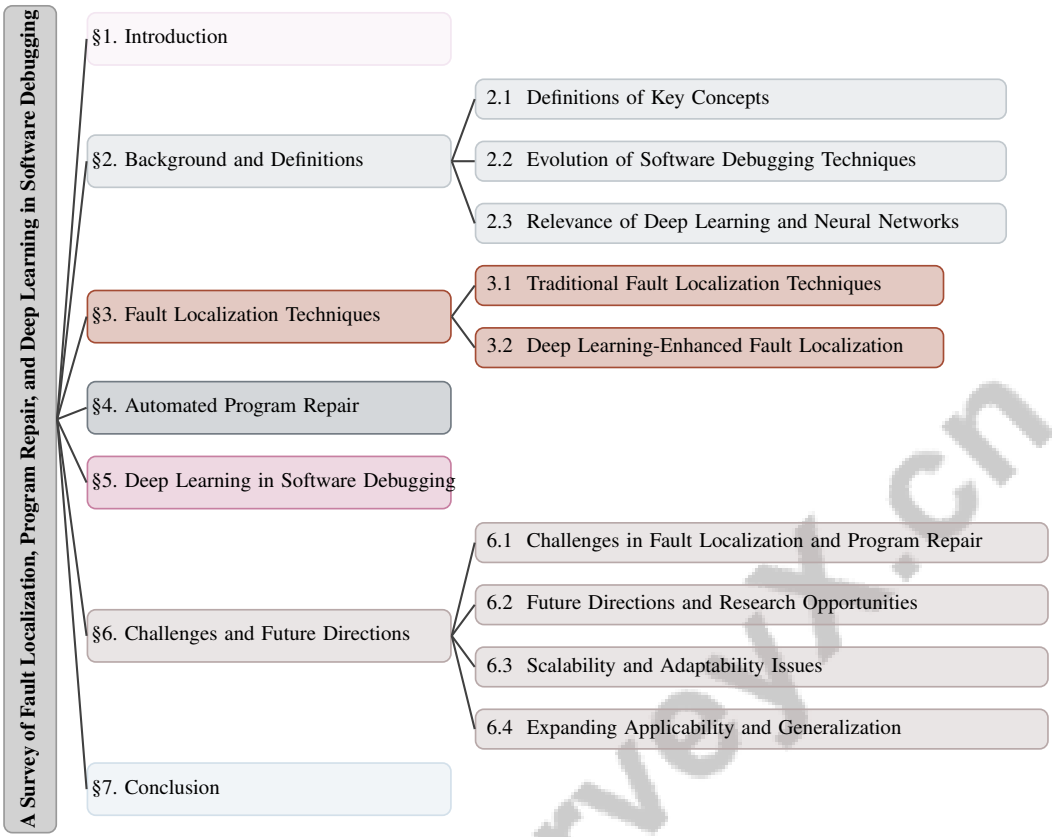

Figure 1: chapter structure

APR is also vital in mitigating system failures and security vulnerabilities, frequently caused by software bugs [7]. By autonomously generating corrective patches, APR bolsters software robustness and contributes positively to the security landscape.

The significance of APR is further highlighted by its influence on software development benchmarks that evaluate repair systems on real-world Java bugs, thereby propelling advancements in automatic software repair [8]. As deep learning continues to progress, the role of APR in software engineering is anticipated to expand, adeptly addressing the complexities of software defects with enhanced precision and efficiency.

1.2 Limitations of Traditional Debugging Methods

Traditional debugging methods face considerable challenges in addressing the complexities of modern software systems. They often rely on static analysis techniques that necessitate perfect control flow graph matching and bijective relationships between variable sets, conditions rarely met in practice, leading to failures in correcting erroneous programs [9]. Additionally, these methods struggle with the out-of-vocabulary (OOV) problem in programming languages and underutilize code-related information, which limits their effectiveness across various programming environments [10].

The reliance on enumerative solutions to predict fixes for each potential bug location results in a significant loss of contextual information, thereby undermining the accuracy of traditional APR techniques [3]. Moreover, these methods are inefficient due to their dependence on historical data and bug localizers, which constrains their ability to produce high-quality fix suggestions [4]. The inefficiency is exacerbated by the computational costs and language-specific nature of these methods, limiting their applicability across different programming languages [2].

Existing APR tools often neglect runtime behavior or desired program states in the repair process, complicating the determination of intended fixes [11]. This limitation is particularly acute when addressing large-scale buggy programs, as traditional methods can be time-consuming and resource-

intensive [12]. Furthermore, generating sufficient high-quality training samples remains a challenge, exacerbated by the limitations of current datasets and the inherent complexity of software bugs [13].

Benchmarks used to evaluate traditional debugging methods often fail to capture the semantic complexity of real-world bugs, leading to ineffective assessments of repair techniques [8]. Additionally, deep learning models integrated into traditional debugging processes necessitate extensive training data and computational resources, and are prone to overfitting, further emphasizing the limitations of conventional methodologies [14]. The high resource demands of Large Language Models (LLMs) and their fixed prompts and feedback loops, which do not emulate human debugging processes, present significant obstacles to their practical application in software development. These challenges highlight the need for advanced techniques to surpass the constraints of traditional debugging methods and address the complexities of modern software development environments.

1.3 Structure of the Survey

This survey is systematically organized to provide an in-depth exploration of the interplay between fault localization, program repair, and deep learning within the context of software debugging. It begins with an introductory section outlining the significance of automated program repair, emphasizing its importance in modern software development and the limitations of traditional debugging methods. Section 2 offers background and definitions, clarifying key concepts such as fault localization, program repair, deep learning, automated program repair, and neural networks, establishing a foundational understanding for subsequent analysis.

Section 3 delves into fault localization techniques, comparing traditional methods with those enhanced by deep learning. Section 4 focuses on automated program repair, discussing both established and emerging methodologies, particularly the advancements facilitated by neural networks in patch generation and evaluation. The following section, Section 5, explores the application of deep learning in software debugging, highlighting the integration of AI techniques and presenting case studies to illustrate practical implementations, including the role of natural language processing in enhancing debugging processes.

The survey addresses challenges and future directions in Section 6, pinpointing current obstacles in the integration of deep learning with fault localization and program repair, and proposing potential research avenues. This section also discusses scalability, adaptability, and the expansion of applicability and generalization of deep learning techniques in software debugging. The survey concludes with a synthesis of key findings, reflecting on the impact of deep learning on fault localization and program repair, and reiterating the potential benefits and challenges associated with these advanced computational techniques.The following sections are organized as shown in Figure 1.

2 Background and Definitions

2.1 Definitions of Key Concepts

Fault Localization (FL) is pivotal in software debugging, pinpointing code segments responsible for errors, thereby improving software quality [15]. Traditional techniques like Spectrum-Based Fault Localization (SBFL) utilize execution sequences to identify faults but often lack accuracy compared to semantic code search methods [16]. Advanced approaches, particularly those leveraging deep learning, aim to address these shortcomings. DeepFault, for instance, employs whitebox testing to examine neuron behavior in neural networks, enhancing fault detection by generating new inputs [17]. Similarly, tools like DeepDiagnosis enhance debugging by localizing faults, reporting errors, and suggesting fixes in neural networks [18].

