

ExploreRepair: Evidence-Specialized Agents for Vulnerability Repair under Information Constraints

Duc Manh Tran¹, Ratnadira Widyasari², Ivana Clairine Irsan³, Huihui Huang³,
Ting Zhang⁴, David Lo³, Rahul Gopinath¹, Hong Jin Kang¹

¹The University of Sydney, Australia, ²University of Victoria, Canada,

³Singapore Management University, Singapore, ⁴Monash University, Australia

Abstract

Automated vulnerability repair must often operate with incomplete vulnerability information: developers may lack an accurate vulnerability classification or a provided proof-of-vulnerability (PoV) test, yet still need to produce a repair. Without detailed information, a key challenge is to explore multiple plausible interpretations of the vulnerability before committing to a repair. To address this challenge, we propose EXPLOREREPAIR, a vulnerability repair framework that reconciles the vulnerability observations and repair implementations from a team of exploration agents. We first collect these observations and repair implementations from three agents, each tasked with exploring a specific dimension of the vulnerability. Then, a Lesson Aggregator and a Patch Aggregator independently synthesize repairs from these observations and concrete implementations before a final Aggregator reconciles their proposals without benchmark feedback. On PatchEval, EXPLOREREPAIR achieves repair success rates of 39.6% and 71.6% with GPT-5.6-Luna and DeepSeek-V4.1-Flash, respectively, exceeding EvoRepair, a state-of-the-art vulnerability repair approach, by 16.0 and 25.4 percentage points. On VulnLoc+, it achieves success rates of 51.4% and 62.9%, respectively, exceeding LoopRepair’s variant without PoV tests by 20.0 and 17.1 percentage points.

1 Introduction

Automated vulnerability repair (AVR) focuses on removing security flaws while preserving intended program behavior (Gao et al., 2021; Zhang et al., 2022; Zhou et al., 2024; Nong et al., 2025; Ye et al., 2026; Li et al., 2026; Han et al., 2026). Recent approaches depend on complete vulnerability information, such as detailed vulnerability descriptions or proof-of-vulnerability (PoV) tests, to guide repair. These artifacts help characterize the flaw and guide patch generation, for example by providing

feedback for validating and refining candidate repairs.

In practice, developers often have to fix vulnerabilities without access to critical vulnerability information, as it is frequently missing or unavailable (Guo et al., 2022; Sun et al., 2023). For example, only 4.1% of 75,807 CVEs had an available public PoV within one year of disclosure (Householder et al., 2020). Nevertheless, state-of-the-art AVR approaches, such as EvoRepair and ContraFix, depend on information only available from a complete vulnerability analysis, including PoVs and oracle feedback, to guide repair (Hu et al., 2026; Liu et al., 2026).

In this study, we investigate vulnerability repair without complete information. This setting reflects real-world practice, in which such information is frequently missing, especially for newly discovered vulnerabilities whose fixes are urgently required. To compensate for the loss of complete vulnerability information, a repair method has to explore the codebase to **gather evidence while avoiding premature commitment to an incorrect repair hypothesis**.

The need to **simultaneously consider heterogeneous evidence types** makes this task difficult. First, the vulnerable code location provides incomplete *structural evidence*, e.g., information about how inputs can reach the code, and parallel repair sites may lie elsewhere in the repository (Tang et al., 2025). Second, source code provides incomplete *semantic evidence*, e.g., the relevant API contract and intended behavior must be reconstructed. Third, the absence of a ground-truth PoV removes *executable evidence* that demonstrates the vulnerable behavior.

To this end, we propose EXPLOREREPAIR, a vulnerability repair framework with two phases, exploration and aggregation. During *exploration*, three independent, **evidence-specialized agents** investigate repository structure, infer developer in-

tent, or construct and execute a PoV. Each agent returns a candidate patch and the trajectory of its exploration.

During *aggregation*, the agents’ trajectories and patches are **hierarchically aggregated** to leverage their complementary information and avoid committing to a single dimension of analysis. First, we use an LLM to distill the agents’ trajectories and candidate patches into lessons that capture their vulnerability observations and supporting evidence. Next, the Lesson Aggregator and Patch Aggregator independently synthesize repairs from the distilled lessons and candidate patches, respectively. This allows the two aggregators to consider the agents’ vulnerability diagnoses and concrete repair implementations separately. Finally, the Aggregator reconciles their proposals, combining complementary changes and resolving conflicting assumptions to produce the final repair.

We evaluate EXPLOREREPAIR on 169 Python, JavaScript, and Go vulnerabilities from PatchEval (Wei et al., 2025) and 35 C/C++ vulnerabilities from VulnLoc+ (Ye et al., 2026). On PatchEval, EXPLOREREPAIR achieves repair success rates of 39.6% and 71.6% with GPT-5.6-Luna and DeepSeek-V4.1-Flash, respectively, exceeding a single-agent baseline using the same model by 7.1 and 14.2 percentage points. Our framework also outperforms EvoRepair, a state-of-the-art vulnerability repair approach, by 16.0 and 25.4 percentage points, respectively. These results demonstrate that EXPLOREREPAIR’s improvements are unlikely to be attributable to additional tools or instructions. On VulnLoc+, EXPLOREREPAIR achieves repair success rates of 51.4% and 62.9%, exceeding LoopRepair by 20.0 and 17.1 percentage points. Even when LoopRepair is given access to the ground-truth PoV, EXPLOREREPAIR outperforms it by 2.9 and 11.4 percentage points, respectively.

In short, this paper makes three contributions:

- We formulate information-constrained vulnerability repair as a task of repairing the vulnerability using repository context without relying on vulnerability metadata or oracle feedback.
- We propose EXPLOREREPAIR, a new automated vulnerability repair approach that employs evidence-specialized exploration with hierarchical lesson–patch aggregation to guide repair generation.
- We evaluate EXPLOREREPAIR on two benchmarks comprising 204 vulnerabilities in total, showing that it substantially outperforms state-of-the-art vulnerability repair methods.

2 Related Work

2.1 Automated Vulnerability Repair

Recent program analysis-based approaches generate repairs from executions that trigger the vulnerability. They often require language-specific analyses or a demonstration of the security flaw (Hong et al., 2020; Gao et al., 2021; Zhang et al., 2022; Shariffdeen et al., 2025). For example, LoopRepair requires a PoV test and uses information from its execution to refine C/C++ patches (Ye et al., 2026).

Training-based approaches learn from historical fixes using CWE knowledge, security patterns, or compiler feedback (Zhou et al., 2024; Li et al., 2026; Yang et al., 2026; Zhang et al., 2026). Recent LLM-based repair systems use repository context (Nong et al., 2025; Heo et al., 2026; Rashid et al., 2026) to understand vulnerabilities. For example, IntentFix analyzes the codebase and reconstructs developer intent when synthesizing a repair. EvoRepair (Hu et al., 2026) retrieves experiences from earlier repairs, guided by ground-truth PoV tests. ContraFix (Liu et al., 2026) mutates the PoV to contrast runtime behavior. KeaRepair (Cao et al., 2026) combines verified program facts with knowledge from historical fixes. AgenticRepair (Fu et al., 2026) uses a multi-agent approach to examine different context sources. Similar to EXPLOREREPAIR, these methods coordinate agents for codebase exploration; however, they do not prevent premature commitment to an incorrect repair.

2.2 Multi-Agent Reasoning and Trajectory Distillation

Multi-agent systems assign software tasks to specialized roles, as in MetaGPT and ChatDev (Hong et al., 2024; Qian et al., 2024; Islam et al., 2024). Related systems coordinate agents for code generation, review, vulnerability detection, and general program repair (Tang et al., 2024; Luo et al., 2026; Widiasari et al., 2026; Chen, 2025; Lee et al., 2025). While most organize agents around discussion of a shared task, EXPLOREREPAIR assigns each agent to explore a different dimension of the codebase independently.

