Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

HSR-APR: A Hybrid Secure Reflective Automated Program Repair Framework with Adaptive Reflection Memory

Esraa Fady

esraa_fady@yahoo.com

Menoufia university Egypt

Mahmoud Hussien

Menoufia university Egypt

Ashraf Elsisi

Menoufia university Egypt

Afaf Mosa

Menoufia university Egypt

Article

Keywords: Automated Program Repair, Hybrid APR, Secure Program Repair, Self-Reflection and Debugging, Retrieval-Augmented Generation, Adaptive Reflection Memory

Posted Date: November 9th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-10924086/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.

HSR-APR: A Hybrid Secure Reflective Automated Program Repair Framework with Adaptive Reflection Memory

Esraa Fady[1*], Mahmoud Hussien[1], Ashraf Elsisi[1], Afaf Mosa[1].
Computer Science Department, Faculty of Computers and Information, Minufiya University, Egypt.
Corresponding author: Esraa Fady. **Email**: esraa_fady@yahoo.com

Abstract

Automated Program Repair (APR) has benefited from the rapid development of Large Language Models (LLMs), but existing approaches still rely heavily on functional testing, provide limited security validation, insufficient integration of traditional program analysis, and have limited ability to learn from failed repair attempts. This research proposes HSR-APR (Hybrid Secure Reflective Automated Program Repair), a hybrid framework that integrates static analysis, fault localization, Retrieval-Augmented Generation (RAG), LLM-based patch generation, functional and security validation, self-reflection, and Adaptive Reflection Memory (ARM). The framework iteratively evaluates generated patches, analyzes failed attempts, stores repair experiences in ARM, and retrieves relevant knowledge to improve subsequent repair prompts. HSR-APR is evaluated on 300 SWE-Bench Lite instances using four configurations: Baseline LLM, Hybrid APR, Hybrid APR with Security Validation, and HSR-APR. Results show that HSR-APR achieves an 85% Repair Success Rate (255/300), compared with 63% for the Baseline LLM, while increasing the Test Pass Rate from 72% to 92% and Security Warning Reduction from 8% to 87%. It also reduces average repair iterations from 5.8 to 2.9 and execution time from 18 to 13 minutes, demonstrating that combining hybrid analysis, security validation, self-reflection, and adaptive memory improves the effectiveness, security, and efficiency of LLM-based APR.

Keywords Automated Program Repair; Hybrid APR; Secure Program Repair; Self-Reflection and Debugging; Retrieval-Augmented Generation; Adaptive Reflection Memory.

Introduction

Automated Program Repair (APR) is an important research area in software engineering that aims to automatically identify and correct defects in software systems. Traditional APR techniques typically rely on program analysis, fault localization, predefined repair templates, heuristic search, or constraint-solving techniques to generate candidate patches [1]. Although these approaches have achieved considerable progress, their effectiveness is often constrained by the complexity of real-world software defects, the diversity of programming patterns, and the difficulty of generating patches that correctly capture the intended program behavior [14].

The emergence of Large Language Models (LLMs) has introduced a new direction in automated program repair. LLMs have demonstrated strong capabilities in code understanding, code generation, and reasoning over natural-language descriptions of software defects [2], [15]. Consequently, LLM-based APR approaches have increasingly been investigated for generating patches directly from bug reports, source-code context, test failures, and repository information. Unlike traditional template-based APR

techniques, LLM-based systems can generate more flexible patches and potentially address a wider range of defect types [21].

Despite these advances, existing LLM-based APR systems still face several important limitations. First, many approaches primarily depend on functional test cases to determine whether a generated patch is acceptable. Although test-based validation is essential, passing the available tests does not necessarily guarantee that a patch is correct for all possible program behaviors or that it does not introduce security vulnerabilities [6], [7]. Second, security is often treated as a separate concern rather than as an integral part of the repair process. Consequently, an LLM may generate a functionally valid patch that introduces insecure programming patterns or fails to eliminate an existing vulnerability. Third, LLM-based repair can benefit from traditional program analysis techniques, but the integration of static analysis, fault localization, retrieval mechanisms, and LLM-based generation remains limited in many existing approaches [7], [15].

Another important limitation concerns the iterative nature of automated repair. A repair system may generate several unsuccessful patches before producing an acceptable solution. In conventional approaches, information obtained from failed attempts is often discarded, causing the system to repeatedly explore similar unsuccessful repair strategies [11]. This limitation motivates the use of self-reflection and self-debugging mechanisms that allow the repair agent to analyze its own failures and modify subsequent repair attempts. However, simply generating a reflection after a failed attempt is not sufficient if the resulting knowledge is not systematically retained and reused across iterations [23], [24].

To address these limitations, this research proposes HSR-APR (Hybrid Secure Reflective Automated Program Repair), a hybrid framework that combines traditional program analysis with LLM-based repair, security validation, self-reflection, and adaptive repair memory. The proposed framework begins with static analysis and fault localization to identify suspicious code regions and provide structured information about the reported defect. Retrieval-Augmented Generation (RAG) is then used to retrieve relevant repository context and provide the LLM Repair Agent with information that can support more accurate patch generation.

After generating a candidate patch, HSR-APR performs functional validation using the available test suite and performs an additional security validation stage using security-analysis tools. This design directly addresses the limitation of relying exclusively on functional tests. If the generated patch fails functional or security validation, the Reflection Module analyzes the failure, identifies the likely failure cause, and produces a structured reflection record. The record is stored in the Adaptive Reflection Memory (ARM), which maintains information about previous repair attempts, failed tests, security warnings, failure causes, reflection summaries, and successful repair strategies. During subsequent iterations, relevant memory records are retrieved and incorporated into an improved repair prompt, allowing the system to use knowledge obtained from previous repair attempts.

The proposed framework is evaluated using the complete 300-instance test subset of SWE-Bench Lite. Four configurations are compared to examine the contribution of the proposed components: Baseline LLM, Hybrid APR, Hybrid APR with Security Validation, and the complete HSR-APR framework. The experimental results show a progressive improvement across the configurations. The Repair Success Rate increases from 63% for the Baseline LLM to 71% for Hybrid APR, 75% for Hybrid APR with Security Validation, and 85% for HSR-APR. Similarly, the Test Pass Rate increases from 72% to 92%, while Security Warning Reduction increases from 8% to 87%. HSR-APR also reduces the average number of repair iterations from 5.8 to 2.9 and the average execution time from 18 minutes to 13 minutes.

These results provide evidence that the limitations identified in existing LLM-based APR can be addressed through the integrated design of HSR-APR. In particular, the security-validation component addresses the lack of explicit security assessment, the hybrid analysis and retrieval components improve contextual repair information, and the Reflection Module combined with Adaptive Reflection Memory enables the system to learn from unsuccessful repair attempts. Therefore, HSR-APR moves automated program repair from a single-step patch-generation process toward an iterative and knowledge-aware repair process that considers functional correctness, security, and repair efficiency.