Automated Program Repair (APR) involves methodologies for autonomously identifying and fixing software defects through patch generation. Approaches include search-based, constraint-based, and learning-based methods, with the latter utilizing extensive bug/fix datasets for training [19]. The integration of large language models (LLMs) into APR has shown potential in generating contextually relevant patches, overcoming the limitations of traditional feedback-dependent methods [7]. Techniques like TraceFixer leverage execution traces and desired program states to enhance bug fix precision [11].

Deep learning has reframed program repair as a neural machine translation task, converting buggy code into its corrected form, thus improving APR methods' accuracy and generalization across various bug types [6]. Graph neural networks facilitate program equivalence and analysis, crucial for effective repair [9]. Datasets like Defects4J, containing real-world Java bugs and patches, support controlled studies in software testing, advancing FL and APR technique development and evaluation [8].

2.2 Evolution of Software Debugging Techniques

Software debugging has evolved from traditional methods to sophisticated AI-driven approaches, significantly impacting modern software engineering. Initially, debugging relied on static analysis and SBFL, which used test coverage data correlations to identify faults. However, these methods often lacked precision due to their inability to capture complex software interactions essential for accurate localization and repair. Recent methods using natural language processing have improved file identification by analyzing defect reports and source code, reducing files inspected per defect by over 91%. Integrating multiple software artifacts has enhanced localization accuracy and patch plausibility [20, 21, 22, 23].

Deep learning has revolutionized debugging, enhancing precision and scalability [10]. Neural Program Repair (NPR) systems utilize deep learning to improve fault localization and repair efficacy [10]. LLMs have further advanced this evolution, showing significant potential in code generation and fault localization [24].

Alternative data sources, such as stack traces, enhance debugging techniques beyond traditional test execution data [25]. Specialized benchmarks for dynamically typed languages, like Python, address unique challenges in effective fault localization across diverse environments [26]. These benchmarks are crucial due to diverse tasks and metrics, complicating study comparisons and hindering field progress [27].

Advanced techniques like DEAR address multi-hunk, multi-statement bugs, marking a shift from traditional single-statement fixes [6]. Despite advancements, challenges in scaling these techniques for large programs persist, leading to long execution times and fault validation difficulties in extensive codebases [12].

The continuous evolution of software debugging techniques is essential for meeting modern software engineering demands. Categorizing research into APR and code generation using LLMs establishes frameworks for understanding AI integration [5]. These advancements ensure robust, efficient, and adaptable debugging processes for contemporary software systems.

2.3 Relevance of Deep Learning and Neural Networks

Deep learning and neural networks have become integral to software debugging advancements, significantly enhancing both fault localization and APR. These AI-driven methodologies offer novel approaches for precise and efficient bug identification and fixing. The RAP-Gen framework, for instance, combines bug-fix pattern retrieval with a code-aware language model to generate patches, improving the APR process [28]. This hybrid approach highlights neural networks' ability to integrate diverse data sources for effective debugging solutions.

In fault localization, methods like DEEPRL4FL treat the task as an image recognition problem, using convolutional neural networks (CNNs) to detect patterns in code coverage data [29]. This application demonstrates deep learning's potential to transform traditional debugging tasks through pattern recognition. Similarly, CosFL enhances fault localization by integrating semantic code search principles, using natural language queries to describe buggy functionalities [16]. These advancements underscore deep learning techniques' adaptability in addressing modern software systems' complexities.

Neural networks have also revolutionized APR techniques. Systems like DeepFix use multi-layered sequence-to-sequence neural networks to predict and rectify erroneous code segments, capturing long-term dependencies [30]. The RING engine exemplifies this by automating repair across multiple programming languages using a large language model [31]. Large-scale datasets of buggy and fixed code, as seen in NPR systems, significantly advance APR by providing robust training data enhancing model performance [1].

Deep learning extends beyond traditional debugging methods, as demonstrated by RepairAgent, an autonomous LLM-based agent that autonomously decides which tools to invoke for information gathering, repair ingredient collection, and fix validation [7]. This agent exemplifies neural networks' potential to automate complex decision-making processes in software debugging.

Deep learning and neural networks have ushered in a new era of software debugging, characterized by enhanced accuracy, efficiency, and scalability. As these technologies evolve, they promise increasingly sophisticated solutions to contemporary software development challenges, solidifying their relevance in the field [32].

3 Fault Localization Techniques

Category	Feature	Method
Traditional Fault Localization Techniques	Data-Driven Enhancements	DD[33], HAPR-TB[34], TS-FL[20], SBEST[25]
	Probabilistic Approaches	L2S[35], FL-PSM[36]
	Machine Learning Integration	TGL[37]
Deep Learning-Enhanced Fault Localization	Program Analysis	DCAF[24], RPAFT[38], VARDT[39], FLF[15]
	Model Behavior Insights	FC[40], AMA[41]
	Comprehensive Debugging	DEAR[6]

Table 1: This table provides a comprehensive summary of fault localization techniques, categorizing them into traditional and deep learning-enhanced methods. It highlights various features and methods employed in each category, illustrating the evolution from data-driven and probabilistic approaches to advanced program analysis and model behavior insights. The table underscores the integration of machine learning and deep learning in modern fault localization efforts, reflecting the shift towards more sophisticated debugging solutions.

Exploring fault localization techniques is crucial for understanding the evolution of software debugging. Traditional methods, like Spectrum-Based Fault Localization (SBFL), rely on failing tests to identify bugs, but this condition is rarely met in practice, with only 3.33% of bugs linked to such tests. Innovative approaches now leverage stack traces from crash reports, as seen in the SBEST approach, which integrates stack trace data with test coverage to improve fault localization metrics [42, 25, 43, 44]. However, challenges remain in complex scenarios due to reliance on test cases. Table 3 presents a detailed comparison of fault localization techniques, showcasing the transition from traditional methods to deep learning-enhanced strategies.

The emergence of deep learning-enhanced fault localization techniques marks a paradigm shift in debugging. These methods use sophisticated algorithms to detect intricate patterns in code, significantly improving Automated Program Repair (APR) efficiency and bug detection. The integration of Large Language Models (LLMs) facilitates context-aware fixes, enhancing automated debugging [5, 45, 46, 47, 48]. This section explores traditional fault localization techniques to provide context for advancements in deep learning methodologies.

