Dean&Francis
ISSN 2959-6157

# Analysis of Research Status in the Field of Automated Program Repair

**Jishang Han[1], Dereck Huang[2, *]**

[1]Liaocheng No.1 Middle School of Shandong Province, Liaocheng, China
[2]Zhixin Middle School (Ersha Island Campus) of Guangzhou City, Guangzhou, China
*Corresponding author: Dereckhuang2024@outlook.com

**Abstract:**
As software systems keep developing and becoming more widely used, it's impossible to avoid the program bugs that come with them. To reduce the pressure on developers when modifying programs and make bug fixes more efficient, Automated Program Repair (APR) plays a key role. This paper summarizes the main research progress in the field of APR, analyzes in detail the features, development process, and performance of various methods applied in this field. By comparing the advantages and disadvantages of these methods, it also discusses the challenges faced in the APR field and the possible directions for its future development. After discussion and comparison, it can be found that APR technology has made noticeable progress—especially the methods based on Large Language Models (LLMs) and hybrid agent methods. However, core challenges still exist, such as limitations of datasets and insufficient ability of models to understand code. In the future, it will be necessary to build more representative datasets, improve models' ability to understand complex code logic, optimize the cooperation mechanism between multiple models, and reduce reliance on specific tools.

**Keywords:** Automated Program Repair (APR), Large Language Model (LLM), Code Repair

## 1. Introduction

As software systems keep developing, various program bugs appear one after another. To make sure systems work properly, developers have to spend a lot of time and energy fixing these bugs. For this reason, Automated Program Repair (APR) is used to fix program bugs, cut down costs and improve efficiency—it has become an important research direction in the field of software engineering. The goal of APR is to use automatic methods to find and fix errors in programs, so as to make software development more efficient.

Looking at the timeline of APR's development, the early template-based methods worked for simple code situations. But as code became more and more complex, these template methods could no longer solve the problems. In recent years, with the development of artificial intelligence technology—especially the use of Large Language Models (LLMs)—

ISSN 2959-6157

the APR field has made great progress. Over the past three years, LLM-based methods have developed rapidly, and different LLM models have been updated one after another. At first, researchers tried to use Codex-based APR methods to do simple, small-scale code repair tests. Later, they used more models like GPT-4o and Code Llama to do experiments, aiming to make APR work better. For example, AdaPatcher—a recent LLM-based tool—is a two-stage framework, used to find bugs and fix code respectively [1]. The first stage is bug location (bug locator). Through Self-Debug Learning, AdaPatcher uses Code Diff samples to train the Bug Locator, which helps it find the exact positions of bugs. The second stage is program repair (Program Modifier). It introduces three technologies: Location-Aware Repair Learning, Hybrid Training for Selective Reference and Adaptive Preference Learning. These technologies make sure the fixed code is both correct and simple. This two-stage framework not only makes LLMs better at finding program bugs, but also effectively prevents the problem of reduced readability caused by rewriting the program.

During the development of automated program repair, many scholars have done in-depth research from different angles. For instance, researchers put forward a program repair technology based on semantic understanding [2]. It finds and fixes errors by deeply analyzing the semantics of code, and achieved good results in small code library tests. However, when dealing with large and complex projects, its efficiency and accuracy drop because of the complexity of semantic analysis and the limits of computing resources. Another research focuses on using reinforcement learning for program repair [2]. By constantly trying, correcting mistakes and optimizing, it lets the model learn effective repair strategies. But this method needs a lot of samples and computing resources during training, and its convergence speed is slow.

Besides, for specific problems, we can use manually built agents or less powerful low-level agents to set limits. For example, the hybrid method proposed by AAIS showed better results than traditional single methods in related evaluations. However, the existing APR methods still face many challenges, such as limited datasets and insufficient ability of models to understand code. This paper will deeply discuss and compare these mainstream methods, the common datasets and evaluation standards in this field, and the technical level of current mainstream methods. It will also analyze possible solutions based on existing problems.