While prior work used multi-agent approaches, such approaches do not automatically improve di-

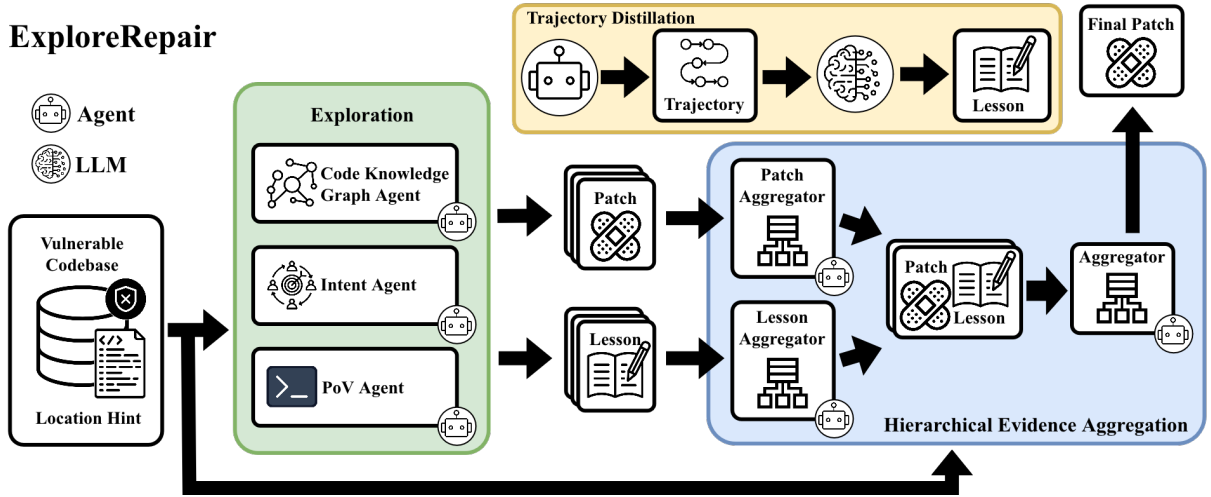


Figure 1: EXPLOREREPAIR has two phases. (1) In its exploration phase, three exploration agents independently produce trajectories and patches. (2) In its aggregation phase, the Lesson Aggregator and Patch Aggregator synthesize repairs from the agents’ observations and repair implementations, respectively. The final Aggregator reconciles these two repairs. Trajectory distillation uses an LLM to summarize exploration trajectories and candidate patches into role-specific lessons, and aggregation trajectories into lessons for the final Aggregator.

versity or correctness (Encheng et al., 2026). In particular, incorrect initial analysis can mislead other agents (Oh et al., 2026; Yoffe et al., 2025; Zhu et al., 2026). To avoid this, EXPLOREREPAIR uses independent agents for exploration, with each agent specializing in a single evidence type.

An agent’s trajectory is a sequence of thoughts, actions, and environment observations generated while solving a task (Yao et al., 2023). Prior work summarizes information from trajectories into feedback, workflows, or reusable strategies (Shinn et al., 2023; Wang et al., 2025; Yang et al., 2025). ReasoningBank distills lessons from successful and failed trajectories for later retrieval (Ouyang et al., 2026). Because each trajectory is lengthy and excessively large contexts can lead to confused reasoning (Liu et al., 2024), EXPLOREREPAIR distills each agent’s exploration into lessons during aggregation, before synthesizing the repair.

3 Problem Setting

Let C denote the repository at its vulnerable revision and let ℓ denote a function-level localization hint containing one or more file and line ranges. Prior AVR methods may also receive additional vulnerability information, such as CWE labels or developer-written PoV tests, as well as feedback from evaluating candidate patches against a benchmark oracle. We denote these artifacts by F .

In an **information-constrained vulnerability repair** setting, the repair agent $\mathcal{A}$ receives only (C, ℓ) . It can inspect the repository’s code and can execute its test suites, but has no access to F . It produces a single patch p :

$$p = \mathcal{A}(C, \ell).$$

An ideal patch should eliminate the vulnerability, as determined by the benchmark’s PoV test. At the same time, it should preserve existing functionality, as assessed by the repository’s unit tests (Yu et al., 2026; Rashid et al., 2026; Sun et al., 2026).

4 EXPLOREREPAIR

4.1 Overview

Figure 1 outlines EXPLOREREPAIR, which is a two-phase approach. First, in its exploration phase, EXPLOREREPAIR invokes three exploration agents that collect evidence along different dimensions of analysis. Second, in the aggregation phase, the trajectories and patches produced by each exploration agent are distilled into lessons before they are aggregated into a single final patch, produced as the output of EXPLOREREPAIR.

In the exploration phase, each of the three exploration agents is individually tasked with producing a repair for the vulnerability. To do so, each agent must collect and consider evidence along its assigned dimension of the codebase that may be

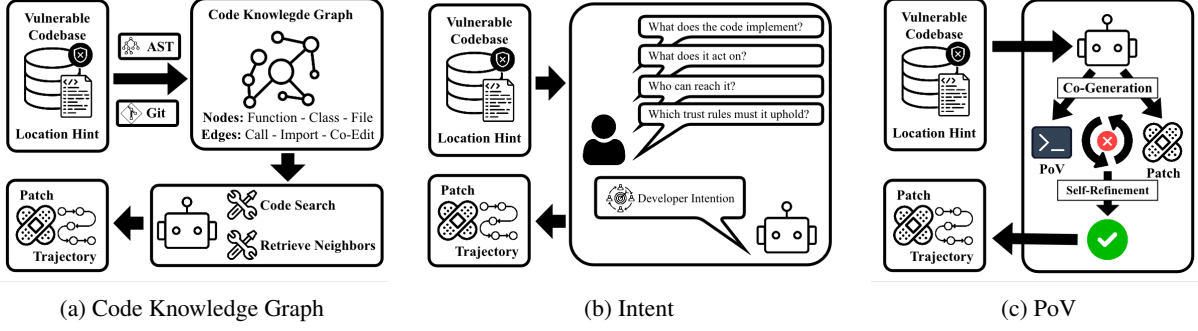


Figure 2: The three exploration agents receive the same repository and localization hint. They investigate (a) repository structure, (b) intended behavior and trust constraints, and (c) failures that demonstrate the vulnerability.

relevant to the repair. One agent collects structural evidence captured in a Code Knowledge Graph. The graph represents files, functions, and classes and their call and import relationships, extracted from abstract syntax trees (ASTs). We also use Git history to link files changed in the same commit. The Intent agent collects semantic evidence by focusing on reconstructing developer intent. Finally, the PoV agent focuses on constructing executable evidence of the vulnerability. Each exploration agent produces a pair of a trajectory and a candidate patch. These outputs are used in the aggregation phase.

In the aggregation phase, to leverage the diverse information produced by the different evidence-specialized agents, the trajectories and patches are hierarchically aggregated. First, we use an LLM to distill each agent’s trajectory and candidate patch into a role-specific lesson covering code structure, developer intent, or observed execution, together with the proposed change and remaining uncertainties. Next, in parallel, the Lesson Aggregator synthesizes a repair from the lessons, while the Patch Aggregator synthesizes one from the exploration agents’ patches. Each of these aggregators captures a complementary aspect of the repair. The former captures the agents’ diagnoses and inferred security constraints, while the latter preserves concrete implementation details that may be lost during distillation. The final Aggregator then reconciles the two proposals by examining their supporting evidence and implementation choices, combining compatible changes or resolving conflicting assumptions to produce the final repair.

4.2 Evidence-Specialized Exploration

4.2.1 Structural Evidence

The Code Knowledge Graph agent investigates the code related to the localized function (Figure 2a). For each vulnerability, we construct a graph of functions, classes, and files from abstract syntax trees. Call edges connect callers and callees, import edges connect dependent files, and shared-commit edges connect files that developers changed together. We also embed the node text in a vector store for that vulnerability.