3.1 Traditional Fault Localization Techniques

Method Name	Method Limitations	Enhancement Strategies	Applicability Challenges
SBEST[25]	Failing Tests	Probabilistic Models	Complex Scenarios
TS-FL[20]	Quality Reliance	Probabilistic Models	Complex Scenarios
TGL[37]	Failing Tests	Deep Learning	Complex Scenarios
DD[33]	Failing Tests	Probabilistic Models	Semantic Bugs
HAPR-TB[34]	Infinite Traces	Dynamic Analysis	Semantic Bugs
L2S[35]	Failing Tests	Probabilistic Models	Complex Scenarios
FLF[15]	Unrealistic Assumptions	Machine Learning-based	Complex Scenarios
FL-PSM[36]	Failing Tests	Probabilistic Models	Semantic Bugs

Table 2: Comparison of traditional fault localization methods, highlighting their limitations, enhancement strategies, and applicability challenges. The table provides insights into the constraints faced by each method and the strategies proposed to overcome these limitations, emphasizing the need for advanced techniques to address complex scenarios and semantic bugs.

Traditional fault localization (FL) techniques, such as SBFL, have been foundational in software debugging, correlating test coverage data with program failures. These methods aim to reduce debugging costs by identifying faulty code segments but are often limited by their reliance on failing

tests, which are not always available in real-world scenarios [25]. The necessity for test cases or detailed execution traces poses additional challenges, particularly in user-submitted reports lacking such data [20]. Table 2 presents an analytical overview of various traditional fault localization techniques, outlining their inherent limitations, potential enhancement strategies, and the challenges they face in practical applicability.

Traditional FL methods typically analyze faults at the line level, which restricts their ability to address complex faults involving multiple lines or omitted statements [37]. They also struggle with non-numerical errors and providing actionable fixes, limiting their real-world applicability [33]. Furthermore, these methods have difficulty localizing termination and liveness bugs, which generate infinite traces, complicating debugging in dynamic environments [34].

Traditional methods have shown limited success with backdoor defects, achieving an average localization effectiveness of only 17.64% WJI, highlighting their inadequacy in complex scenarios [49]. The use of predefined templates and constraints further limits flexibility, especially in multi-hunk and multi-fault repairs [35]. Additionally, these methods often fail to address semantic bugs, which, while common, are overshadowed by the time-intensive nature of debugging memory and concurrency issues [15].

Enhancements to traditional FL methods include integrating probabilistic models to manage uncertainty and variability in software faults [36]. However, the core issue remains the limited precision and adaptability of these methods in complex systems, necessitating exploration into advanced techniques like deep learning [50]. The lack of explainability in AI models used for software engineering further complicates their deployment and acceptance in critical applications [51]. These limitations highlight the ongoing need for innovation in fault localization methodologies.

3.2 Deep Learning-Enhanced Fault Localization

Deep learning techniques have significantly advanced fault localization by enhancing the precision and efficiency of identifying faulty code segments. These methods leverage deep learning models' ability to capture complex patterns and program semantics that traditional methods often miss. For instance, DEEPRL4FL uses Convolutional Neural Networks (CNNs) to analyze code coverage matrices and data dependencies, effectively identifying faulty code statements and methods [29]. This showcases deep learning's transformative potential in debugging through pattern recognition.

Integrating program dependence analysis with filtering mechanisms, as seen in REPEATNPR, enhances neural program repair tasks by combining deep learning with program analysis for improved fault localization [38]. Similarly, VARDT employs a decision tree model with program dependency information to identify fault-relevant variables, using decision trees' interpretability to pinpoint root causes of test failures [39].

Attention mechanisms have also enhanced fault localization processes. By analyzing attention weights, developers gain insights into model behavior, leading to more effective fault localization [41]. This underscores the importance of explainability in AI models, especially in critical applications like software debugging.

Fix-Con introduces a novel approach by comparing source and target models to identify and repair discrepancies, highlighting deep learning's role in automating fault localization tasks [40]. This technique demonstrates deep learning's adaptability in addressing modern software complexities.

Moreover, incorporating code reduction techniques, as shown by DeepCode AI Fix, enhances LLM performance in fault localization by focusing on relevant code snippets [24]. This illustrates the potential for optimizing LLMs for specific debugging tasks, improving their applicability in real-world scenarios.

Despite these advancements, challenges persist, particularly in evaluating patch correctness. Current methods often rely on manual author checks and automated tests, which may introduce biases and lead to potential overfitting [52]. Addressing these challenges is crucial for enhancing the reliability of deep learning-enhanced fault localization techniques.

The integration of deep learning into fault localization represents a new paradigm in software debugging, characterized by improved precision, efficiency, and adaptability. As these techniques evolve, they promise increasingly sophisticated solutions to modern software challenges. Notably,

advanced techniques like DEAR utilize deep learning to accurately identify and fix multiple buggy statements in a single patch, showcasing comprehensive solutions in complex debugging scenarios [6]. Leveraging large, curated datasets like RunBugRun, which provides executable buggy/fixed code pairs, is critical for training models in deep learning-enhanced fault localization [1]. Developing frameworks that combine data from multiple sources, including bug classification schemas and historical data, further enriches the debugging process [15].

Feature	Traditional Fault Localization Techniques	Deep Learning-Enhanced Fault Localization
Fault Identification Method	Test Coverage Analysis	Pattern Recognition
Complexity Handling	Limited Multi-line Faults	Handles Complex Patterns
Enhancement Techniques	Probabilistic Models	Attention Mechanisms

Table 3: This table provides a comparative analysis of traditional fault localization techniques and deep learning-enhanced methods. It highlights key differences in fault identification methods, complexity handling, and enhancement techniques, illustrating the transition towards more sophisticated debugging approaches. Such comparisons are essential for understanding the evolution and effectiveness of fault localization strategies in software engineering.

4 Automated Program Repair

4.1 Traditional and Emerging Methodologies

Automated Program Repair (APR) has advanced from traditional methods to AI-driven approaches. Traditional techniques include search-based, constraint-based, and template-based methods. Search-based methods, like genetic algorithms, explore vast fault spaces but face challenges due to complexity and resource demands. Fault localization techniques often excel at identifying related files but struggle with specific defects, necessitating a comprehensive understanding of bug and patch properties in datasets such as Defects4J to improve outcomes [20, 21, 53]. Constraint-based methods rely on formal specifications, limiting their applicability due to the scarcity of such specifications. Template-based methods use predefined templates, which lack generalizability and require manual adaptation.

Emerging methodologies integrate learning-based approaches, incorporating deep learning and neural networks. DEAR (Deep learning-based Automated Repair) exemplifies this by combining spectrum-based techniques with data-flow analysis to address complex bugs, employing a two-tier, tree-based LSTM model for code transformations [54, 6, 18, 37]. Learning-based approaches like Ratchet generate patches from historical fixes, contrasting with traditional techniques. The integration of Large Language Models (LLMs) for bug fixing shows promise, though optimization is needed to address inefficiencies.

Hybrid models such as SimFix enhance repair by mining search spaces from patches and similar code, while T5APR uses a unified model based on CodeT5 and a checkpoint ensemble strategy to improve patch recommendations. Graph Neural Networks (GNNs) have shown success in variable mapping, outperforming existing tools and achieving high repair rates on evaluation datasets.