The main contributions of this research are summarized as follows:

First, we propose **HSR-APR**, a hybrid automated program repair framework that integrates Large Language Models, Static Code Analysis, Retrieval-Augmented Generation, Functional Validation, and Security Validation within a unified repair pipeline. Second, we introduce an **Adaptive Reflection Memory (ARM)** mechanism that continuously records failure causes, security violations, and successful repair strategies to improve subsequent repair iterations. Third, we design a security-aware validation process that verifies generated patches using static security analysis tools before accepting them as valid software repairs. Finally, we evaluate the proposed framework on the SWE-Bench Lite benchmark and compare its performance with conventional LLM-based APR approaches using functional, security, and efficiency metrics.

The remainder of this paper is organized as follows: Section 2 reviews related work in automated program repair, LLM-based approaches, security validation, and reflection memory mechanisms. Section 3 highlights the identified research gaps that motivate our proposed framework. Section 4 presents the detailed architecture and components of the HSR-APR framework, including the repair algorithm and Adaptive Reflection Memory (ARM) update procedure. Section 5 describes the experimental setup, dataset, configurations, and evaluation metrics. Section 6 presents and discusses the experimental results on SWE-Bench Lite. Finally, Section 7 concludes the paper and outlines future research directions.

Related Works

The rapid evolution of Automated Program Repair (APR) has led to significant improvements in software maintenance and debugging. Early APR techniques primarily relied on handcrafted repair rules and search-based optimization, while recent advances in Large Language Models (LLMs) have fundamentally transformed the field by enabling intelligent code understanding and automated patch generation. As illustrated in Fig. 1, the field has progressively evolved from Traditional APR to LLM-Based APR, Agentic APR, and Hybrid APR, leading to the proposed HSR-APR framework, which integrates hybrid program analysis, security validation, reflection, and adaptive memory. This evolution provides the foundation for the research directions discussed in the following sections.

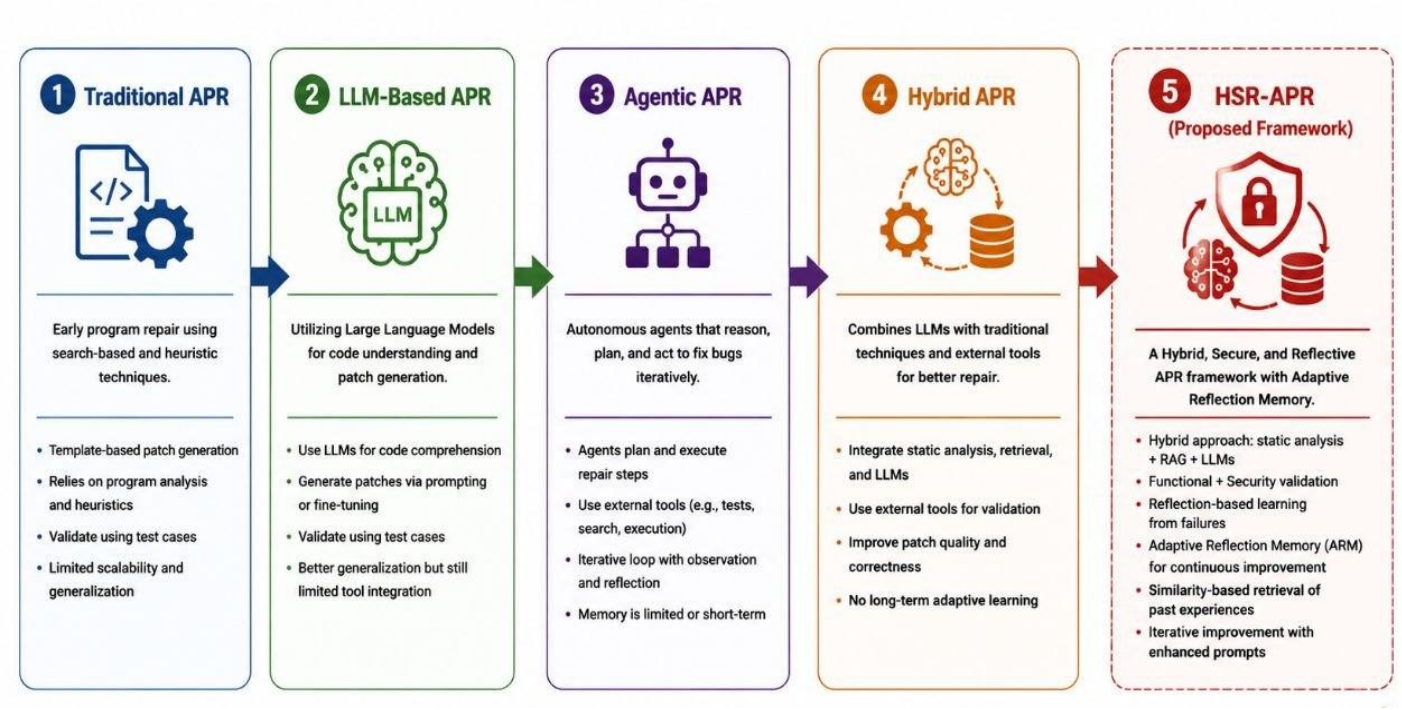


Figure 1: Evolution of Automated Program Repair.

Traditional Automated Program Repair

Traditional APR approaches generate software patches using predefined repair templates, search-based algorithms, genetic programming, or constraint-solving techniques. Representative systems such as GenProg, PAR, and Angelix demonstrated that automatically generated patches could successfully repair a subset of software defects. However, these approaches heavily depend on manually designed repair patterns and limited semantic understanding of source code. Consequently, their performance decreases significantly when handling complex software repositories, previously unseen bug types, or large-scale applications [1], [2].

Although traditional APR established the foundation of automatic software repair, its limited flexibility motivated researchers to investigate learning-based repair techniques capable of understanding programming semantics [1], [2].

LLM-Based Automated Program Repair

The emergence of Large Language Models has dramatically changed the APR landscape. Modern code-oriented LLMs are capable of understanding program semantics, generating source code, and reasoning about software behavior.

Current LLM-based APR approaches can generally be categorized as follow:

Prompt-Based Repair: Prompt-based approaches directly provide buggy code, error messages, and repair instructions to an LLM without additional model training. These methods are simple to deploy and benefit from the extensive programming knowledge embedded in foundation models [4]. However, their performance is highly dependent on prompt quality and contextual information, while generated patches often lack systematic validation and iterative improvement.

Fine-Tuning-Based Repair: Fine-tuning approaches adapt pre-trained code models using large bug-fix datasets to improve repair accuracy. Models such as RepairLLaMA demonstrate that task-specific fine-tuning can significantly improve repair quality compared with generic prompting. Nevertheless, these approaches require substantial computational resources and large-scale training datasets, limiting their applicability in practical software engineering environments [3].

Agentic Program Repair: Recent studies have introduced agent-based repair systems that treat the LLM as an autonomous software engineering agent capable of repository exploration, bug localization, patch generation, and iterative debugging. Representative systems including RepairAgent, AutoCodeRover, and PatchAgent have demonstrated remarkable performance on repository-level benchmarks such as SWE-Bench. These systems extend traditional prompting by enabling tool usage and multi-step reasoning throughout the repair process [5], [6], [7], [8]. Despite these advances, most existing agentic systems primarily evaluate repair correctness through functional test execution while providing limited support for security validation or long-term learning from unsuccessful repair attempts [5], [7], [26].