The agent uses two tools to query the graph. `search_code` retrieves nodes related to a natural-language or code query, and `graph_neighbors` returns calls and imports connected to a selected node. The agent traces how external inputs reach the localized function, examines helpers and validators, and compares related paths that perform similar operations. It uses these findings to diagnose the vulnerability and produce a patch, returning its trajectory alongside the patch.

4.2.2 Semantic Evidence

The Intent agent reconstructs developer intent by identifying the expected behavior and trust assumptions before attempting a repair (Figure 2b). It addresses four questions adapted from IntentFix (Heo et al., 2026): what specification or API contract governs the code, who can reach it and control its inputs, what resources it protects, and what constraints should hold for untrusted data. From these observations, the agent infers up to five security constraints that should be enforced.

The agent answers these questions in order and supports its claims with file and line references. It records uncertainty when the repository does not provide enough information. It then identifies where the implementation violates the inferred

constraints and produces a patch, returning the trajectory that explains its diagnosis.

4.2.3 Executable Evidence

The PoV agent constructs a test to demonstrate the vulnerability it has inferred (Figure 2c). It first traces attacker-controlled input from a public API, command-line interface, or endpoint to the localized code. It then writes and executes a PoV test against the vulnerable repository, revising the test until the observed behavior demonstrates the suspected security violation. Finally, it repairs the root cause and reruns the same test to check the repair.

The agent derives this test from the repository and its own diagnosis. It cannot access the developer-written PoV used for benchmark evaluation. The prompt prohibits weakening the generated test or adding a repair specific to its input, and asks the agent to address the underlying flaw in production code. The agent returns its patch and the execution observations that support it.

4.3 Hierarchical Evidence Aggregation

The aggregation phase requires EXPLORE REPAIR to aggregate its findings while leveraging the complementary information from the trajectories and proposed patches. Each agent provides a trajectory and a proposed patch. Its trajectory provides evidence and justification for its diagnosis, while its proposed patch provides implementation details, such as the code locations where new code should be added.

The Lesson Aggregator and Patch Aggregator first synthesize repairs separately, allowing each to examine the information in its inputs before a final Aggregator reconciles their proposals. This process has three steps.

1) Distilling the agents’ observations. We use a separate LLM call to distill each agent’s trajectory and candidate patch into a short lesson. The lesson records the suspected vulnerability, supporting code locations or execution observations, proposed change, and unresolved claims. The LLM accounts for the agent’s role: structural lessons focus on code paths and repair locations, semantic lessons focus on expected behavior and trust constraints, and executable lessons focus on the observed violation and the effect of the patch.

2) Aggregating observations and implementations separately. The Lesson Aggregator receives the three lessons and synthesizes a repair

from the agents’ observations. In parallel, the Patch Aggregator receives the three candidate diffs and synthesizes a repair from their concrete implementations. Both can inspect the original repository and are instructed to check the supplied attempts, since any of them may be incorrect. The lessons provide the reasoning and uncertainty behind a repair, while the patches retain implementation details that the lessons may omit.

3) Reconciling the repair proposals. After the Lesson Aggregator and Patch Aggregator propose their repairs, we use an LLM to distill each aggregator’s trajectory into a lesson. The final Aggregator receives these two lessons and their corresponding patches, checks their claims against the repository, and produces the final patch. It can adopt one repair, combine compatible changes, reject assumptions shared by both attempts, or construct a new patch. Its decision uses the supplied observations, implementation details, and repository code.

5 Experimental Setup

5.1 Benchmark

PatchEval provides runnable test artifacts and pre-built Docker images for 230 CVEs across Python, JavaScript, and Go (Wei et al., 2025). Of the 176 instances with both a PoV and a unit-test suite, we exclude seven whose benchmark test patches do not compile against the vulnerable repository. We evaluate on the remaining 169 vulnerabilities: 64 in Python, 49 in JavaScript, and 56 in Go. Table 1 summarizes their vulnerability types.

To evaluate transfer to C/C++, we use VulnLoc+, the benchmark introduced with LoopRepair (Ye et al., 2026). We use 35 of the 40 vulnerabilities included in the benchmark, excluding vulnerabilities whose PoV tests already pass on the vulnerable version before a correct patch is applied. VulnLoc+ provides PoVs but no unit tests. Table 2 summarizes the vulnerabilities in the benchmark.

5.2 Metrics

On PatchEval, a patch is considered **plausible** if it passes the benchmark PoV test and **successful** if it also passes the repository’s unit tests. Let N denote the number of evaluated CVEs, S the number of successful repairs, and P the number of plausible patches. We report **Repair Success Rate (RSR)** and **Overfitting Rate (OFR)** as percentages:

$$\text{RSR} = \frac{S}{N} \times 100, \quad \text{OFR} = \frac{P - S}{N} \times 100.$$

Table 1: PatchEval CVEs by language and CWE. A CVE may have multiple CWEs; *Other* groups the 27 types with fewer than 10 instances.

Vulnerability Type	CWE	Py	JS	Go	Total
Path traversal	CWE-22	14	4	9	27
Code injection	CWE-94	11	9	6	26
Ext. file path control	CWE-73	14	4	7	25
Command injection	CWE-77	11	7	6	24
OS command injection	CWE-78	11	6	6	23
Open redirect	CWE-601	6	6	4	16
Unnecessary privileges	CWE-250	6	2	4	12
Improper privilege mgmt.	CWE-269	6	2	4	12
Improper authorization	CWE-285	6	2	4	12
SQL injection	CWE-89	4	3	4	11
Missing authorization	CWE-862	-	4	6	10
Other (27 CWEs)		24	25	29	78
CVEs		64	49	56	169

Table 2: VulnLoc+ (Ye et al., 2026) vulnerabilities by type. The benchmark spans 9 C projects.

Type	Projects	Total
Buffer overflow	9	23
Integer overflow	3	3
Divide by zero	2	3
Null-pointer dereference	2	2
Data-type overflow	1	2
Use after free	1	2
Vulnerabilities	9	35

OFR measures the proportion of all evaluated CVEs whose patches pass the PoV but fail the repository’s unit tests. As VulnLoc+ does not provide unit tests, we measure RSR using the benchmark PoV alone and do not report OFR for this benchmark.

5.3 Baselines

We compare EXPLOREREPAIR with IntentFix, which infers developer intent (Heo et al., 2026), and EvoRepair, which retrieves experiences from prior trajectories (Hu et al., 2026). We adapt EvoRepair to exclude ground-truth PoVs and benchmark evaluation feedback. On VulnLoc+, we evaluate LoopRepair, a C/C++ repair approach (Ye et al., 2026), in its released PoV-assisted setting and a no-PoV variant. We also include a standalone LLM, Single Agent, and Multi-Agent. Baseline configurations are provided in Appendix D.

5.4 Exploration and Aggregation Configurations

We evaluate the 3-in-1 Agent to compare independent exploration against combining all three exploration mechanisms within a single agent. We also

compare four aggregation configurations using the same exploration agents’ outputs.

3-in-1 Agent. The *3-in-1 Agent* extends the Single Agent baseline with Code Knowledge Graph tools, Intent guidance, and PoV-generation guidance. It uses all three exploration mechanisms within a single trajectory to investigate the vulnerability and produce a repair. This allows us to compare joint exploration within one agent against independent explorations followed by hierarchical aggregation.

Patch and Lesson Aggregators. We also compare four aggregation configurations using the same exploration agents’ outputs. The *Patch Aggregator* receives only the exploration agents’ candidate patches. The *Lesson Aggregator* receives only the three lessons distilled from their observations. *Direct Lessons + Patches* gives one agent the three lessons and candidate patches together. In the complete system, the Lesson Aggregator and Patch Aggregator independently produce repairs. The final Aggregator receives these repairs and their distilled lessons, then reconciles their proposals.