A significant APR challenge is validating patches, requiring execution against test suites to ensure correctness. This is compounded by the volume of patches produced, many of which may be incorrect or overly tailored—a phenomenon known as patch overfitting. Methods like Regression Test Selection (RTS) optimize testing by executing only impacted tests, enhancing APR efficiency [55, 56, 57]. Classifying machine-generated patches into categories like Same Location Same Modification (SLSM) and Different Location Different Modification (DLDM) aids in understanding diversity. Innovative approaches like CURE combine language models with code-aware search strategies and subword tokenization to create effective search spaces for patches.

The RepairAgent exemplifies AI integration, autonomously interacting with codebases to identify and fix bugs, highlighting the potential for AI-driven methodologies to enhance APR effectiveness [7]. Reliable benchmarks, such as those by [8], are crucial for improving reproducibility and transparency in research.

The APR landscape is rapidly evolving, emphasizing the integration of AI techniques, particularly large language models, to address traditional method limitations. This shift is driven by the need to manage software defects effectively and enhance debugging efficiency, with deep learning ad-

AI-generated, for reference only.

vancements leading to innovative APR techniques. These developments present opportunities and challenges, necessitating further exploration of LLM applications and deployment strategies within the APR domain [58, 46]. By leveraging deep learning, neural networks, and hybrid models, modern APR systems increasingly tackle contemporary software defects with improved accuracy and efficiency, setting new benchmarks for future research and applications in automated repair.

4.2 Enhancements through Neural Networks

The integration of deep learning and neural networks into Automated Program Repair (APR) has significantly improved bug-fixing precision and adaptability. The DEAR framework exemplifies this by incorporating deep learning and data-flow analysis, enhancing fault localization and correction in complex systems [6].

T5APR uses a checkpoint ensemble strategy to merge outputs from various training stages, improving patch generation accuracy [2]. This demonstrates neural networks' potential to refine repair processes through diverse training outputs.

The CosFL technique illustrates deep learning's impact on fault localization by successfully identifying many bugs in the Top-1 position, outperforming existing methods [16]. This improves APR processes by enhancing fault localization performance.

Recent research introduces novel fault localization methods utilizing historical data for more accurate bug detection and repair [15]. This highlights historical data's role in augmenting neural networks' APR capabilities.

The RepairAgent showcases neural networks' dynamic capabilities in APR by autonomously managing tool invocations and the repair process, demonstrating AI-driven systems' adaptability [7]. This emphasizes neural networks' potential to automate complex decision-making in software debugging.

Advancements in deep learning and neural networks have ushered in a transformative era in APR, characterized by increased accuracy, efficiency, and adaptability. As these technologies evolve, they promise sophisticated solutions to modern software system challenges, solidifying their role in automated repair. Future research should prioritize expanding diverse datasets, investigating few-shot learning methodologies, and applying code transformation techniques. These efforts aim to enhance APR, particularly in addressing security vulnerabilities, by improving code representations, minimizing reliance on extensive training data, and leveraging advanced models like LLMs for vulnerability detection and repair [59, 60, 24, 61, 62].

4.3 Improving Patch Generation and Evaluation

Benchmark	Size	Domain	Task Format	Metric
Defects4J[8]	357	Java Software Repair	Automatic Bug Repair	Test-suite adequacy, Correctness
SRepair[63]	522	Software Engineering	Function-level Program Repair	Plausible Fixes
BigIssue[64]	10,905	Software Engineering	Bug Localization	F1-score, Recall
kPAR[21]	22,954	Automated Program Repair	Bug Localization	Ochiai
LLM-FL[65]	195	Software Testing	Fault Localisation	TOP-1, TOP-5
APR-RTS[56]	2,532,915	Software Engineering	Patch Validation	Number of Test Executions
DLAPR[66]	58	Deep Learning	Program Repair	Fault Detection, Fault Localization
Globug[67]	61,431	Fault Localization	Bug Localization	MRR, MAP

Table 4: This table presents a comprehensive overview of various benchmarks utilized in the domain of automated program repair (APR) and software engineering. It details the size, domain, task format, and evaluation metrics of each benchmark, providing insight into their applicability and scope in improving patch generation and evaluation techniques.

Enhancing patch generation and evaluation is crucial for advancing APR systems' effectiveness. Recent methodologies have increasingly leveraged comprehensive datasets and sophisticated machine learning models to tackle patch accuracy challenges. For instance, Vasic et al.'s joint model achieves a maximum localization accuracy of 71

Program reduction techniques, as highlighted by Albataineh et al., focus on slicing programs to retain only necessary parts for fault localization and patch generation, reducing computational overhead and improving repair precision [12]. Vidziunas et al. further emphasize that program reduction can significantly enhance APR tool performance by reducing repair time and test executions without compromising repair quality [68].

Advanced APR systems like T5APR have demonstrated notable effectiveness in generating correct patches, successfully fixing 1,985 bugs across various benchmarks and proving competitive with state-of-the-art approaches [2]. Similarly, Ratchet has shown remarkable efficacy in producing syntactically valid bug fixes, with an empirical study reporting a 98.7

Natural language processing (NLP) techniques have also played a vital role in improving patch evaluation. For example, Quatrain utilizes NLP to correlate bug descriptions with generated patch descriptions, aiding in patch correctness classification [69]. This highlights NLP's significance in bridging the gap between human understanding and machine-generated patches.

Review4Repair integrates code review comments into the repair process, achieving a top-1 accuracy increase of 20.33

Template-based approaches like GAMMA have successfully repaired 82 bugs in the Defects4J-v1.2 dataset, outperforming existing state-of-the-art methods [70]. This success illustrates refining traditional template-based methodologies with modern enhancements.

Despite these advancements, challenges persist, particularly regarding overfitting on test suites, as noted by Martinez et al. [8]. Addressing this challenge is essential for improving automated patches' reliability and generalization.

Progress in patch generation and evaluation techniques within APR is critical for developing robust and scalable repair systems. Table 4 provides a detailed overview of the benchmarks employed in recent studies to enhance patch generation and evaluation within automated program repair systems. Recent advancements in deep learning and large language models enable more effective bug-fixing strategies by leveraging learned patterns from extensive code repositories and addressing complex software defects, including semantic errors and security vulnerabilities [58, 46, 60]. By utilizing comprehensive datasets, integrating machine learning models, and refining defect classifications, researchers can enhance automated software repair's accuracy and reliability, ultimately contributing to more efficient debugging processes.

5 Deep Learning in Software Debugging

5.1 Integration of AI Techniques in Software Debugging

The adoption of AI techniques, especially deep learning, has revolutionized software debugging by enhancing bug detection and resolution. Systems like RING simulate developer debugging while automating tasks through AI, capturing complex patterns often missed by traditional methods [31]. Conversational AI models, such as ChatGPT, enhance APR processes by improving interaction and understanding, thereby refining repair outcomes [71]. Toggle further optimizes LLMs for bug fixing through varied prompting strategies [37].