## 2. Overview of Mainstream Methods

Generally speaking, in the research on automated program repair over the past three years, the main methods include template-based methods, methods based on Large Language Models (LLMs), and methods that combine agents and weak agents.

### 2.1 Template-based Methods

Systems like GenProg use some fixed conversion rules to deal with problems such as null pointer checks and array out-of-bounds. Although they may achieve good results for some simple and common errors, they have many shortcomings. They can only handle errors under certain formats and rules, and cannot solve uncertain problems. Moreover, they are completely ineffective when dealing with large-scale dynamic code libraries [3].

### 2.2 LLM-based Methods

These methods have become a research focus in recent years. Examples include models like GPT-4, CodeLlama, DeepSeekCoder, and Qwen2.5Coder. They have advantages such as strong context understanding, context-aware reasoning, and good natural language comprehension. They can learn common bug templates and understand the semantic intent of code from a large amount of code, enabling adaptive context-related repairs in complex code libraries. For instance, AdaPatcher is an LLM-based method with some customized improvements [1].

### 2.3 Methods Combining Agents and Low-level Agents

Represented by the AAIS method, this approach solves problems by integrating the adaptability of agents with the efficient control flow of low-level agents. Traditional agent-based methods allow LLMs to access code libraries and write test cases by building relatively complex Agent-Computer interfaces. While these methods have high fault tolerance and can solve common problems well, they require high reasoning costs and are prone to infinite loops. On the other hand, low-level agent-based methods extract the structure of the code library in advance through static analysis, and provide information to LLMs in the process of location-fix-selection to achieve program repair. This method saves reasoning costs relatively, but it is limited by the size of the LLM context window, which may lead to the loss of useful information.

AAIS introduces adaptive defect location and multi-model assisted generation. In the SWE-Bench benchmark test, compared with Agentless-1.5 (the current most advanced low-level agent method), the maximum function-level location accuracy of AAIS reaches 113.32%. Besides, AAIS has increased by 63.85%, 16.79%, and 64.43% respectively in other aspects. For the Issue solving task, AAIS has also improved by 35.67%.

# 3. Common Datasets and Evaluation Criteria

## 3.1 Common Datasets

Common datasets in this field include CodeNet [2], Defects4J [4], and SWE-Bench [5]. CodeNet has more than 14 million code samples and covers many kinds of programming languages; Defects4J is a defect dataset for Java projects; SWE-Bench focuses on complex real-world repair tasks.

## 3.2 Main Evaluation Criteria

The key indicators to assess the performance of APR methods include repair success rate, location accuracy, patch quality, and efficiency indicators. Among them, repair success rate is used to check whether the patch made after repair can help all test cases of the program pass. It is a basic indicator to measure how effective the repair is; location accuracy focuses on whether the method can accurately find the exact place where errors exist in the program, and it directly affects the efficiency and direction of the following repair steps; patch quality mainly assesses whether bad patch design will bring new errors during the repair process, and it is the key to ensuring the stability of the program after repair; efficiency indicators show whether the method can be used in real application scenarios by counting the time and hardware resources used in the repair process. These four indicators together form an evaluation system that fully measures the overall performance of APR methods, and provide a unified and objective reference standard for comparing different methods.

# 4. Development and Analysis of Mainstream Methods

When sorted out according to the timeline, the early template-based methods were suitable for simple code situations. But as code became more and more complex, template-based methods could not solve the problems. In the past three years, LLM-based methods have developed rapidly, and different LLM models have been updated one after another. For example, researchers first tried to use Codex-based APR methods for simple small-scale code tests. Later, they used more models like GPT-4o and Code Llama to do experiments and improve the effect of APR. Besides, for problems in a certain aspect, manually built agents or weaker low-level agents can be used for restriction. For instance, the hybrid method proposed by AAIS in 2025 has better results in relevant evaluations than traditional single methods.

From the perspective of performance improvement range, LLM-based methods have more advantages in the improvement range than template-based methods. Compared with template-based methods, the improvement range is quite large: the code accuracy rate can rise from the original 30%-40% to 60%-70%, and the code improvement rate can reach 15%-25%. Compared with Agentless-1.5 (the most advanced low-level agent method), the function-level location accuracy of AAIS has increased by 46.94%-113.32%.