5.5 General-Purpose Coding Agent

We also compare EXPLOREREPAIR with Codex, OpenAI’s general-purpose coding agent (OpenAI, 2026), to assess whether specialized exploration and aggregation improve vulnerability repair beyond a general-purpose coding workflow. Codex uses its native agent loop and receives the same repository and localization hint, with benchmark artifacts removed and external capabilities disabled. We run Codex with GPT-5.6-Luna and DeepSeek-V4.1-Flash.

5.6 Implementation and Hardware

Our experiments use OpenAI GPT-5.6-Luna and DeepSeek-V4.1-Flash through OpenRouter. We set a time limit for each phase to 1800 s per vulnerability. The Code Knowledge Graph agent uses bge-large-en-v1.5 to embed code and retrieves $k = 5$ locations. We run the agents in isolated, network-disabled Docker containers with the repository at its vulnerable commit. To prevent data leakage, we remove benchmark PoVs, reference patches, and grading scripts from the containers. Repository tests remain available. Throughout the experiments, the agents do not have access to the benchmark evaluation code or results. The experiments were run on a machine running Ubuntu

Table 3: Repair outcomes on 169 PatchEval vulnerabilities. Each entry reports Repair Success Rate (RSR) followed by Overfitting Rate (OFR) in parentheses.

Approach	RSR $\uparrow$ (OFR $\downarrow$)	
	GPT-5.6-Luna	DeepSeek-V4.1-Flash
IntentFix	4.7% (1.2%)	9.5% (1.2%)
EvoRepair	23.7% (3.0%)	46.2% (1.8%)
LLM	10.7% (1.2%)	23.7% (2.4%)
Multi-Agent	30.2% (1.2%)	55.6% (1.2%)
Single Agent	32.5% (0.6%)	57.4% (0.6%)
Codex	34.3% (0.6%)	43.8% (0.6%)
Structural (Graph)	36.1% (1.2%)	55.6% (0.0%)
Semantic (Intent)	29.6% (0.0%)	59.8% (0.0%)
Executable (PoV)	33.7% (0.0%)	58.0% (0.6%)
3-in-1 Agent	30.8% (1.2%)	53.3% (0.6%)
EXPLOREREPAIR	39.6% (0.0%)	71.6% (0.6%)

22.04.5, with an AMD EPYC 7763 processor and 251 GB of RAM.

6 Results

6.1 Overall Repair Effectiveness

We investigate the overall effectiveness of EXPLOREREPAIR. Table 3 shows the experimental results of the baselines, individual exploration agents, and EXPLOREREPAIR on PatchEval.

EXPLOREREPAIR outperforms EvoRepair under the same information constraints. EvoRepair retrieves experiences from prior trajectories to guide successive repair attempts (Hu et al., 2026). With the adaptation described in Section 5.3, it achieves an RSR of 23.7% using GPT-5.6-Luna. EXPLOREREPAIR achieves 39.6%, a gain of 16.0 percentage points (Table 3). EvoRepair’s OFR is 3.0%, compared with 0.0% for EXPLOREREPAIR. With DeepSeek-V4.1-Flash, EvoRepair achieves an RSR of 46.2%, compared with 71.6% for EXPLOREREPAIR, a gain of 25.4 percentage points. Its OFR is 1.8%, compared with 0.6% for EXPLOREREPAIR.

EXPLOREREPAIR achieves the highest repair success rate with both models. With GPT-5.6-Luna, EXPLOREREPAIR achieves an RSR of 39.6%, a gain of 7.1 percentage points compared with 32.5% for Single Agent. With DeepSeek-V4.1-Flash, its RSR is 71.6%, compared with 57.4% for Single Agent, a gain of 14.2 percentage points. IntentFix achieves RSRs of 4.7% and 9.5%, yielding gains of 34.9 and 62.1 percentage points

for EXPLOREREPAIR, respectively. EXPLOREREPAIR’s OFR is 0.0% with GPT-5.6-Luna and 0.6% with DeepSeek-V4.1-Flash, compared with 0.6% for Single Agent with either model.

Multi-Agent achieves RSRs of 30.2% with GPT-5.6-Luna and 55.6% with DeepSeek-V4.1-Flash, below Single Agent and EXPLOREREPAIR with both models. The improvements of EXPLOREREPAIR over the multi-agent baseline suggest that generic subagent delegation does not achieve the benefits of evidence-specialized explorations and hierarchical aggregation. These results show that the effectiveness of EXPLOREREPAIR goes beyond a single model.

The strongest exploration agent differs between models. The Code Knowledge Graph agent performs best with GPT-5.6-Luna, achieving an RSR of 36.1%, whereas the Intent agent performs best with DeepSeek-V4.1-Flash, achieving 59.8%. Neither agent performs best with both models. Full results are available in Appendix A.

Aggregating independent explorations outperforms the 3-in-1 Agent. The 3-in-1 Agent, which has access to all three exploration mechanisms, achieves RSRs of 30.8% with GPT-5.6-Luna and 53.3% with DeepSeek-V4.1-Flash. EXPLOREREPAIR improves on these rates by 8.9 and 18.3 percentage points, respectively. This suggests that agents struggle to correctly reason over the different evidence types. The results show the effectiveness of aggregating independent explorations of each dimension.

6.2 Aggregation Analysis

To understand the contribution of the aggregation phase to the effectiveness of EXPLOREREPAIR, we conduct a detailed ablation analysis. In these experiments, we compare the Lesson Aggregator, Patch Aggregator, Direct Lessons + Patches, and the complete EXPLOREREPAIR using the same exploration agents’ outputs.

Reconciling the two aggregators’ repairs improves success rates. EXPLOREREPAIR achieves RSRs of 39.6% and 71.6%, exceeding both separate aggregators and Direct Lessons + Patches. Compared with the stronger separate aggregator, the final Aggregator repairs one additional vulnerability with GPT-5.6-Luna and four with DeepSeek-V4.1-Flash, corresponding to gains of 0.6 and 2.4 percentage points, respectively.

Table 4: Effectiveness of different aggregation methods on PatchEval. Each entry reports Repair Success Rate (RSR) followed by Overfitting Rate (OFR) in parentheses.

Configuration	RSR $\uparrow$ (OFR $\downarrow$)	
	GPT-5.6-Luna	DeepSeek-V4.1-Flash
Lesson Aggregator	39.1% (0.6%)	63.9% (0.6%)
Patch Aggregator	37.3% (0.6%)	69.2% (1.2%)
Direct Lessons + Patches	37.3% (0.6%)	67.5% (0.6%)
EXPLOREPAIR	39.6% (0.0%)	71.6% (0.6%)

Aggregation enables successful repairs even when all three exploration agents fail. On PatchEval, EXPLOREPAIR successfully repairs 11 vulnerabilities with DeepSeek-V4.1-Flash and 3 vulnerabilities with GPT-5.6-Luna that none of the three exploration agents could repair individually. These results demonstrate that aggregation can produce successful repairs by combining information from individually unsuccessful explorations.

The stronger aggregator differs between models. While EXPLOREPAIR outperforms the baselines with both models, there are differences in the individual performance of each aggregator. The Lesson Aggregator outperforms the Patch Aggregator with GPT-5.6-Luna, achieving an RSR of 39.1% versus 37.3%, while the Patch Aggregator performs better with DeepSeek-V4.1-Flash, achieving 69.2% versus 63.9%. Direct Lessons + Patches achieves RSRs of 37.3% and 67.5%, respectively. With either model, giving the lessons and patches to one agent does not outperform the stronger of the two separate aggregators.

6.3 C/C++ Programming Language Transfer

Next, we investigate whether EXPLOREPAIR’s effectiveness transfers to other programming languages. Table 5 compares EXPLOREPAIR with LoopRepair on VulnLoc+. We run LoopRepair both with its required PoV and without it.