Explainability techniques, including attention mechanism analysis in transformer models, offer insights into AI behavior, essential for trust in AI-driven tools [41]. Systems like DeepCode AI Fix showcase AI's adaptability in managing complex software systems using LLMs [24]. Specialized measures like SMSFL improve fault localization, enhancing debugging efficiency [72]. Benchmarks such as BugSwarm provide real-world software failure datasets, fostering robust AI-driven debugging solutions [73].

AI techniques have also advanced program repair through systems like ThinkRepair, which utilizes structured examples and iterative feedback to enhance LLM reasoning for bug fixing [74]. AlphaRepair, based on CodeBERT, exemplifies deep learning integration in software debugging, optimizing repair methodologies [61]. MUFIN enhances AI-driven debugging by generating diverse, high-quality training samples, improving generalization to unseen bugs [13]. FL4Deep innovates by using a Knowledge Graph to represent relationships across deep learning system components [18].

AI integration in software debugging heralds a new era of precision and efficiency, offering sophisticated solutions to modern software challenges. Innovations like automated scientific debugging, which uses LLMs for generating hypotheses and explanations, and improved fault localization techniques that combine natural language bug reports with test executions, promise more precise defect identification. Leveraging various software artifacts for APR is yielding promising results, contributing to more reliable and maintainable software solutions [20, 23, 75, 76].

5.2 Case Studies and Applications

Deep learning in software debugging is exemplified through various case studies, highlighting AI-driven methodologies' efficacy in enhancing APR processes. ThinkRepair, evaluated using datasets like Defects4J and QuixBugs, demonstrates improved LLM reasoning for bug fixing through structured examples and iterative feedback [74]. These datasets, comprising real-world software bugs, facilitate APR technique evaluation across diverse scenarios, enhancing understanding of APR's capabilities and limitations [34, 77, 60]. Defects4J provides a robust collection of Java bugs for controlled experiments, while QuixBugs presents small, algorithmic bugs for assessing repair strategies across various programming languages.

The success of ThinkRepair and similar systems in analyzed case studies underscores the transformative potential of integrating advanced deep learning techniques into software debugging, particularly in improving bug detection and resolution accuracy and efficiency. By leveraging self-directed learning and sophisticated prompt engineering, these systems outperform traditional methods, demonstrating substantial improvements in fixing complex bugs and providing actionable insights for developers [51, 78, 50, 33, 74]. Utilizing AI capabilities, these systems generate more accurate and contextually relevant patches, enhancing software applications' reliability and maintainability. Insights from such case studies will guide the development of more sophisticated and effective debugging tools as the field continues to evolve.

5.3 Integration of Natural Language Processing

The integration of NLP into software debugging has led to significant advancements, particularly in enhancing code interpretability and contextual understanding. NLP techniques translate natural language bug descriptions into actionable debugging insights, bridging the gap between human language and machine-level code understanding. Systems like Quatrain utilize NLP to correlate bug descriptions with generated patch descriptions, facilitating intuitive patch evaluation [69]. By analyzing semantic similarities between bug reports and potential fixes, Quatrain improves patch validation accuracy, reducing incorrect or irrelevant patches.

NLP enhances bug identification and fixing efficiency while generating meaningful code comments and documentation, vital for preserving code readability and fostering collaboration among developers. Insights from code reviews, coupled with advanced models like CodeT5, streamline the debugging process, ensuring well-documented and understood code changes, ultimately improving software quality and team communication [79, 80, 81, 4]. Techniques leveraging NLP for automatic documentation generation significantly reduce the manual effort required for maintaining comprehensive code annotations, enhancing software projects' overall quality and maintainability.

NLP's application in software debugging extends beyond patch evaluation and documentation. It enhances APR systems by providing contextual information that aids in bug identification and resolution. Methods like Review4Repair integrate human insights from code review comments into the repair process, improving fix suggestions' quality [4]. This underscores NLP's potential to augment traditional debugging methodologies with valuable contextual knowledge, leading to more effective and efficient repair processes.

As NLP technologies advance, their integration into software debugging is expected to yield even more sophisticated solutions, enhancing debugging tools' precision and adaptability. By harnessing advanced NLP capabilities of AI-driven systems, developers gain profound insights into code semantics, refining debugging strategies. Recent studies indicate that LLMs facilitate the identification of semantic errors, security vulnerabilities, and runtime failures, leading to more context-aware and accurate debugging solutions. Innovative techniques like Automated Scientific Debugging align automated reasoning with human thought processes, providing intelligible explanations for generated patches, further improving developer decision efficiency and accuracy. Consequently, LLMs stream-

line debugging processes while fostering deeper code understanding, ultimately boosting software development effectiveness [80, 5, 24, 76].

In the realm of software engineering, particularly in the context of fault localization and automated program repair (APR), it is essential to address the myriad challenges that researchers face. Figure 2 illustrates these challenges and future directions, effectively highlighting key issues such as benchmark limitations, patch quality, integration challenges, and resource constraints. Furthermore, the figure outlines promising research opportunities that focus on model enhancements, human-in-the-loop approaches, scalability, adaptability, and the importance of expanding applicability and generalization through the utilization of diverse data. This visual representation not only complements the discussion but also serves as a critical reference point for understanding the complexities and potential advancements in this field.

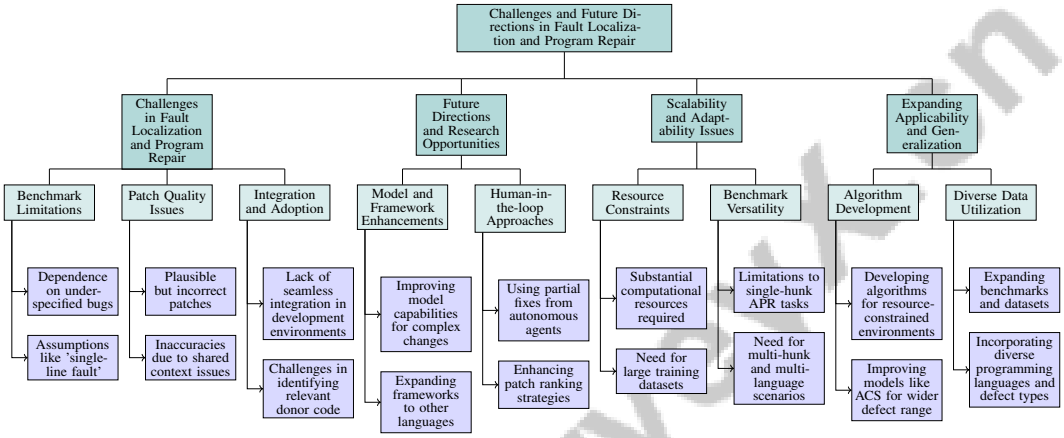

Figure 2: This figure illustrates the challenges and future directions in fault localization and automated program repair (APR), highlighting key issues such as benchmark limitations, patch quality, integration challenges, and resource constraints. It also outlines research opportunities focusing on model enhancements, human-in-the-loop approaches, scalability, adaptability, and expanding applicability and generalization through diverse data utilization.