Hybrid Automated Program Repair

Hybrid APR combines multiple software engineering techniques to improve repair quality. Recent studies integrate static code analysis, fault localization, retrieval mechanisms, and LLM-based code generation into unified repair pipelines. Static analysis helps identify suspicious code regions before repair, while Retrieval-Augmented Generation (RAG) provides relevant repository context, documentation, or historical fixes to improve patch generation [4], [6], [11].

Although hybrid approaches improve repair accuracy, current research typically integrates only a subset of these techniques. Comprehensive frameworks combining static analysis, contextual retrieval, iterative reasoning, and security-aware validation remain limited [6], [11].

Security-Aware Program Repair

As LLM-generated patches become increasingly common, software security has emerged as an important research concern. Recent studies have shown that patches capable of passing functional test suites may still introduce security vulnerabilities such as command injection, insecure API usage, unsafe deserialization, or hard-coded secrets [9], [16].

To address this issue, several security-oriented repair approaches incorporate static security analysis tools during patch validation. While these methods improve software security, they primarily focus on vulnerability repair rather than general-purpose automated program repair and rarely integrate adaptive reasoning mechanisms for iterative improvement [9], [16], [17], [18].

Self-Reflection and Memory in LLM-Based Software Engineering

Self-reflection has recently emerged as an effective strategy for improving LLM reasoning by allowing models to analyze previous failures before generating improved responses. In software engineering applications, reflective reasoning enables repair systems to identify the causes of unsuccessful repair attempts and refine subsequent patches accordingly [10].

More recently, memory-based reasoning has attracted increasing attention within agentic AI systems. Instead of treating repair attempts independently, memory mechanisms preserve information regarding previous failures, successful repair strategies, and execution feedback. This accumulated knowledge enables future repair iterations to avoid repeated mistakes and progressively improve repair

quality. However, the integration of structured reflection memory into automated program repair remains largely unexplored [10].

Study	Year	LLM-Based	Agentic	Hybrid	RAG	Security Validation	Reflection	Memory	Benchmark
RepairLLaMA [3]	2024	✓	✗	✗	✗	✗	✗	✗	HumanEval-Java
RepairAgent [5]	2024	✓	✓	Partial	✗	✗	Partial	✗	Defects4J
AutoCodeRover [6]	2024	✓	✓	✓	✓	✗	✗	✗	SWE-Bench Lite
SWE-Agent [7]	2024	✓	✓	Partial	Partial	✗	✗	✗	SWE-Bench Lite
PatchAgent [8]	2025	✓	✓	✓	Partial	Partial	Partial	✗	Security Benchmarks
SecureFixAgent [16]	2025	✓	Partial	✓	✗	✓	Partial	✗	Python Security Dataset
How Safe Are AI-Generated Patches? [9]	2025	✓	✓	✗	✗	✓	✗	✗	SWE-Bench
HSR-APR (Proposed)	2026	✓	✓	✓	✓	✓	✓	(ARM)	SWE-Bench Lite

Table 1. Comparison of Recent LLM-Based Automated Program Repair Approaches

Table 1 highlights the differences between recent LLM-based Automated Program Repair approaches and the proposed HSR-APR framework. Existing studies have explored individual aspects such as agent-based repair, retrieval-augmented generation, or security validation. However, none of the reviewed approaches jointly integrate hybrid repair, repository-aware retrieval, security validation, iterative self-reflection, and a structured memory mechanism within a unified repair pipeline. The proposed HSR-APR framework addresses this gap by combining these complementary techniques into a single architecture designed for secure and iterative automated program repair [3], [5], [6], [7], [8], [9], [16].

Proposed Framework

This section presents the proposed HSR-APR framework and explains how its main components work together to achieve secure, iterative, and context-aware automated program repair. The framework

integrates program analysis, retrieval-based contextual information, LLM-based patch generation, functional and security validation, reflection, and adaptive memory into a unified repair process. The following subsections describe the framework architecture and the interaction among its components in detail.

Overview of the Proposed Framework

The proposed framework, named HSR-APR (Hybrid Secure Reflective Automated Program Repair), is designed to address the limitations identified in existing LLM-based Automated Program Repair systems [6]. The framework combines traditional program analysis techniques with Large Language Models, Retrieval-Augmented Generation, functional and security validation, self-reflection, and Adaptive Reflection Memory. The main objective of HSR-APR is to transform automated program repair from a single-step patch-generation process into an iterative, context-aware, security-aware, and learning-oriented repair process. Instead of generating a patch and relying only on functional tests to determine its quality, the proposed framework continuously analyzes, validates, reflects on, and improves generated patches.

Figure 2 illustrates the overall workflow of HSR-APR and the interaction among its major components. The process starts with a Bug Report and progresses through Static Analysis, Fault Localization, Retrieval-Augmented Generation, LLM-based patch generation, Functional Validation, Security Validation, Reflection, and Adaptive Reflection Memory. When a candidate patch fails validation, the failure information is analyzed and stored in the memory, allowing the system to generate an improved prompt and iteratively refine the repair in subsequent attempts. If the generated patch fails validation, the system analyzes the failure, stores the resulting knowledge in memory, and generates an improved prompt for the next repair iteration.

The complete HSR-APR workflow can be summarized as follows:

The complete HSR-APR workflow begins by analyzing the buggy repository using Static Analysis, followed by fault localization to identify the likely source of the defect. Relevant repository context is then retrieved using Retrieval-Augmented Generation (RAG) to support the Repair Agent in generating a candidate patch. The generated patch undergoes Functional Validation and Security Validation to assess both its correctness and security. If the patch passes both validation stages, it is accepted as a valid repair. Otherwise, the Reflection Module is invoked to analyze the failure and identify potential causes and improvement strategies. The resulting repair experience is then stored in the Adaptive Reflection Memory (ARM), and relevant information is used to generate an improved repair prompt. The Repair Agent subsequently generates a new candidate patch based on the improved prompt, and the validation process is repeated iteratively until a valid patch is obtained or the maximum number of repair attempts is reached.