The framework transfers to C/C++. Without access to a provided PoV, EXPLOREPAIR achieves RSRs of 51.4% with GPT-5.6-Luna and 62.9% with DeepSeek-V4.1-Flash, compared with 31.4% and 45.7% for LoopRepair without a PoV, gains of 20.0 and 17.1 percentage points, respectively. Even when LoopRepair receives the ground-truth PoV, EXPLOREPAIR achieves higher RSRs, exceeding it by 2.9 and 11.4 percentage points, re-

Table 5: Repair Success Rate (RSR) on 35 VulnLoc+ C/C++ vulnerabilities. OFR cannot be measured because this benchmark has no unit tests. LoopRepair + PoV receives the ground-truth PoV while the other configurations do not.

Approach	GPT-5.6-Luna	DeepSeek-V4.1-Flash
	RSR $\uparrow$	
LoopRepair + PoV	48.6%	51.4%
LoopRepair (no PoV)	31.4%	45.7%
Single Agent	48.6%	57.1%
EXPLOREPAIR	51.4%	62.9%

spectively.

6.4 Comparison with a General-Purpose Coding Agent

Next, we compare EXPLOREPAIR with Codex (OpenAI, 2026) using GPT-5.6-Luna and DeepSeek-V4.1-Flash at medium reasoning effort. Both receive the same localization hint and repository, with external capabilities disabled. Codex uses its native agent loop, and EXPLOREPAIR uses ReAct agents with the exploration and aggregation phases. As Table 3 shows, EXPLOREPAIR achieves an RSR of 39.6%, compared with 34.3% for Codex, a gain of 5.3 percentage points. Their OFRs are 0.0% and 0.6%, respectively. With DeepSeek-V4.1-Flash, EXPLOREPAIR achieves an RSR of 71.6%, compared with 43.8% for Codex, a gain of 27.8 percentage points. Both have an OFR of 0.6%. Under the same model and vulnerability information, EXPLOREPAIR’s exploration and aggregation also outperform a general-purpose coding agent.

7 Conclusion

We propose EXPLOREPAIR, which collects vulnerability observations and repair implementations from three exploration agents. A Lesson Aggregator and a Patch Aggregator independently synthesize repairs from these outputs, and a final Aggregator reconciles their proposals without benchmark feedback. Experiments on PatchEval and VulnLoc+ show that EXPLOREPAIR outperforms the evaluated baselines, including IntentFix, LoopRepair, other LLM-based approaches, and Codex. We plan to expand the evaluation of EXPLOREPAIR to a more diverse range of programming languages in future work.

Limitations

First, EXPLOREREPAIR’s results depend on the underlying LLM. While EXPLOREREPAIR outperforms the baselines for the two models used in our experiments, the strongest exploration agent differs between them. As such, the experimental results are likely to vary with other models, prompts, or sampling seeds. We ran the experiments three times at a sampling temperature of 1, with repair counts varying by at most one across runs.

Second, some baselines were designed to use vulnerability reports or PoVs. Removing these artifacts transforms the setting for which the baseline approaches were designed. We preserved their original code where possible, and thoroughly assessed the implementation details. During this process, we identified an evaluation flaw in IntentFix’s replication package, which has been confirmed by its authors.

Ethical Considerations

Automated repair can help maintainers respond to security flaws, but an incorrect patch may give developers unwarranted confidence in the repair. Automated repair methods can also reveal exploitable behavior that can be used by malicious adversaries. However, advances in offensive capabilities also increase the need to improve automated defensive tools.

Our experiments use vulnerabilities from public benchmarks within isolated, network-disabled containers. Agents used in our experiments were prevented from communicating over the network and interacting with other deployed systems. The replication package contains repair infrastructure and public benchmark outputs, but excludes credentials and previously undisclosed vulnerabilities.

References

- Sicong Cao, Hao Ma, Le Yu, Kangyi Ding, Xiaolei Liu, Terry Yue Zhuo, Bo Wang, Xingwei Lin, Xiaobing Sun, Linzhang Wang, et al. 2026. Knowledge-enhanced agentic vulnerability repair. *arXiv preprint arXiv:2607.00820*.
- Yujia Chen. 2025. Autoreview: An llm-based multi-agent system for security issue-oriented code review. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, pages 1022–1024.
- Cui Encheng, Shaowen Peng, Kazuhiro Ito, XU Jinsha, Hisada Shohei, Shoko Wakamiya, and Eiji Aramaki. 2026. Single-agent generation surpasses multi-agent systems in semantic diversity. In *Findings of the Association for Computational Linguistics: ACL 2026*, pages 37993–38008.
- Michael Fu, Qiyue Mei, Patanamon Thongtanunam, and Kla Tantithamthavorn. 2026. Agenticrepair: Multifaceted program context engineering for agentic vulnerability repair. *arXiv preprint arXiv:2607.29422*.
- Xiang Gao, Bo Wang, Gregory J Duck, Ruyi Ji, Yingfei Xiong, and Abhik Roychoudhury. 2021. Beyond tests: Program vulnerability repair via crash constraint extraction. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(2):1–27.
- Hao Guo, Sen Chen, Zhenchang Xing, Xiaohong Li, Yude Bai, and Jiamou Sun. 2022. Detecting and augmenting missing key aspects in vulnerability descriptions. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(3):1–27.
- Seunghoon Han, YoungJae Kim, Yeseung Lee, and Jooyong Yi. 2026. Symradar: Poc-centered bounded verification for vulnerability repair. In *Proceedings of the 2026 IEEE/ACM 48th International Conference on Software Engineering*, pages 3761–3773.
- Jinseok Heo, Dongwook Choi, Jinyoung Kim, Misoo Kim, and Eunseok Lee. 2026. Intentfix: Automated logic vulnerability repair via llm-driven intent modeling. In *Proceedings of the 2026 IEEE/ACM 48th International Conference on Software Engineering*, pages 843–854.
- Seongjoon Hong, Junhee Lee, Jeongsoo Lee, and Hakjoo Oh. 2020. Saver: scalable, precise, and safe memory-error repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 271–283.
- Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiaowu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Steven Yau, Zijuan Lin, Liyang Zhou, et al. 2024. Metagpt: Meta programming for a multi-agent collaborative framework. In *International Conference on Learning Representations*, volume 2024, pages 23247–23275.
- Allen D Householder, Jeff Chrabaszcz, Trent Novelly, David Warren, and Jonathan M Spring. 2020. Historical analysis of exploit availability timelines. In *13th USENIX Workshop on Cyber Security Experimentation and Test (CSET 20)*.
- Haichuan Hu, Guoqing Xie, Qianjun Zhang, Jiawei Liu, Shengcheng Yu, Chunrong Fang, Zhenyu Chen, and Liang Xiao. 2026. Evorepair: Enhancing vulnerability repair agents through experience-based self-evolution. *arXiv preprint arXiv:2605.30105*.
- Md Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. 2024. Mapcoder: Multi-agent code generation for competitive problem solving. In

- Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 4912–4944.
- Cheryl Lee, Chunqiu Steven Xia, Longji Yang, Jentse Huang, Zhouruixing Zhu, Lingming Zhang, and Michael R Lyu. 2025. Unidebugger: Hierarchical multi-agent framework for unified software debugging. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 18248–18277.
- Jia Li, Zhuangbin Chen, Yuxin Su, and Michael R Lyu. 2026. Vulkey: Automated vulnerability repair guided by domain-specific repair patterns. *Proceedings of the ACM on Software Engineering*, 3(FSE):2465–2487.
- Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the middle: How language models use long contexts. *Transactions of the association for computational linguistics*, 12:157–173.
- Simiao Liu, Fang Liu, Peiding Wang, Taichuan Li, Yinghao Zhu, Xiaoli Lian, and Li Zhang. 2026. Contrafix: Skill-enhanced contrastive runtime analysis for vulnerability repair. *arXiv preprint arXiv:2605.17450*.
- Wenqiang Luo, Jacky Wai Keung, Boyang Yang, Jacques Klein, Tegawende F Bissyande, Haoye Tian, and Bach Le. 2026. Unlocking llm repair capabilities through cross-language translation and multi-agent refinement. In *Proceedings of the 2026 IEEE/ACM 48th International Conference on Software Engineering*, pages 830–842.
- Yu Nong, Haoran Yang, Long Cheng, Hongxin Hu, and Haipeng Cai. 2025. {APPATCH}: Automated adaptive prompting large language models for {Real-World} software vulnerability patching. In *34th USENIX Security Symposium (USENIX Security 25)*, pages 4481–4500.
- Jihwan Oh, Minchan Jeong, Jongwoo Ko, and Se-Young Yun. 2026. From belief entrenchment to robust reasoning in llm agents. *Transactions of the Association for Computational Linguistics*, 14:1286–1307.
- OpenAI. 2026. [Codex](#). Accessed: 2026-10-13.
- Siru Ouyang, Jun Yan, I Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long Le, Samira Daruki, Xiangru Tang, et al. 2026. Reasoningbank: Scaling agent self-evolving with reasoning memory. In *International Conference on Learning Representations*, volume 2026, pages 94327–94354.
- Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, et al. 2024. Chatdev: Communicative agents for software development. In *Proceedings of the 62nd annual meeting of the association for computational linguistics (volume 1: Long papers)*, pages 15174–15186.
- Syed Md Mukit Rashid, Abdullah Al Ishtiaq, Kai Tu, Yilu Dong, Tianwei Wu, Ali Ranjbar, Tianchang Yang, Najrin Sultana, Shagufta Mehnaz, and Syed Rafiul Hussain. 2026. Logiceval: A systematic framework for evaluating automated repair techniques for logical vulnerabilities in real-world software. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 46025–46049.
- Ridwan Shariffdeen, Christopher S Timperley, Yannic Noller, Claire Le Goues, and Abhik Roychoudhury. 2025. Vulnerability repair via concolic execution and code mutations. *ACM Transactions on Software Engineering and Methodology*, 34(4):1–27.
- Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652.
- Jiamou Sun, Zhenchang Xing, Xin Xia, Qinghua Lu, Xiwei Xu, and Liming Zhu. 2023. Aspect-level information discrepancies across heterogeneous vulnerability reports: Severity, types and detection methods. *ACM Transactions on Software Engineering and Methodology*, 33(2):1–38.
- Yuxuan Sun, Yuze Zhao, Yufeng Wang, Yao Du, Zhiyuan Ma, Jinbo Wang, Mengdi Zhang, Kai Zhang, and Zhenya Huang. 2026. Swe-mutation: Can llms generate reliable test suites in software engineering? In *Findings of the Association for Computational Linguistics: ACL 2026*, pages 39651–39674.
- Xunzhu Tang, Jiechao Gao, Jin Xu, Tiezhu Sun, Yewei Song, Saad Ezzini, Wendkūni C. Ouedraogo, Jacques Klein, and Tegawendé F. Bissyandé. 2025. [SynFix: Dependency-aware program repair via RelationGraph analysis](#). In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 4878–4894, Vienna, Austria. Association for Computational Linguistics.
- Xunzhu Tang, Kisub Kim, Yewei Song, Cedric Lothritz, Bei Li, Saad Ezzini, Haoye Tian, Jacques Klein, and Tegawendé F Bissyandé. 2024. Codeagent: Autonomous communicative agents for code review. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 11279–11313.
- Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. 2025. [Agent workflow memory](#). In *Forty-second International Conference on Machine Learning*.
- Zichao Wei, Jun Zeng, Ming Wen, Zeliang Yu, Kai Cheng, Yiding Zhu, Jingyi Guo, Shiqi Zhou, Le Yin, Xiaodong Su, et al. 2025. Patcheval: A new benchmark for evaluating llms on patching real-world vulnerabilities. *arXiv preprint arXiv:2511.11019*.
- Ratnadira Widyasari, Martin Weyssow, Ivana Clairine Irsan, Han Wei Ang, Frank Liauw, Eng Lieh Ouh,

- Lwin Khin Shar, Hong Jin Kang, and David Lo. 2026. Let the trial begin: A mock-court approach to vulnerability detection using llm-based agents. In *Proceedings of the 2026 IEEE/ACM 48th International Conference on Software Engineering*, pages 1030–1042.
- Chengran Yang, Ting Zhang, Jinfeng Jiang, Xin Zhou, Haoye Tian, Mingzhe Du, Jieke Shi, Junkai Chen, Yikun Li, Eng Lieh Ouh, et al. 2026. Securepair: Semantics-aligned, curriculum-driven, and reasoning-enhanced vulnerability repair framework. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 32108–32123.
- Wei Yang, Jinwei Xiao, Hongming Zhang, Qingyang Zhang, Yanna Wang, and Bo Xu. 2025. [Coarse-to-fine grounded memory for LLM agent planning](#). In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 13029–13056, Suzhou, China. Association for Computational Linguistics.
- Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*.
- Zhenlei Ye, Xiaobing Sun, Sicong Cao, Lili Bo, and Bin Li. 2026. Well begun is half done: Location-aware and trace-guided iterative automated vulnerability repair. In *Proceedings of the 2026 IEEE/ACM 48th International Conference on Software Engineering*, pages 1096–1108.
- Luke Yoffe, Alfonso Amayuelas, and William Yang Wang. 2025. Debunc: Improving large language model agent communication with uncertainty metrics. In *EMNLP (Findings)*, pages 23299–23315.
- Zheng Yu, Wenxuan Shi, Xinqian Sun, Zheyun Feng, Meng Xu, and Xinyu Xing. 2026. Patch validation in automated vulnerability repair. *arXiv preprint arXiv:2603.06858*.
- Yifan Zhang, Jieyu Li, Kexin Pei, Yu Huang, and Kevin Leach. 2026. Synthfix: Adaptive neuro-symbolic code vulnerability repair. In *Findings of the Association for Computational Linguistics: ACL 2026*, pages 26857–26870.
- Yuntong Zhang, Xiang Gao, Gregory J Duck, and Abhik Roychoudhury. 2022. Program vulnerability repair via inductive inference. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 691–702.
- Xin Zhou, Kisub Kim, Bowen Xu, DongGyun Han, and David Lo. 2024. Out of sight, out of mind: Better automatic vulnerability repair by broadening input ranges and sources. In *Proceedings of the IEEE/ACM 46th international conference on software engineering*, pages 1–13.
- Xiaochen Zhu, Caiqi Zhang, Yizhou Chi, Tom Stafford, Nigel Collier, and Andreas Vlachos. 2026. Demystifying multi-agent debate: The role of confidence and diversity. In *Findings of the Association for Computational Linguistics: ACL 2026*, pages 33909–33930.

A Motivation for Hierarchical Evidence Aggregation

We compare the vulnerabilities successfully repaired by each exploration agent to determine whether the agents repair different vulnerabilities. The analysis uses the same 169 PatchEval vulnerabilities as Section 6.1. Each set contains the vulnerabilities whose patches pass both the benchmark PoV and unit tests. We count unsuccessful or missing attempts as failures. Figure 3 shows the overlap for both models.