6 Challenges and Future Directions

6.1 Challenges in Fault Localization and Program Repair

Deep learning's integration into fault localization and automated program repair (APR) faces several challenges. A major issue is the dependence on benchmarks with under-specified bugs, resulting in patches that fail to address core problems [8]. Additionally, assumptions like the 'single-line fault' limit the applicability of techniques in real-world scenarios with complex fault patterns [15]. Systems such as T5APR often produce plausible but incorrect patches, necessitating developer validation [2]. Challenges also arise from difficulties in capturing shared context among dependent predictions, leading to inaccuracies [3]. The quality of generated queries and search strategies, as seen in CosFL, further complicate repairs in complex codebases [16]. Existing methods often fail to emulate human debugging processes, which involve iterative understanding and testing, limiting their practical use [7]. Moreover, AI-driven approaches lack seamless integration in development environments, impeding widespread adoption [15]. Identifying relevant donor code within the local context is another challenge, often leading to incorrect patches [70]. Overcoming these challenges requires innovative methodologies to enhance the precision and efficiency of AI-driven debugging solutions.

6.2 Future Directions and Research Opportunities

Enhancing deep learning's role in fault localization and APR offers numerous research opportunities. Improving model capabilities to handle complex changes and context-aware features can significantly boost patch quality [14]. Expanding frameworks like DEAR to other languages and improving

performance on non-failing test bugs is crucial [6]. Future research should focus on expanding training datasets and refining extraction processes to enhance systems like FL4Deep [18]. Enhancing patch ranking strategies and integrating APR models like T5APR into continuous integration pipelines could streamline software development [2]. Extending frameworks to include more languages and debugging tools will improve adaptability [15]. Human-in-the-loop approaches using partial fixes from autonomous agents like RepairAgent could enhance performance on simpler bugs [7]. Addressing challenges posed by under-specified bugs remains critical for improving the reliability of APR systems [8].

6.3 Scalability and Adaptability Issues

Scalability and adaptability present significant hurdles in applying deep learning to APR and fault localization. The substantial computational resources required can be prohibitive, particularly in resource-limited environments [82]. This dependency limits deep learning's applicability in large-scale systems [83]. Additionally, the need for large training datasets is essential for effective deployment [62]. The adaptability of benchmarks is also a concern, as current limitations to single-hunk APR tasks in Java restrict their applicability to other languages and multi-hunk scenarios [84]. This underscores the need for more versatile benchmarks. The interpretability of deep learning models remains a concern, as they often function as black boxes, complicating understanding of their decision-making processes [83]. Addressing these issues requires developing efficient algorithms that function with limited resources and expanding benchmarks to include multi-hunk and multi-language scenarios. This will improve the integration of deep learning techniques into software debugging, promising more scalable and adaptable solutions. Deep learning aids in extracting intricate features from high-dimensional data, enhancing defect prediction and fault localization. Using diverse artifacts and data augmentation can mitigate class imbalance issues, leading to more robust debugging processes [50, 78].

6.4 Expanding Applicability and Generalization

Expanding the applicability and generalization of deep learning in software debugging is crucial for enhancing APR systems. Developing algorithms that work in resource-constrained environments ensures that APR solutions are accessible with limited resources [82]. Improving models like ACS to address a wider range of defects is essential [22]. Expanding the variety and complexity of benchmarks and datasets used for training and evaluating models will enhance their generalization capabilities. This diversity allows models to learn from a broader range of data patterns, improving their applicability across domains such as software engineering, medical image processing, and anomaly detection [32, 51, 78, 85, 86]. Incorporating diverse programming languages, environments, and defect types ensures APR systems are robust and adaptable. This expansion will also contribute to developing comprehensive evaluation frameworks for accurate assessments of model performance. Advancing applicability and generalization in deep learning-based software debugging is vital for ensuring these technologies address modern software complexities. Recent advancements, particularly in Large Language Models (LLMs), have revolutionized APR and code generation, facilitating accurate identification and rectification of semantic errors, vulnerabilities, and runtime failures. Techniques like Automated Scientific Debugging (AutoSD) enhance debugging by aligning automated reasoning with human logic, providing intelligible explanations for patches, thus improving decision-making efficiency. A fault localization framework utilizing comprehensive bug tracking data can streamline bug identification, particularly in complex scenarios. These innovations signify a transformative shift in debugging practices, paving the way for robust software solutions [79, 5, 76, 33, 15].

7 Conclusion

This survey highlights the profound influence of deep learning on enhancing fault localization and automated program repair (APR), showcasing both significant improvements and persisting challenges. DeepLocalize exemplifies the advancements in fault detection and localization, demonstrating superior performance compared to traditional methods. Similarly, Recoder's notable enhancement in addressing single-hunk bugs underscores the potential of deep learning to refine software debugging accuracy. The integration of code language models (CLMs) has further bolstered APR by surpassing

existing techniques, particularly when tailored with domain-specific data. Innovative methodologies like D4C, which optimize output alignment and artifact utilization, have proven effective, marking a meaningful step forward in program repair techniques.

However, fully harnessing deep learning in software engineering remains challenging. Critical insights point to the necessity of refining deep neural network (DNN) variables, enhancing experimental controls, and conducting multiple training cycles to ensure robust outcomes. The availability of data continues to be a significant hurdle, impeding the advancement of deep learning applications in this domain. Despite the progress of neural models, they have yet to match the proficiency of human developers in bug fixing, highlighting the indispensable role of human expertise. Nonetheless, approaches such as RTT-APR indicate deep learning's potential to complement traditional methods by addressing overlooked bugs, suggesting a promising avenue for future developments in software debugging.

References

[1] Julian Aron Prenner and Romain Robbes. Runbugrun – an executable dataset for automated program repair, 2023.

[2] Reza Gharibi, Mohammad Hadi Sadreddini, and Seyed Mostafa Fakhrahmad. T5apr: Empowering automated program repair across languages through checkpoint ensemble, 2024.

[3] Marko Vasic, Aditya Kanade, Petros Maniatis, David Bieber, and Rishabh Singh. Neural program repair by jointly learning to localize and repair, 2019.

[4] Faria Huq, Masum Hasan, Mahim Anzum Haque Pantho, Sazan Mahbub, Anindya Iqbal, and Toufique Ahmed. Review4repair: Code review aided automatic program repairing, 2020.

[5] Avinash Anand, Akshit Gupta, Nishchay Yadav, and Shaurya Bajaj. A comprehensive survey of ai-driven advancements and techniques in automated program repair and code generation, 2024.

[6] Yi Li, Shaohua Wang, and Tien N. Nguyen. Dear: A novel deep learning-based approach for automated program repair, 2022.

[7] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. Repairagent: An autonomous, llm-based agent for program repair, 2024.

[8] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset. *Empirical Software Engineering*, 22:1936–1964, 2017.