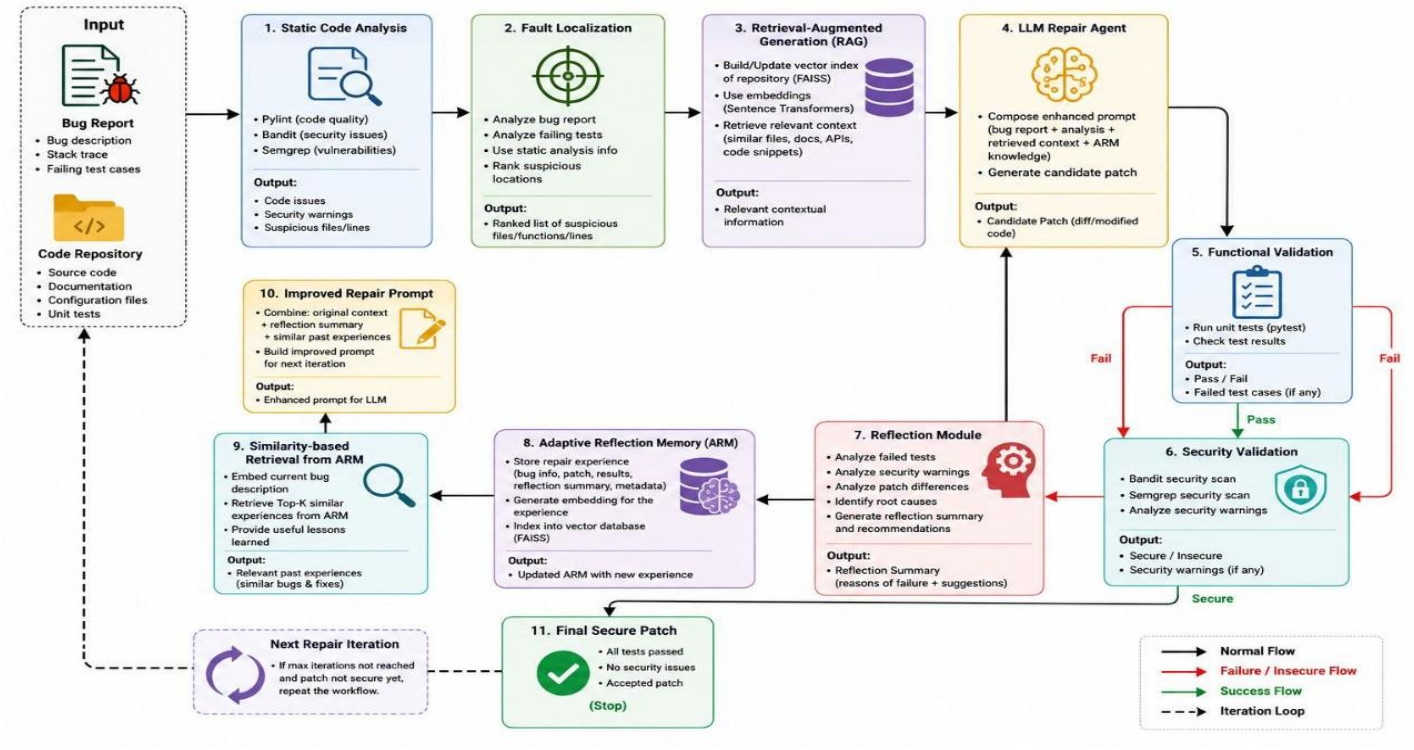


Figure 2: Overall Architecture of the Proposed HSR-APR Framework.

The integration of hybrid analysis, adaptive reflection, and security-aware validation enables HSR-APR to produce more reliable, secure, and context-aware software repairs than conventional LLM-based APR approaches [9], [10], [11]. The process continues until a patch satisfies the required validation criteria or the maximum number of repair iterations is reached.

Bug Report Processing

The repair process starts with the bug report associated with the target software engineering issue. The bug report provides the natural-language description of the problem and may include information about the expected behavior, observed behavior, error messages, and relevant repository components [7].

HSR-APR uses this information as the initial input for the analysis and repair process. The bug report is combined with repository information and subsequent analysis results to construct a contextual representation of the repair problem.

Static Analysis

The Static Analysis Module examines the target repository before patch generation. Its purpose is to provide structured information about the source code and identify potential code-quality and security-related issues [17].

Pylint is used to analyze code-quality characteristics, while Bandit and Semgrep provide security-oriented analysis [18]. The generated reports can contain information about suspicious statements, warning categories, source-code locations, and potential vulnerabilities. This stage addresses part of the limitation associated with relying exclusively on LLM reasoning. Instead of asking the LLM to

independently identify all relevant program characteristics, the framework provides analysis results that can guide the repair process.

Fault Localization

Following static analysis, the Fault Localization Module identifies the most suspicious files, functions, and code regions associated with the reported defect. Fault localization combines information from the bug report, source-code structure, test failures, stack traces, and static-analysis findings. The objective is to reduce the search space and identify the parts of the repository that are most likely responsible for the observed defect [24]. The localized information is subsequently provided to the retrieval and repair stages, allowing the LLM to focus on relevant program regions instead of processing the entire repository indiscriminately [19].

Retrieval-Augmented Generation

The Retrieval-Augmented Generation component provides repository-specific context to the Repair Agent. Relevant source-code fragments and repair-related information are transformed into vector representations using Sentence Transformers. FAISS is then used to index these representations and retrieve information that is semantically related to the current repair problem.

The highest-ranked information is incorporated into the repair prompt. This enables the Repair Agent to use contextual information from the repository and relevant previous repair experiences. RAG therefore complements the LLM by providing additional evidence that may not be available from the original bug report alone [11].

LLM-Based Repair Agent

The Repair Agent is the primary patch-generation component of HSR-APR. It receives a structured repair prompt containing the bug description, localized source-code regions, static-analysis findings, retrieved repository context, and, when available, relevant reflection records from ARM. Based on this information, the LLM generates a candidate patch intended to correct the identified defect while preserving the expected behavior of the software [10].

The Repair Agent does not operate independently from the other modules. Instead, it participates in an iterative feedback loop in which the results of functional and security validation are used to improve subsequent repair attempts. Algorithm 1 presents the overall iterative repair process of HSR-APR. The algorithm starts with the buggy repository and bug report, followed by Static Analysis, Fault Localization, and RAG to provide relevant contextual information to the LLM Repair Agent. The generated patch is then evaluated through Functional and Security Validation. If the patch passes both validations, it is accepted as the final repair. Otherwise, the Reflection Module analyzes the failure, updates the Adaptive Reflection Memory (ARM), and generates an improved prompt for the next repair iteration. This process continues until a successful patch is obtained or the maximum number of iterations is reached.

Input: Buggy Repository $\mathcal{R}$
Bug Report $\mathcal{B}$

Output: Secure Repaired Patch P

```
Perform Static Analysis(R)                ▷ Detect code issues and vulnerabilities
Locate suspicious files                   ▷ Fault localization
Retrieve contextual information using RAG ▷ Get relevant repository context
Initialize Adaptive Reflection Memory ARM ▷ ARM ← Ø
repeat
    Generate Patch using LLM              ▷ Use prompt with context & ARM
    Execute Functional Validation         ▷ Run unit tests (e.g., pytest)
    Execute Security Validation           ▷ Run security tools (Bandit, Semgrep)
    if Patch passes all validations then  ▷ Functional & Security pass
        return Patch (P)                  ▷ Secure repaired patch
    else
        Analyze failure using Reflection Module  ▷ Identify causes & suggestions
        Update Adaptive Reflection Memory (ARM)  ▷ Store experience in ARM
        Generate Improved Prompt          ▷ Enhance prompt using reflection
                                            and similar past experiences
    end if
until Maximum Iterations reached          ▷ Stop if iteration limit is reached
return Failure                            ▷ No valid secure patch found
```

Algorithm 1. HSR-APR Repair Algorithm

Functional Validation

After generating a candidate patch, HSR-APR performs Functional Validation to determine whether the patch satisfies the expected functional behavior. The framework executes the relevant test cases using pytest. The resulting test reports provide information about passed and failed tests.

If the patch passes the required functional validation, it proceeds to the Security Validation stage. If functional validation fails, the failure information is forwarded to the Reflection Module for analysis. This mechanism ensures that functional feedback becomes an explicit part of the iterative repair process [17].

Security Validation

A key component of HSR-APR is the explicit Security Validation stage. Functional tests alone cannot guarantee that a generated patch is secure. Therefore, HSR-APR uses Bandit and Semgrep to analyze the generated patch and identify security warnings or potentially vulnerable programming patterns [18].