Each agent repairs vulnerabilities that the others miss. With GPT-5.6-Luna, the Code Knowledge Graph agent achieves an RSR of 36.1%, compared with 29.6% for the Intent agent and 33.7% for the PoV agent. Their union covers 40.8% of the evaluation set, including 4.7% that the strongest agent misses. Moreover, each role contributes successes that neither of the other two achieves: 5.3% of the evaluation set for the Code Knowledge Graph agent, 0.6% for the Intent agent, and 3.0% for the PoV agent. The same pattern holds with DeepSeek-V4.1-Flash: the union covers 72.2% of the evaluation set, while the strongest agent, Intent, achieves an RSR of 59.8%. The Code Knowledge Graph, Intent, and PoV agents uniquely repair 2.4%, 7.7%, and 4.7% of the evaluation set, respectively. With both models, each agent successfully repairs some vulnerabilities that the other two do not. Using only the strongest agent would miss these repairs.

Aggregation must assess the attempts without benchmark feedback. These results motivate using all three agents’ findings. However, selecting a successful candidate for each vulnerability would require knowing which patches pass the benchmark tests. The agents do not receive those results, so aggregation must assess the observations behind each diagnosis and the implementation details in each patch. EXPLOREREPAIR uses lessons to summarize the agents’ vulnerability observations and patches to retain their concrete repair implementations. The Lesson Aggregator and Patch Aggregator independently synthesize repairs from these inputs, and a final Aggregator reconciles their proposals. Section 6.2 evaluates this design. Appendix C examines cases in which aggregation produces a new fix or loses a successful one.

B Exploration Agents

The exploration agents provide vulnerability observations and candidate patches for aggregation. The Code Knowledge Graph, Intent, and PoV agents investigate repair scope, intended behavior, and observed failures, respectively. This section gives further details of the agents described in Section 4.2 and how they gather evidence for their diagnosis. All three use the same ReAct scaffold and independently investigate the same vulnerable revision. They receive the repository and localization hint under the setting defined in Section 3. They can inspect files, execute shell commands and repository tests, and edit production code beyond the localized function. Each agent runs at medium reasoning effort with a time limit of 1800 s. It returns a production-code patch and a trajectory that records its observations.

B.1 Code Knowledge Graph Agent

The Code Knowledge Graph agent investigates where a repair should be applied and whether the same flaw appears along related paths. Before editing, it traces callers toward entry points and external inputs, examines the helpers and validators used by the localized code, and compares sibling paths that perform similar operations. This analysis can reveal that a guard belongs in a shared helper or that a local fix needs to cover another branch.

For each vulnerability, we construct a separate graph from the repository at its vulnerable revision, using Tree-sitter to parse source files into abstract syntax trees. The graph contains file nodes and symbol nodes for functions, methods, classes, and other language-specific definitions. Symbol nodes retain qualified names and source locations so that the agent can inspect the corresponding code. We resolve call sites and import statements to repository-local symbols and files to create call and import edges. When commit history is available, we add edges between files changed in the same commit. We use only commits up to the vulnerable revision and weight each edge by the number of commits in which the two files changed together.

Two tools support this investigation. `search_code` retrieves locations from a per-instance vector index built with `bge-large-en-v1.5`, returning five results by default. `graph_neighbors` lists incoming and outgoing relations for a symbol or file, with a

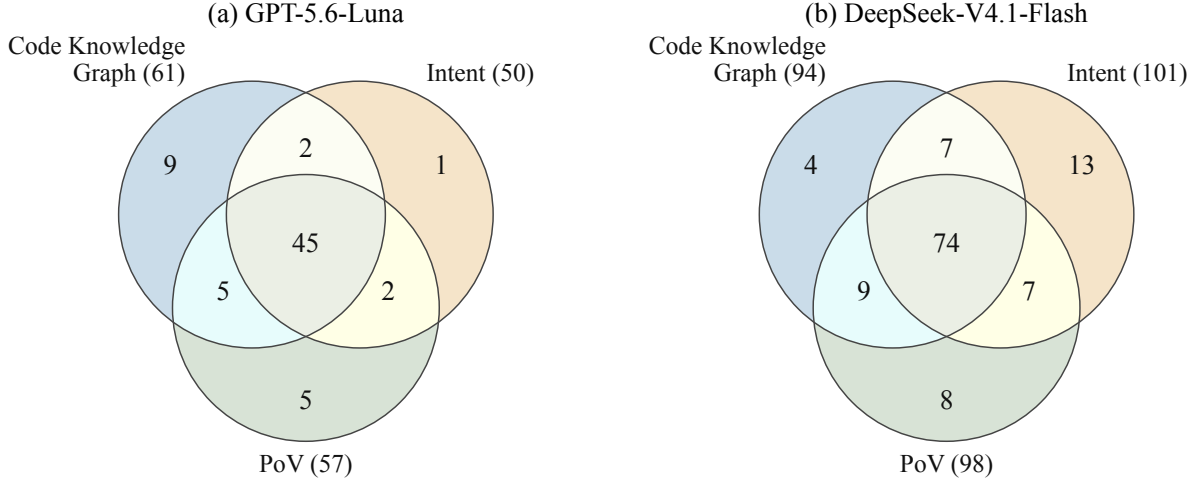


Figure 3: Overlap among successful repairs from the three evidence-specialized agents on 169 PatchEval vulnerabilities. Numbers indicate repairs achieved by exactly the agents whose circles intersect in that region; totals appear beside the agent names.

default limit of 20 neighbors per direction. Both tools help locate relevant source code, while the prompt requires the agent to read that code through the shell before relying on it. The agent records what the relationships imply about repair scope, then implements its patch. We use an LLM to distill the agent’s trajectory and proposed patch into a lesson that includes the repository relationships that led the agent to choose the repair location.

B.2 Intent Agent

The Intent agent establishes what the code is supposed to do before deciding how it is broken. Its analysis follows the four intent questions described in Section 4.2.2, within a single repair trajectory. It begins with the governing specification or API contract, looking for evidence in documentation, comments, tests, constants, and error messages. It then traces the parties whose inputs reach the localized code and determines what the repository assumes about their trustworthiness. Using those findings, it identifies the resources the code accesses and the checks already protecting them, including conventions followed by sibling paths.

The agent uses these findings to identify at most five constraints on untrusted input. Each constraint must name the relevant values and operations, explain when it applies, and identify where it should be enforced. The prompt requires file and line references and asks the agent to acknowledge claims that the repository does not establish. Only after completing this analysis does it diagnose the im-

plementation and produce a repair. Its trajectory records the expected behavior and trust assumptions behind the patch. These observations help explain why the proposed checks are needed.

B.3 PoV Agent

The PoV agent tests whether its diagnosis corresponds to an observable security violation. It first traces input from a public API, command-line interface, or endpoint to the localized code, then constructs a PoV through that entry point. The test checks the program’s behavior and must be executed against the vulnerable revision. The agent revises it until the observed failure demonstrates the suspected violation.

Once the test provides that evidence, the agent repairs the production code and reruns the same test. Its guidance asks it to address every site sharing the weakness and prohibits weakening the test or special-casing its input. The agent saves generated tests in recognizable test paths. The harness uses these paths to exclude the tests from the production-code diff submitted for evaluation. The trajectory records the input, the observed failure, and the response to the repair. Passing the generated PoV supports the agent’s diagnosis, but does not guarantee that its patch passes the hidden benchmark PoV. During aggregation, the agents must therefore check these execution observations against the repository and the other agents’ findings.

```

# fields contains the names requested by the user.
def set_values(self, fields):
    ...
    if fields:
        ...
        # Simple request: take the shortcut.
        if not self.extra and not self.annotations:
            # Copy the names without checking safety.
            field_names = list(fields)
        else:
            # More complex request: process each name.
            self.default_cols = False
            for f in fields:
                # f is one requested name.
                # No safety check here either.
                ...
    ...

```

Listing 1: Original Django code: both paths lack the safety check. Dots stand for code omitted from the excerpt.

C Qualitative Aggregation Analysis

We examine two cases in which aggregation produces a new fix or loses a correct one. Both cases use GPT-5.6-Luna.