[9] Pedro Orvalho, Jelle Piepenbrock, Mikoláš Janota, and Vasco Manquinho. Graph neural networks for mapping variables between programs – extended version, 2023.

[10] Wenkang Zhong, Chuanyi Li, Jidong Ge, and Bin Luo. Neural program repair: Systems, challenges and solutions, 2022.

[11] Islem Bouzenia, Yangruibo Ding, Kexin Pei, Baishakhi Ray, and Michael Pradel. Tracefixer: Execution trace-driven program repair, 2023.

[12] Omar I. Al-Bataineh. On the effectiveness of dynamic reduction techniques in automated program repair, 2024.

[13] André Silva, João F. Ferreira, He Ye, and Martin Monperrus. Mufin: Improving neural repair models with back-translation, 2023.

[14] Hideaki Hata, Emad Shihab, and Graham Neubig. Learning to generate corrective patches using neural machine translation, 2019.

[15] Thomas Hirsch. A fault localization and debugging support framework driven by bug tracking data, 2021.

[16] Yihao Qin, Shangwen Wang, Yan Lei, Zhuo Zhang, Bo Lin, Xin Peng, Liqian Chen, and Xiaoguang Mao. Fault localization from the semantic code search perspective, 2024.

[17] Hasan Ferit Eniser, Simos Gerasimou, and Alper Sen. Deepfault: Fault localization for deep neural networks, 2019.

[18] Mohammad Mehdi Morovati, Amin Nikanjam, and Foutse Khomh. Fault localization in deep learning-based software: A system-level approach, 2024.

[19] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. Automated repair of programs from large language models, 2023.

[20] Zachary P. Fry and Westley Weimer. Fault localization using textual similarities, 2012.

[21] Kui Liu, Anil Koyuncu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, and Yves Le Traon. You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems, 2019.

[22] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. Precise condition synthesis for program repair. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 416–426. IEEE, 2017.

[23] Qiong Feng, Xiaotian Ma, Jiayi Sheng, Ziyuan Feng, Wei Song, and Peng Liang. Integrating various software artifacts for better llm-based bug localization and program repair, 2024.

[24] Berkay Berabi, Alexey Gronskiy, Veselin Raychev, Gishor Sivanrupan, Victor Chibotaru, and Martin Vechev. Deepcode ai fix: Fixing security vulnerabilities with large language models, 2024.

[25] Lorena Barreto Simedo Pacheco, An Ran Chen, Jinqiu Yang, Tse-Hsun, and Chen. Leveraging stack traces for spectrum-based fault localization in the absence of failing tests, 2024.

[26] Mohammad Rezaalipour and Carlo A. Furia. An empirical study of fault localization in python programs, 2024.

[27] Joseph Renzullo, Pemma Reiter, Westley Weimer, and Stephanie Forrest. Automated program repair: Emerging trends pose and expose problems for benchmarks, 2024.

[28] Weishi Wang, Yue Wang, Shafiq Joty, and Steven C. H. Hoi. Rap-gen: Retrieval-augmented patch generation with codet5 for automatic program repair, 2023.

[29] Yi Li, Shaohua Wang, and Tien N. Nguyen. Fault localization with code coverage representation learning, 2021.

[30] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. Deepfix: Fixing common c language errors by deep learning. In *Proceedings of the aaai conference on artificial intelligence*, volume 31, 2017.

[31] Repair is nearly generation: Multilingual program repair with llms.

[32] Saptarshi Sengupta, Sanchita Basak, Pallabi Saikia, Sayak Paul, Vasilios Tsalavoutis, Frederick Atiah, Vadlamani Ravi, and Alan Peters. A review of deep learning with special emphasis on architectures, applications and recent trends, 2019.

[33] Mohammad Wardat, Breno Dantas Cruz, Wei Le, and Hridesh Rajan. Deepdiagnosis: Automatically diagnosing faults and recommending actionable fixes in deep learning programs, 2021.

[34] Omar I. Al-Bataineh and Leon Moonen. Towards extending the range of bugs that automated program repair can handle, 2022.

[35] Yingfei Xiong, Bo Wang, Guirong Fu, and Linfei Zang. Learning to synthesize, 2018.

[36] Hannes Thaller, Lukas Linsbauer, Alexander Egyed, and Stefan Fischer. Towards fault localization via probabilistic software modeling, 2020.

[37] Soneya Binta Hossain, Nan Jiang, Qiang Zhou, Xiaopeng Li, Wen-Hao Chiang, Yingjun Lyu, Hoan Nguyen, and Omer Tripp. A deep dive into large language models for automated bug localization and repair, 2024.

[38] Yuwei Zhang, Ge Li, Zhi Jin, and Ying Xing. Neural program repair with program dependence analysis and effective filter mechanism, 2023.

[39] Jiajun Jiang, Yumeng Wang, Junjie Chen, Delin Lv, and Mengjiao Liu. Variable-based fault localization via enhanced decision tree, 2022.

[40] Nikolaos Louloudakis, Perry Gibson, José Cano, and Ajitha Rajan. Fix-con: Automatic fault localization and repair of deep learning model conversions between frameworks, 2024.

[41] Ahmad Haji Mohammadkhani, Chakkrit Tantithamthavorn, and Hadi Hemmati. Explainable ai for pre-trained code models: What do they learn? when they do not work?, 2023.

[42] Davide Ginelli, Oliviero Riganelli, Daniela Micucci, and Leonardo Mariani. Exception-driven fault localization for automated program repair, 2022.

[43] Leping Li and Hui Liu. A hybrid approach to fine-grained automated fault localization, 2021.

[44] Higor A. de Souza, Marcos L. Chaim, and Fabio Kon. Spectrum-based software fault localization: A survey of techniques, advances, and challenges, 2017.

[45] Jiajun Jiang, Zijie Zhao, Zhirui Ye, Bo Wang, Hongyu Zhang, and Junjie Chen. Enhancing redundancy-based automated program repair by fine-grained pattern mining, 2023.

[46] Kai Huang, Zhengzi Xu, Su Yang, Hongyu Sun, Xuejun Li, Zheng Yan, and Yuqing Zhang. A survey on automated program repair techniques, 2023.

[47] Martin Monperrus. A critical review of "automatic patch generation learned from human-written patches": Essay on the problem statement and the evaluation of automatic software repair, 2014.

[48] Imen Azaiz, Oliver Deckarm, and Sven Strickroth. Ai-enhanced auto-correction of programming exercises: How effective is gpt-3.5?, 2023.

[49] Yisong Xiao, Aishan Liu, Xinwei Zhang, Tianyuan Zhang, Tianlin Li, Siyuan Liang, Xianglong Liu, Yang Liu, and Dacheng Tao. Bdefects4nn: A backdoor defect database for controlled localization studies in neural networks, 2024.

[50] Görkem Giray, Kwabena Ebo Bennin, Ömer Köksal, Önder Babur, and Bedir Tekinerdogan. On the use of deep learning in software defect prediction, 2022.