The Security Validation stage evaluates whether the generated patch introduces new security problems or fails to address existing security-related issues. A patch is considered acceptable only when

it satisfies the defined functional and security requirements. This design directly addresses the research gap related to the lack of integrated security validation in LLM-based APR.

Reflection Module

If a candidate patch fails functional or security validation, the Reflection Module is activated. Rather than simply requesting another patch from the LLM, the Reflection Module analyzes why the previous repair attempt failed. It considers information such as failed test cases, security warnings, the generated patch, and the detected failure cause [10].

The module produces a structured reflection summary describing what went wrong and what should be changed in the next repair attempt. This process implements the self-reflection and self-debugging aspect of HSR-APR.

Adaptive Reflection Memory

The Adaptive Reflection Memory is one of the main contributions of the proposed framework. ARM stores structured information from previous repair attempts so that the system can reuse successful strategies and avoid repeating unsuccessful ones. Each memory record can contain the bug identifier, repair attempt ID, generated patch summary, failed test cases, security warnings, failure cause, reflection summary, and successful repair strategy [10]. Algorithm 2 describes the process of updating the Adaptive Reflection Memory (ARM) after each repair attempt. It stores the generated patch, failed test cases, security warnings, failure causes, and reflection summary. If the repair is successful, the successful repair strategy is also recorded. The resulting reflection record is then stored in ARM, and previous records are ranked according to their similarity to the current problem so that relevant repair experiences can be retrieved in subsequent iterations.

Input: Repair Attempt A
Functional Validation Report F
Security Validation Report S
Current Reflection Memory ARM

Output: Updated Adaptive Reflection Memory ARM

```
Initialize Reflection Record R                          ▷ Create a new reflection record
Store Repair Attempt ID (A.id) in R                     ▷ Unique identifier of the attempt
Store Bug Identifier (A.bug_id) in R                    ▷ Link to the corresponding bug
Store Generated Patch (A.patch) in R                    ▷ Save the generated patch
if Functional Validation Failed then                    ▷ Check functional validation result
    Extract Failed Test Cases (F.failed_tests)          ▷ Get failed test cases
    Identify Failure Cause (F.cause)                    ▷ Analyze why tests failed
    Add Failure Reason to R                             ▷ Store functional failure info
end if
if Security Validation Failed then                      ▷ Check security validation result
    Extract Security Warnings (S.warnings)              ▷ Get security tool warnings
    Classify Vulnerability Type (S.type)                ▷ e.g., SQL Injection, XSS, etc.
    Add Security Information to R                       ▷ Store vulnerability details
end if
if Patch Successfully Repaired Bug then                 ▷ Check overall repair outcome
    Record Successful Repair Strategy (A.strategy)      ▷ Store useful strategy/insight
end if
Generate Reflection Summary (R.summary)                 ▷ Summarize causes and lessons
Save Reflection Record R into ARM                       ▷ Persist the record in memory
Rank Previous Records in ARM by Similarity              ▷ Update index for retrieval
return Updated ARM                                      ▷ Return the updated memory
```

Algorithm 2. Adaptive Reflection Memory (ARM) Update Algorithm

Improved Prompt Generation

After reflection and memory retrieval, HSR-APR generates an improved repair prompt. The improved prompt incorporates the original problem description together with the newly identified failure cause, validation feedback, security information, and relevant historical repair experiences retrieved from ARM. The Repair Agent then uses this improved prompt to generate a new candidate patch.

This process creates an iterative repair loop:

Candidate Patch → Validation → Failure Analysis → Reflection → Memory Update → Improved Prompt → New Candidate Patch

The loop continues until the generated patch satisfies the required validation criteria or the predefined maximum number of iterations is reached.

Overall Repair Process

The overall operation of HSR-APR can be summarized as a sequence of interconnected stages rather than independent modules. The Bug Report provides the initial problem description, while Static Analysis and Fault Localization provide structured information about the suspected defect. RAG enriches the repair context, and the LLM Repair Agent generates a candidate patch.

The candidate patch is then evaluated through Functional Validation and Security Validation. A successful patch is returned as the final repair. If the patch fails, the Reflection Module identifies the cause of failure and updates the Adaptive Reflection Memory. Relevant previous experiences are retrieved from ARM and used to construct an improved prompt for the next repair attempt.

Through this iterative process, HSR-APR directly addresses the four identified research gaps. Hybrid program analysis and RAG address the limited integration of traditional analysis techniques, Security Validation addresses the lack of explicit security assessment, Reflection enables learning from unsuccessful attempts, and ARM provides a structured mechanism for retaining and reusing repair knowledge.

The experimental results further support this design. HSR-APR achieves an 85% Repair Success Rate compared with 63% for the Baseline LLM, increases Test Pass Rate from 72% to 92%, improves Security Warning Reduction from 8% to 87%, reduces Average Repair Iterations from 5.8 to 2.9, and reduces Average Execution Time from 18 minutes to 13 minutes.

These results indicate that the proposed components work together to improve not only the ability of the system to generate successful patches, but also its security awareness and efficiency during iterative automated program repair.

Experimental Setup

This section describes the methodology adopted to design and evaluate the proposed HSR-APR framework. It explains the experimental workflow, dataset, implementation components, repair process, validation mechanisms, and evaluation criteria used to assess the effectiveness of the proposed approach. The following subsections provide detailed descriptions of each stage of the methodology.

Experimental Objective

The experimental evolution is designed to assess the effectiveness of the proposed HSR-APR framework and to determine the contribution of its major components. In particular, the experiment investigates whether integrating hybrid program analysis, retrieval-augmented generation, security validation, self-reflection, and Adaptive Reflection Memory can improve automated program repair compared with a conventional LLM-based repair approach.

To provide a controlled evolution, the proposed framework is compared with progressively enhanced repair configurations. This design makes it possible to observe the performance improvement associated with each major addition to the repair pipeline rather than evaluating HSR-APR only against a single baseline.

System Architecture and Repair Workflow

The HSR-APR methodology follows a sequential and iterative repair pipeline that integrates program analysis, retrieval, LLM-based patch generation, functional validation, security validation, and reflection. The process starts with the Bug Report, which describes the reported defect and its expected behavior when available.

First, Static Analysis examines the repository to identify relevant code information and potential security issues. Fault Localization then combines the bug information, test results, and analysis findings to identify suspicious files and code regions. The resulting context is provided to Retrieval-Augmented Generation (RAG), which retrieves relevant repository information and, when available, previous repair experiences from Adaptive Reflection Memory (ARM). This information is combined with the bug report and analysis results to construct an enriched prompt for the LLM Repair Agent, which generates a candidate patch.

The generated patch is evaluated through Functional Validation and Security Validation. A patch is accepted only if it satisfies both functional and security requirements. If either validation stage fails, the results are passed to the Reflection Module, which analyzes the failure and stores the resulting repair experience in ARM. Relevant experiences are then retrieved to construct an improved prompt, and the Repair Agent generates another patch. This generate–validate–reflect–retrieve–repair cycle continues until a valid patch is obtained or the maximum number of repair iterations is reached.

Dataset

The experimental evolution is conducted using SWE-Bench Lite. The study evaluates the complete official test subset consisting of 300 instances. Each instance represents a real-world software engineering issue associated with a repository and provides the information required to reproduce and evaluate the repair task [14].