According to the actual test data, Copilot Autofix uses artificial intelligence to fix vulnerabilities in open-source projects more than three times faster than manual repair. Moreover, its speed in fixing CRLF injection vulnerabilities, XSS vulnerabilities and SQL injection vulnerabilities is 7 times and 12 times that of manual repair respectively [6]. However, APR also has its own limitations. First, it is limited by the existing window size of the model and cannot handle large code libraries well. Second, the accuracy of program processing results in specific scenarios still needs to be improved. Third, it requires computing resources and costs. Therefore, limited by the current practical stage, APR is still difficult to play a role in a wide range of application scenarios.

# 5. Challenges and Prospects

## 5.1 Current Challenges

### 5.1.1 Limitations of LLM-based APR Models

Although LLMs have strong abilities in semantic understanding and generation, they still cannot fully „understand“ complex code logic. For example, when dealing with parts that contain complex data structures or algorithms, the repair patches they generate may seem compilable, but in fact, they do not match the original meaning of the code or have logical errors [7]. Besides, for large code libraries, the context window of LLMs will have an impact—during the analysis process, it is difficult to obtain comprehensive and effective context information, which may lead to failure in generating repair results.

### 5.1.2 Shortcomings of Hybrid Agent Models (e.g., AAIS)

Hybrid models that combine agents and low-level agents can improve repair effects to a certain extent, but they still have some shortcomings. First, with multi-model assistance, the cooperation between different models and

ISSN 2959-6157

the integration of information may not be optimal, which further affects the quality of patch generation. Second, this type of model needs some static analysis tools to do preparatory work. If these analysis tools have poor support for certain special code structures or programming languages, the effect of the entire repair process will be affected [8].

#### 5.1.3 Defects of Datasets

Existing datasets for automated program repair have many shortcomings:

The code defects in some datasets from early studies are generated through simple programming errors designed manually, which cannot well reflect the real situations of developers. For example, in the early days, most studies only used small datasets with simple programming task errors—these datasets have small scale and single scenarios, making it difficult to fully evaluate the performance of APR methods in real and complex scenarios.

When some related studies analyze errors in specific fields or certain types, they may encounter the problem of unbalanced data categories. For instance, when studying the automatic repair of concurrent defects, it is found that there are more samples of one type of concurrent error, while there are fewer samples of other types. This will make APR methods prone to overfitting on error types with more samples during training, and have poor effects on handling error types with fewer samples.

In addition, with the development of APR, the demand for zero-shot learning is also increasing. In fact, there may be some new types (such as new vulnerabilities or new code patterns) that are not included in the training set. Most existing datasets cannot well support the zero-shot learning process, so existing APR methods cannot be applied to such situations. For example, in the field of repairing blockchain smart contract code, its own code structure and vulnerability patterns are relatively new. At present, there are very few APR datasets corresponding to this code structure and vulnerability patterns for research, so it is difficult to directly apply some existing methods for effective repair [9,10].

### 5.2 Future Development Directions

#### 5.2.1 Building More Representative Datasets

Datasets need to cover real repair scenarios, balance data categories, and support zero-shot learning to solve the current problem of data limitations.

#### 5.2.2 Improving Models‘ Ability to Understand Complex Code Logic

Through technical optimization, enable models to more accurately “interpret” the code logic corresponding to complex data structures and algorithms, and reduce the generation of patches with logical errors.

#### 5.2.3 Optimizing Multi-Model Collaboration Mechanisms

Improve the collaboration efficiency and information integration quality of different models in hybrid agent models, and enhance the accuracy of patch generation.

#### 5.2.4 Reducing Dependence on Specific Tools

Reduce models’ dependence on specific static analysis tools, enhance the adaptability of models in scenarios with different code structures and programming languages, and expand the scope of technical application.