C.1 Producing a Fix from Unsuccessful Attempts

CVE-2024-42005 affects how Django handles names supplied to a database query. These names must be checked for dangerous characters; otherwise, an attacker can use a crafted name to change the meaning of the database command, causing SQL injection. The affected function handles simple requests through a shortcut and more complex requests through a separate path (Listing 1). The missing protection is the same in both: the names are used without the required safety check.

Each exploration agent finds part of the problem, as shown in Listing 2. The Code Knowledge Graph agent checks names on the complex path, but misses the shortcut. The Intent agent adds a check in another function that creates query names. The PoV agent adds # to the list of forbidden characters, but does not make either path perform the check. All three repairs fail the benchmark PoV test because the attack takes the unchecked shortcut.

The Lesson Aggregator uses these findings to inspect the function again and notices that the shortcut still has no check. It keeps the earlier protections and adds a check for every name on the shortcut (Listing 3). The resulting repair passes both the benchmark PoV and unit tests.

Combining the exploration agents’ patches leaves the shortcut unprotected, as does the Patch

```

# Code Knowledge Graph agent: attack test fails.
# This edit is on the complex path only.
else:
    self.default_cols = False
    for f in fields:
        # Reject a name containing forbidden characters.
        self.check_alias(f)
        # Remaining code omitted.

# Intent agent: attack test fails.
# Check a name created by a different function.
def add_filtered_relation(self,
                        filtered_relation, alias):
    self.check_alias(alias)
    filtered_relation.alias = alias
    ...

# PoV agent: attack test fails.
# This pattern lists forbidden characters.
# Add #, which can start an SQL comment.
# The pattern helps only where a check is called.
FORBIDDEN_ALIAS_PATTERN = _lazy_re_compile(
    r"['\"`\\]\[\;\s]|--|/\*|\/|/|"
)

```

Listing 2: Specialized agents’ Django repairs: each adds a useful protection, but none checks the shortcut used by the attack.

```

# Keep the PoV agent's forbidden-character change.
FORBIDDEN_ALIAS_PATTERN = _lazy_re_compile(
    r"['\"`\\]\[\;\s]|--|/\*|\/|/|"
)

# Keep the Intent agent's check in this function.
def add_filtered_relation(self,
                        filtered_relation, alias):
    self.check_alias(alias)
    filtered_relation.alias = alias
    ...

# NEW: check names on the simple path too.
if not self.extra and not self.annotations:
    # Visit each requested name and check its safety.
    for f in fields:
        self.check_alias(f)
    # Use the names only after these checks.
    field_names = list(fields)
else:
    self.default_cols = False
    for f in fields:
        # Keep the Code Knowledge Graph agent's check.
        self.check_alias(f)
        # Remaining code omitted.

```

Listing 3: Successful aggregated Django repair. The highlighted lines add the missing shortcut check, which no specialized agent produced.

Aggregator’s repair. The Lesson Aggregator supplies the missing check, which the final Aggregator retains. The final repair passes both the benchmark PoV and unit tests, even though all three exploration agents’ repairs fail.

C.2 Losing a Correct Fix During Aggregation

CVE-2021-39163 affects how Synapse lists rooms associated with a group. Group membership does not imply membership in each room, so a group member must not receive private-room metadata solely because the room is listed in the group. The

```

# Group visibility and room visibility are distinct.
room_results = await self.store.get_rooms_in_group(
    group_id, include_private=is_user_in_group
)
chunk = []
for room_result in room_results:
    room_id = room_result["room_id"]
    joined_users = await self.store.get_users_in_room(room_id)
    entry = await self.room_list_handler.generate_room_entry(
        room_id, len(joined_users),
        with_alias=False, allow_private=True
    )
    if not entry:
        continue
    # Remaining entry fields omitted.
    chunk.append(entry)
return {"chunk": chunk,
        "total_room_count_estimate": len(room_results)}

```

Listing 4: Original Synapse listing: private-room metadata is allowed regardless of room membership.

```

# Code Knowledge Graph agent: PoV and unit tests pass.
allow_private = requester_user_id in joined_users
entry = await self.room_list_handler.generate_room_entry(
    room_id, len(joined_users), with_alias=False,
    allow_private=allow_private,
)
# Remaining loop omitted; count only returned rooms.
return {"chunk": chunk, "total_room_count_estimate":
        len(chunk)}

```

```

# Intent and PoV agents: benchmark PoV fails.
# Both use group membership in the listing and summary.
entry = await self.room_list_handler.generate_room_entry(
    room_id, len(joined_users),
    with_alias=False,
    allow_private=is_user_in_group,
)

```

Listing 5: Exploration agents’ Synapse repairs: only the Code Knowledge Graph agent uses room membership as the authorization boundary.

original code passes `allow_private=True` when generating room entries, bypassing the private-room visibility check (Listing 4).

The Code Knowledge Graph agent checks whether the requester belongs to the room and counts only the returned entries (Listing 5). Its repair passes both the benchmark PoV and unit tests. The Intent and PoV agents instead use group membership to control private-room visibility; both repairs fail the benchmark PoV.

The Lesson Aggregator adopts the group-membership interpretation. The Patch Aggregator also selects it, explicitly rejecting the room-membership check as inconsistent with the group API’s visibility semantics and omitting the count correction. The final Aggregator accepts this shared interpretation and produces the same unsuccessful changes (Listing 6).

The final repair fails the benchmark PoV: a group member who has not joined a private room still re-

```

# Both intermediate aggregators and the final Aggregator
# use this check in the listing and summary paths.
entry = await self.room_list_handler.generate_room_entry(
    room_id, len(joined_users), with_alias=False,
    allow_private=is_user_in_group,
)
# The listing still counts rooms before filtering.
return {"chunk": chunk,
        "total_room_count_estimate": len(room_results)}

```

Listing 6: Unsuccessful final Synapse repair: group membership still permits disclosure of private-room metadata.

ceives its name and metadata. This case shows that the complete aggregation phase can discard a successful exploration repair when both intermediate aggregators adopt the same incorrect authorization assumption.

D Baseline Configurations and Exclusions

IntentFix. IntentFix infers developer intent from the vulnerable file to generate and refine a patch (Heo et al., 2026).

EvoRepair. EvoRepair retrieves experiences from prior trajectories to guide successive repair attempts (Hu et al., 2026). Its original implementation accesses the ground-truth PoV and benchmark evaluation feedback; we hide both in our experiments. The original study allows up to 15 repair turns, whereas our adaptation allows up to three refinement turns under the time limit described in Section 5.6.

LoopRepair. LoopRepair is designed for C/C++ and uses PoVs and taint traces to rank and refine LLM patches (Ye et al., 2026). We evaluate both its released PoV-assisted setting and a no-PoV variant on VulnLoc+. In the no-PoV variant, we replace CrashRepair’s PoV-based localization with the same reported crash location provided to EXPLORE REPAIR and select the first generated patch that compiles, without PoV validation or iterative refinement.

LLM-based baselines. The standalone large language model (LLM) receives only the code at the vulnerable location and returns replacement functions in one response. It cannot inspect other files, use tools, or receive feedback. Single Agent uses a ReAct scaffold with shell tools and medium reasoning effort. Multi-Agent extends this scaffold with a `spawn_subagent` tool, allowing a lead agent to create subagents of itself and assign subtasks to

them. The lead agent uses their reports and code changes to produce the final repair.

Excluded approaches. We select baselines that support the target languages and can run with the vulnerability information available in our setting. We exclude training-based approaches, such as SeCuRepair (Yang et al., 2026) and VulMaster (Zhou et al., 2024), which learn from reference fixes, because EXPLOREPAIR is an inference-time approach. We attempted to evaluate APPATCH using its released implementation, but could not run it successfully in our environment. ContraFix requires a PoV test and therefore cannot be used in our setting (Liu et al., 2026). AgenticRepair and KeaRepair had no publicly available replication packages at the time of evaluation (Fu et al., 2026; Cao et al., 2026).