The same 300 instances are used across all experimental configurations to ensure a controlled comparison. The configurations include the Baseline LLM, Hybrid APR, Hybrid APR with Security Validation, and the complete HSR-APR framework. Using an identical evolution set allows differences in performance to be attributed more directly to the repair methodology rather than to differences in the evaluated problems [15].

Implementation Environment

The proposed framework as shown in Table 2 is implemented in Python because of its extensive support for software analysis, machine learning, testing, and security-analysis tools. The LLM Repair Agent is implemented using GPT-5 / Qwen2.5-Coder as the code-generation component.

Sentence Transformers is used to generate semantic embeddings for repository information and repair experiences. FAISS is used as the vector retrieval mechanism for identifying relevant contextual information. Pylint is used for static code analysis, while Bandit and Semgrep are used for security-oriented analysis. Functional validation is performed using pytest and the available testing infrastructure associated with the evaluated repository.

Component	**Technology**
Programming Language	Python
Large Language Model (Repair Agent)	GPT-5 / Qwen2.5-Coder

Benchmark	SWE-Bench Lite
Test Instances	300
Static Analysis	Pylint
Security Analysis	Bandit, Semgrep
Testing Framework	Pytest
Retrieval Engine	FAISS
Embedding Model	sentence-transformers
Version Control	Git

Table 2. Proposed Framework Is Implemented

These tools collectively implement the major stages of the proposed methodology, including program analysis, retrieval, patch generation, functional validation, security validation, and reflection-based repair.

Experimental Configurations

Four experimental configurations are evaluated to investigate the contribution of the proposed HSR-APR components.

The **first** configuration, Baseline LLM, represents a conventional LLM-based repair setting in which the model generates candidate patches using the available bug and source-code context without the complete hybrid analysis, security validation, reflection, and adaptive-memory mechanisms.

The **second** configuration, Hybrid APR, extends the baseline by incorporating Static Analysis and Retrieval-Augmented Generation. This configuration is designed to evaluate whether providing structured program-analysis information and repository-specific contextual information improves the repair process.

The **third** configuration, Hybrid APR + Security Validation, extends Hybrid APR by introducing explicit security validation using Bandit and Semgrep. This configuration evaluates the contribution of security-aware patch validation and allows the effect of security analysis to be compared directly with the previous configuration.

The **fourth** configuration is the complete HSR-APR framework. It integrates Static Analysis, Fault Localization, RAG, LLM-based patch generation, Functional Validation, Security Validation, Reflection, and Adaptive Reflection Memory. This configuration represents the complete proposed methodology and is used to evaluate the combined contribution of the proposed components.

The progressive structure of these configurations provides an ablation-oriented comparison in which the effect of adding major components can be observed systematically.

Evaluation Metrics

The performance of the four configurations is evaluated using Repair Success Rate, Test Pass Rate, Security Warning Reduction, Average Repair Iterations, and Average Execution Time.

Repair Success Rate measures the percentage of benchmark instances for which an acceptable repair is successfully generated. It is calculated as:

$$RSR = \frac{N_{successful}}{N_{total}} \times 100$$

where N_successful represents the number of successfully repaired instances and N_total represents the total number of evaluated instances.

Test Pass Rate measures the proportion of evaluated tests successfully passed by the generated patches. Security Warning Reduction measures the reduction in security warnings after applying the generated repair and is calculated as:

$$SWR = \frac{W_{before} - W_{after}}{W_{before}} \times 100$$

where W_before represents the number of security warnings before repair and W_after represents the number remaining after repair.

Average Repair Iterations measures the number of repair attempts required to obtain an acceptable patch, while Average Execution Time measures the average time required to process an instance from the beginning of the repair procedure to its final outcome. It is calculated as:

$$ARI = \frac{\sum_{i=1}^{N} I_i}{N}$$

where (I_i) represents the number of repair iterations required for instance (i), and (N) represents the total number of evaluated instances.

For HSR-APR, the reported Average Repair Iterations is 2.9, compared with 5.8 for the Baseline LLM. This indicates that the proposed reflection and adaptive-memory mechanisms reduce the number of iterations required to reach a successful repair.

The Average Execution Time (AET) measures the mean time required to process the evaluated instances, from the beginning of the repair process until a successful repair or termination condition is reached. It is calculated as:

$$AET = \frac{\sum_{i=1}^{N} T_i}{N}$$

where (T_i) represents the total execution time required for instance (i), and (N) represents the total number of evaluated instances.

These metrics were selected to evaluate the proposed framework from multiple perspectives. Repair Success Rate and Test Pass Rate measure functional effectiveness, Security Warning Reduction evaluates

security awareness, and Average Repair Iterations and Average Execution Time measure repair efficiency.

Results and Discussion

This section presents the experimental setup used to evaluate HSR-APR under controlled and reproducible conditions. It describes the benchmark dataset, implementation environment, experimental configurations, evaluation procedure, and performance metrics used for comparison. The setup is designed to provide a fair comparison between the proposed framework and the baseline approaches.

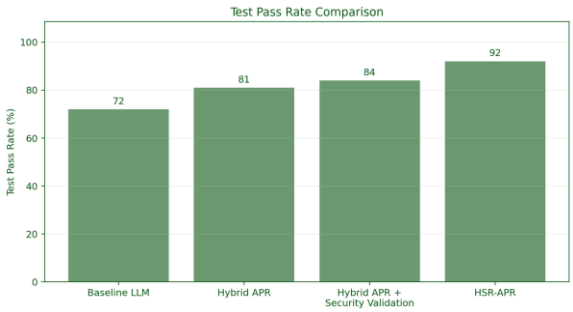
Overall Results

The evolution was conducted on the complete 300-instance SWE-Bench Lite test subset. The results obtained by the four configurations are presented below.

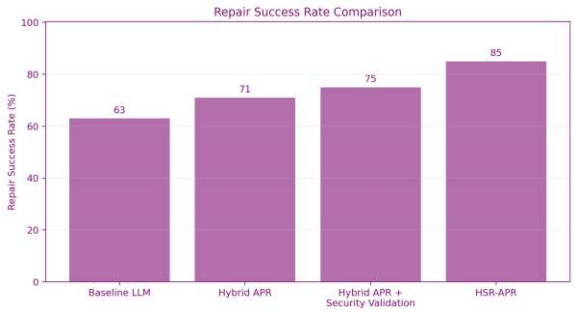
Method	Successful Repairs	Repair Success Rate	Test Pass Rate	Security Warning Reduction	Avg. Repair Iterations	Avg. Execution Time
Baseline LLM	189/300	63%	72%	8%	5.8	18 min
Hybrid APR	213/300	71%	81%	22%	4.6	16 min
Hybrid APR + Security Validation	225/300	75%	84%	68%	4.2	17 min
HSR-APR	255/300	85%	92%	87%	2.9	13 min

Table 3. Experimental Results of the Evaluated APR Configurations

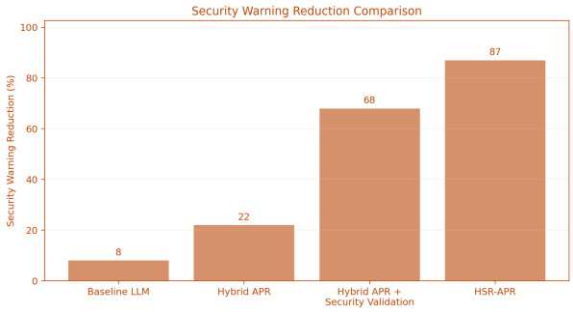
The results show a consistent improvement as additional components are introduced into the repair pipeline. The complete HSR-APR framework achieves the best performance across all five reported metrics.