## 6. Conclusion

In summary, this paper conducts a systematic study on the field of Automated Program Repair (APR). First, it sorts out the mainstream technical paths in this field over the past three years, covering traditional template-based methods, advanced methods based on Large Language Models (LLMs), and hybrid methods that integrate agents and low-level agents. It also analyzes in detail the technical principles, core advantages, and applicable boundaries of each type of method—such as the limitations of template-based methods in fixing simple errors, the breakthroughs of LLM methods in understanding complex code semantics, and the innovations of hybrid methods like AAIS in balancing reasoning costs and repair accuracy. Second, this paper systematically organizes the common datasets in the APR field (such as CodeNet that covers multiple languages, Defects4J for Java projects, and SWE-Bench that focuses on actual complex tasks) and multi-dimensional evaluation criteria (repair success rate, location accuracy, patch quality, and efficiency indicators), providing an objective reference framework for comparing the performance of different methods. At the same time, through actual test data and case analysis, it quantifies the performance differences of various methods. For example, LLM methods have a 30%-40% improvement in code accuracy compared with template methods, and the AAIS method has a significant optimization effect on traditional low-level agent methods in the SWE-Bench benchmark test.

From the above research, it can be seen that automated program repair technology has made significant progress in recent years. In particular, the application based on LLMs has greatly promoted the practical possibility of complex code repair scenarios. However, the existing APR application methods still face challenges such as insufficient representativeness of datasets (e.g., less coverage of real scenarios, unbalanced categories, and weak support for zero-shot learning), insufficient in-depth understanding of complex code logic by models, the need to optimize the collaboration mechanism of hybrid models, and high dependence on specific tools. Future research should focus on building more representative datasets, improving the ability of models to understand complex code and its logic, optimizing the collaboration efficiency of multi-models, and reducing tool dependence. In this way, we can promote the wide application of APR technology in actual software development scenarios, effectively reduce the burden of developers in fixing vulnerabilities, and improve the efficiency and quality of software production.

Authors Contribution

All the authors contributed equally and their names were listed in alphabetical order.

## References

[1] Dai Z, Chen B, Zhao Z, Tang X, Wu S, Yao C, ... & Chen J. Less is More: Adaptive Program Repair with Bug Localization and Preference Learning. In Proceedings of the AAAI Conference on Artificial Intelligence. 2025. (Vol. 39, No. 1, pp. 128-136).

[2] Dikici S, Bilgin T T. Advancements in automated program repair: a comprehensive review. Knowledge and Information Systems, 2025: 1-47.

[3] Ji T, Qi Y H & Mao X G. Tool based on software automatic repair evaluation defect localization technology: GenProg-FL. Computer Science, 2014, 41(09), 88-91 124.

[4] Jiang J J, Xiong Y F & Xia X. A manual inspection of Defects4J bugs and its implications for automatic program repair. Science China (Information Sciences), 2019, 62(10), 31-46.

[5] Yu B X, Zhu Y X, He P J, and Kang D. UTBoost: Rigorous Evaluation of Coding Agents on SWE-Bench. In Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 3762–3774, Vienna, Austria. Association for Computational Linguistics. 2025.

[6] Ahmed E. AI Copilots: Boosting Software Engineers' Productivity or Replacing Them? SIGSOFT Softw. Eng. Notes 50, 3 2025, 12–13. https://doi.org/10.1145/3743095.3743098

[7] Tian Y, Yan W, Yang Q, Zhao X, Chen Q, Wang W, Luo Z, Ma L, & Song D. CodeHalu: Investigating Code Hallucinations in LLMs via Execution-based Verification. Proceedings of the AAAI Conference on Artificial Intelligence, 39(24), 25300-25308. 2025.

[8] Liu Y, Zhang L, Zhang Z. A Survey of Test Based Automatic Program Repair. J. Softw., 2018, 13(8): 437-452.

[9] Winter E, Nowack V, Bowes D, et al. Let's talk with developers, not about developers: A review of automatic program repair research. IEEE Transactions on Software Engineering, 2022, 49(1): 419-436.

[10] Puri R, Kung D S, Janssen G, Zhang W, Domeniconi G, Zolotov V, ... & Reiss F. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. arxiv preprint arxiv:2105.12655. 2021.