[51] Sicong Cao, Xiaobing Sun, Ratnadira Widyasari, David Lo, Xiaoxue Wu, Lili Bo, Jiale Zhang, Bin Li, Wei Liu, Di Wu, and Yixin Chen. A systematic literature review on explainability for machine/deep learning-based software engineering research, 2025.

[52] Shangwen Wang, Ming Wen, Liqian Chen, Xin Yi, and Xiaoguang Mao. How different is it between machine-generated and developer-provided patches? an empirical study on the correct patches generated by automated program repair techniques, 2019.

[53] Victor Sobreira, Thomas Durieux, Fernanda Madeiral, Martin Monperrus, and Marcelo A. Maia. Dissection of a bug dataset: Anatomy of 395 patches from defects4j, 2018.

[54] Shouliang Yang, Junming Cao, Hushuang Zeng, Beijun Shen, and Hao Zhong. Locating faulty methods with a mixed rnn and attention model, 2021.

[55] Manish Motwani. High-quality automated program repair, 2021.

[56] Yiling Lou, Jun Yang, Samuel Benton, Dan Hao, Lin Tan, Zhenpeng Chen, Lu Zhang, and Lingming Zhang. When automated program repair meets regression testing – an extensive study on 2 million patches, 2024.

[57] Davide Ginelli, Matias Martinez, Leonardo Mariani, and Martin Monperrus. A comprehensive study of code-removal patches in automated program repair, 2021.

[58] Quanjun Zhang, Chunrong Fang, Yang Xie, YuXiang Ma, Weisong Sun, Yun Yang, and Zhenyu Chen. A systematic literature review on large language models for automated program repair, 2024.

[59] Anastasiia Grishina. Enabling automatic repair of source code vulnerabilities using data-driven methods, 2022.

[60] Quanjun Zhang, Chunrong Fang, Yuxiang Ma, Weisong Sun, and Zhenyu Chen. A survey of learning-based automated program repair, 2023.

[61] Chunqiu Steven Xia and Lingming Zhang. Less training, more repairing please: Revisiting automated program repair via zero-shot learning, 2024.

[62] Xiang Gao, Yannic Noller, and Abhik Roychoudhury. Program repair, 2022.

[63] Jiahong Xiang, Xiaoyang Xu, Fanchu Kong, Mingyuan Wu, Zizheng Zhang, Haotian Zhang, and Yuqun Zhang. How far can we go with practical function-level program repair?, 2024.

[64] Paul Kassianik, Erik Nijkamp, Bo Pang, Yingbo Zhou, and Caiming Xiong. Bigissue: A realistic bug localization benchmark, 2023.

[65] Yonghao Wu, Zheng Li, Jie M. Zhang, Mike Papadakis, Mark Harman, and Yong Liu. Large language models in fault localisation, 2023.

[66] Jialun Cao, Meiziniu Li, Ming Wen, and Shing chi Cheung. A study on prompt design, advantages and limitations of chatgpt for deep learning program repair, 2023.

[67] Nima Miryeganeh, Sepehr Hashtroudi, and Hadi Hemmati. Globug: Using global data in fault localization, 2021.

[68] Linas Vidziunas, David Binkley, and Leon Moonen. The impact of program reduction on automated program repair, 2024.

[69] Haoye Tian, Xunzhu Tang, Andrew Habib, Shangwen Wang, Kui Liu, Xin Xia, Jacques Klein, and Tegawendé F. Bissyandé. Is this change the answer to that problem? correlating descriptions of bug and code changes for evaluating patch correctness, 2022.

[70] Quanjun Zhang, Chunrong Fang, Tongke Zhang, Bowen Yu, Weisong Sun, and Zhenyu Chen. Gamma: Revisiting template-based automated program repair via mask prediction, 2023.

[71] Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. An analysis of the automatic bug fixing performance of chatgpt, 2023.

[72] Vangipuram Radhakrishna. Design and analysis of novel kernel measure for software fault localization, 2016.

[73] David A. Tomassi and Cindy Rubio-González. A note about: Critical review of bugswarm for fault localization and program repair, 2019.

[74] Xin Yin, Chao Ni, Shaohua Wang, Zhenhao Li, Limin Zeng, and Xiaohu Yang. Thinkrepair: Self-directed automated program repair, 2024.

[75] Manish Motwani and Yuriy Brun. Better automatic program repair by using bug reports and tests together, 2023.

[76] Sungmin Kang, Bei Chen, Shin Yoo, and Jian-Guang Lou. Explainable automated debugging via large language model-driven scientific debugging, 2023.

[77] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. Automated program repair in the era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1482–1494. IEEE, 2023.

[78] Xiangping Chen, Xing Hu, Yuan Huang, He Jiang, Weixing Ji, Yanjie Jiang, Yanyan Jiang, Bo Liu, Hui Liu, Xiaochen Li, Xiaoli Lian, Guozhu Meng, Xin Peng, Hailong Sun, Lin Shi, Bo Wang, Chong Wang, Jiayi Wang, Tiantian Wang, Jifeng Xuan, Xin Xia, Yibiao Yang, Yixin Yang, Li Zhang, Yuming Zhou, and Lu Zhang. Deep learning-based software engineering: Progress, challenges, and opportunities, 2024.

[79] Thomas Hirsch and Birgit Hofer. What we can learn from how programmers debug their code, 2021.

[80] Nghi D. Q. Bui, Yue Wang, and Steven Hoi. Detect-localize-repair: A unified framework for learning to debug with codet5, 2022.

[81] Zelin Zhao, Zhaogui Xu, Jialong Zhu, Peng Di, Yuan Yao, and Xiaoxing Ma. The right prompts for the job: Repair code-review defects with large language model, 2023.

[82] Jisung Kim and Byeongjung Lee. Mcrepair: Multi-chunk program repair via patch optimization with buggy block, 2023.

[83] Safeeullah Soomro, Mohammad Riyaz Belgaum, Zainab Alansari, and Mahdi H. Miraz. Fault localization models in debugging, 2018.

[84] Guochang Li, Chen Zhi, Jialiang Chen, Junxiao Han, and Shuiguang Deng. Exploring parameter-efficient fine-tuning of large language model on automated program repair, 2024.

[85] Quentin Fournier, Daniel Aloise, Seyed Vahid Azhari, and François Tetreault. On improving deep learning trace analysis with system call arguments, 2021.

[86] Sira Vegas and Sebastian Elbaum. Pitfalls in experiments with dnn4se: An analysis of the state of the practice, 2023.

Disclaimer:

SurveyX is an AI-powered system designed to automate the generation of surveys. While it aims to produce high-quality, coherent, and comprehensive surveys with accurate citations, the final output is derived from the AI's synthesis of pre-processed materials, which may contain limitations or inaccuracies. As such, the generated content should not be used for academic publication or formal submissions and must be independently reviewed and verified. The developers of SurveyX do not assume responsibility for any errors or consequences arising from the use of the generated surveys.