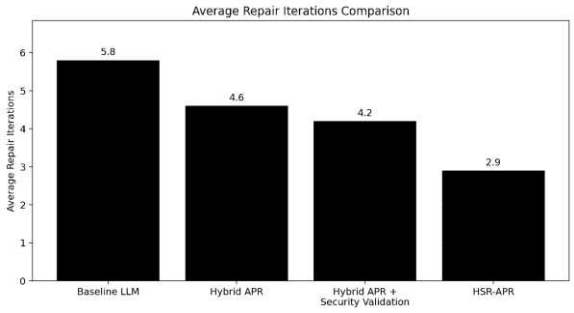
Repair Success Rate across the evaluated configurations



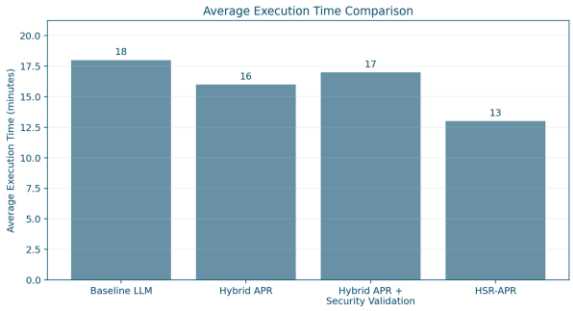
Test Pass Rate across the evaluated configurations



Security Warning Reduction across the evaluated configurations



Average Repair Iterations across the evaluated configurations



Average Execution Time across the evaluated configurations

The bar charts make the progressive contribution of each configuration visually explicit. HSR-APR is consistently the strongest configuration for repair success, test pass rate, and security warning reduction, while requiring the fewest repair iterations and the shortest average execution time.

Repair Success Rate

The Baseline LLM achieves a Repair Success Rate of 63%, corresponding to 189 successfully repaired instances out of 300. Introducing Static Analysis and RAG through Hybrid APR increases the success rate to 71%, corresponding to 213 successful repairs. The addition of Security Validation further increases the success rate to 75%, or 225 successful repairs. Finally, HSR-APR achieves an 85% Repair Success Rate, corresponding to 255 successful repairs out of 300 instances. Therefore, HSR-APR improves the Repair Success Rate by 22 percentage points compared with the Baseline LLM. In relative terms, this represents an improvement of approximately 34.9% over the baseline.

The results indicate that combining static analysis and repository-aware retrieval with iterative reflection and adaptive memory provides additional guidance that improves the ability of the LLM to generate successful repairs.

Test Pass Rate

The Baseline LLM achieves a Test Pass Rate of 72%. This increases to 81% with Hybrid APR and 84% after introducing Security Validation. HSR-APR achieves the highest Test Pass Rate at 92%. Compared with the Baseline LLM, HSR-APR provides an improvement of 20 percentage points, corresponding to approximately 27.8% relative improvement.

The results suggest that the combination of repository context, program analysis, validation feedback, and reflection contributes to generating patches that better satisfy the expected behavior of the repaired software.

Security Warning Reduction

Security performance shows the most significant difference between the evaluated configurations. The Baseline LLM achieves only 8% Security Warning Reduction, while Hybrid APR achieves 22%, when explicit Security Validation is introduced, the reduction increases substantially to 68%. The complete HSR-APR framework achieves 87% Security Warning Reduction compared with the baseline, HSR-APR improves this metric by 79 percentage points.

The results demonstrate the importance of explicitly integrating security analysis into automated program repair. More importantly, the improvement from 68% to 87% between Hybrid APR + Security Validation and HSR-APR indicates that reflection and adaptive memory provide additional value in avoiding previously observed insecure repair patterns.

Average Repair Iterations

The Baseline LLM requires an average of 5.8 repair iterations. Hybrid APR reduces this value to 4.6 iterations, while Hybrid APR + Security Validation reduces it further to 4.2 iterations. HSR-APR achieves the lowest value at only 2.9 iterations. Therefore, HSR-APR reduces the average number of repair iterations by 2.9 iterations, representing a 50% reduction compared with the baseline.

The results provide important evidence regarding the effectiveness of the Reflection and Adaptive Reflection Memory mechanisms. The system requires fewer attempts to reach an acceptable repair, suggesting that information from previous failures helps guide subsequent repair decisions.

Average Execution Time

The Baseline LLM has an average execution time of 18 minutes. Hybrid APR reduces the average time to 16 minutes, while Hybrid APR + Security Validation requires 17 minutes due to the additional security-analysis stage. Despite incorporating additional analysis and reflection mechanisms, HSR-APR achieves the lowest average execution time at 13 minutes. This represents a reduction of approximately 27.8% compared with the Baseline LLM.

The results suggest that although HSR-APR introduces additional processing stages, the reduction in repeated repair attempts can compensate for the computational overhead of analysis, security validation, and memory retrieval.

Conclusion and Future Work

This research proposed HSR-APR (Hybrid Secure Reflective Automated Program Repair), a framework designed to address important limitations of existing LLM-based Automated Program Repair systems. The proposed framework integrates Static Analysis, Fault Localization, Retrieval-Augmented Generation, LLM-based patch generation, Functional Validation, Security Validation, Reflection, and Adaptive Reflection Memory within a unified iterative repair process.

The framework was evaluated on the complete 300-instance SWE-Bench Lite test subset using four experimental configurations. The results demonstrate that the proposed HSR-APR framework achieves the strongest overall performance.

HSR-APR successfully repairs 255 out of 300 instances, corresponding to an 85% Repair Success Rate, compared with 63% for the Baseline LLM. The Test Pass Rate reaches 92%, while Security Warning Reduction reaches 87%. Furthermore, the proposed framework reduces the average number of repair iterations from 5.8 to 2.9, representing a 50% reduction, and reduces the average execution time from 18 minutes to 13 minutes.

These results support the central premise of the research: automated program repair can benefit from moving beyond single-step LLM-based patch generation toward a hybrid, security-aware, reflective, and memory-enhanced repair process.

The key contribution of HSR-APR is therefore not only the generation of patches but the integration of analysis, retrieval, validation, reflection, and adaptive knowledge reuse into a unified repair cycle.

Despite the promising results, several directions remain for future investigation.

First, the framework should be evaluated on additional APR benchmarks and larger datasets to investigate its generalizability beyond the evaluated SWE-Bench Lite instances [14], [15].

Second, future work can extend HSR-APR to additional programming languages, such as Java, JavaScript, and C/C++, to determine whether the proposed architecture is language-independent.

Third, the Adaptive Reflection Memory can be enhanced using more sophisticated representations of repair experiences. Future versions could explicitly model relationships between bugs, failed patches, failure causes, security vulnerabilities, and successful repair strategies.

Fourth, future research can investigate multi-agent HSR-APR architectures, in which specialized agents are responsible for fault localization, patch generation, security analysis, testing, and reflection.

Fifth, the Reflection Efficiency Score proposed in this research can be evaluated across multiple datasets and APR systems to determine whether it can serve as a general metric for measuring the effectiveness of reflection-based repair.

Finally, future work should investigate the trade-off between repair quality, security, memory size, and computational cost. This can help optimize HSR-APR for large-scale repository-level software repair while maintaining its ability to learn from previous repair attempts.

Acknowledgements

Special thanks are extended to the Egyptian Knowledge Bank (EKB) for providing access to essential scientific resources that greatly supported this work.

Authors contributions

E.F., M.H., A.E., and A.M. were responsible for the study's conceptualization, while the methodology was designed by E.F., M.H., and A.E. The software development and validation were handled by E.F. and M.H., and A.E., and A.M., respectively. E.F., M.H., and A.E. conducted the formal analysis, and the investigation was performed by E.F. and M.H. A.E. and A.M. managed data curation, and E.F. and M.H. wrote the original draft. The manuscript was reviewed and edited by N.M. and H.M., with visualization by E.F. and M.H. The project was supervised by A.E., and A.M. All authors have reviewed and approved the final manuscript.

Competing interests

The authors declare no competing interests.

Data Availability

The datasets analyzed in this study are publicly available. The SWE-bench Lite dataset is publicly available at: https://huggingface.co/datasets/princeton-nlp/SWE-bench_Lite. All data supporting the findings of this study are contained within these publicly available resources. No additional proprietary data was generated or analyzed.

Funding

This work received open access funding from the Science, Technology and Innovation Funding Authority (STDF) in cooperation with the Egyptian Knowledge Bank (EKB).

References

1. C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, “GenProg: A Generic Method for Automatic Software Repair,” IEEE Transactions on Software Engineering, vol. 38, no. 1, pp. 54–72, 2012, doi: 10.1109/TSE.2011.104.
2. S. Mechtaev, J. Yi, and A. Roychoudhury, “Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis,” in Proceedings of the 38th International Conference on Software Engineering (ICSE), 2016, pp. 691–701.
3. A. Silva, S. Fang, and M. Monperrus, “RepairLLaMA: Efficient Representations and Fine-Tuned Adapters for Program Repair,” arXiv:2312.15698, 2023; IEEE Transactions on Software Engineering, 2025, doi: 10.1109/TSE.2025.3581062.
4. S. B. Hossain, N. Jiang, Q. Zhou, X. Li, W.-H. Chiang, Y. Lyu, H. Nguyen, and O. Tripp, “A Deep Dive into Large Language Models for Automated Bug Localization and Repair,” arXiv:2404.11595, 2024.
5. I. Bouzenia, P. T. Devanbu, and M. Pradel, “RepairAgent: An Autonomous, LLM-Based Agent for Program Repair,” arXiv:2403.17134, 2024; in Proceedings of the 47th International Conference on Software Engineering (ICSE), 2025.
6. Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury, “AutoCodeRover: Autonomous Program Improvement,” in Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), 2024, pp. 1592–1604, doi: 10.1145/3650212.3680384.
7. J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. R. Narasimhan, and O. Press, “SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering,” arXiv:2405.15793, 2024.
8. Z. Yu, Z. Guo, Y. Wu, J. Yu, M. Xu, D. Mu, Y. Chen, and X. Xing, “PatchAgent: A Practical Program Repair Agent Mimicking Human Expertise,” in 34th USENIX Security Symposium, 2025.

9. A. Sajadi, K. Damevski, and P. Chatterjee, “How Safe Are AI-Generated Patches? A Large-scale Study on Security Risks in LLM and Agentic Automated Program Repair on SWE-bench,” arXiv:2507.02976, 2025.
10. N. Shinn, F. Cassano, E. Berman, A. Gopinath, K. Narasimhan, and S. Yao, “Reflexion: Language Agents with Verbal Reinforcement Learning,” arXiv:2303.11366, 2023.
11. P. Lewis et al., “Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks,” in Advances in Neural Information Processing Systems (NeurIPS), 2020.
12. J. Johnson, M. Douze, and H. Jégou, “Billion-scale similarity search with GPUs,” IEEE Transactions on Big Data, vol. 7, no. 3, pp. 535–547, 2019. (FAISS.)
13. N. Reimers and I. Gurevych, “Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks,” in Proceedings of EMNLP-IJCNLP, 2019, pp. 3982–3992.
14. C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. R. Narasimhan, “SWE-bench: Can Language Models Resolve Real-world GitHub Issues?” in The Twelfth International Conference on Learning Representations (ICLR), 2024.
15. C. E. Jimenez, J. Yang, and J. Geng, “SWE-bench Lite,” benchmark documentation, 2024. (Curated 300-task subset of SWE-bench.)
16. J. Gajjar, K. Subramaniakuppusamy, R. Puthal, and K. Ranaware, “SecureFixAgent: A Hybrid LLM Agent for Automated Python Static Vulnerability Repair,” arXiv:2509.16275, 2025; in ICMLA, 2025.
17. PyCQA, “Bandit: Security linter for Python,” software project and documentation, accessed 2026.
18. Semgrep, “Semgrep Code: Static Application Security Testing,” software documentation, accessed 2026.
19. Pylint Development Team, “Pylint: Static code analyser for Python,” software documentation, accessed 2026.
20. D. Kim, J. Nam, J. Song, and S. Kim, “Automatic Patch Generation Learned from Human-Written Patches,” in Proc. 35th IEEE/ACM International Conference on Software Engineering (ICSE), 2013, pp. 802–811, doi: 10.1109/ICSE.2013.6606626.
21. M. Durieux, C. Le Goues, M. Monperrus, and T. X. B. D. Nguyen, “A Critical Review on the Evolution of Automated Program Repair Systems,” Journal of Systems and Software, vol. 171, Art. no. 110817, 2021, doi: 10.1016/j.jss.2020.110817.
22. Z. Fan, X. Gao, M. Mirchev, A. Roychoudhury, and S. H. Tan, “Automated Repair of Programs from Large Language Models,” arXiv:2205.10583, 2022.
23. R. Just, D. Jalali, and M. D. Ernst, “Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs,” in Proc. International Symposium on Software Testing and Analysis (ISSTA), 2014, pp. 437–440, doi: 10.1145/2610384.2628055.
24. B. Yang, Z. Cai, F. Liu, B. Le, L. Zhang, T. F. Bissyandé, Y. Liu, and H. Tian, “A Survey of LLM-based Automated Program Repair: Taxonomies, Design Paradigms, and Applications,” arXiv:2506.23749, 2025.
25. J. Sun, F. Li, X. Qi, H. Zhang, and J. Jiang, “Empirical Evolution of Large Language Models in Automated Program Repair,” arXiv:2506.13186, 2025.
26. J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding,” in Proc. NAACL-HLT, 2019, pp. 4171–4186.
27. T. Brown et al., “Language Models are Few-Shot Learners,” in Advances in Neural Information Processing Systems (NeurIPS), vol. 33, 2020, pp. 1877–1901.
28. C. S. Le Goues, M. Pradel, and A. Roychoudhury, “Automated Program Repair: A Step Towards Practical Software Engineering,” Communications of the ACM, vol. 62, no. 12, pp. 74–83, 2019.