CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Agentic AI Framework for Web Vulnerability Detection, Mitigation and Patching

Master's thesis in Computer science and engineering

Xuanhao Liu, Jiangzhao Xie

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2026

Master's thesis 2026

Agentic AI Framework for Web Vulnerability Detection, Mitigation and Patching

Xuanhao Liu, Jiangzhao Xie

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2026

Agentic AI Framework for Web Vulnerability Detection, Mitigation and Patching

Xuanhao Liu, Jiangzhao Xie

Supervisor: Atmane Ayoub Mansour Bahar, Mohamed Hashim Changrampadi, Department of Computer Science and Engineering
Examiner: Romaric Duvignau, Department of Computer Science and Engineering

Master's Thesis 2026
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2026

Agentic AI Framework for Web Vulnerability Detection, Mitigation and Patching

Xuanhao Liu, Jiangzhao Xie
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Modern web applications expose increasingly complex attack surfaces, and existing security automation is still most effective at producing candidate findings rather than reviewable repairs. Static and dynamic analysis tools can identify possible weaknesses at scale, but the path from a finding to a validated, scoped, and review-ready patch remains weakly automated and difficult to inspect. This thesis investigates whether a multi-stage agentic AI workflow can improve web vulnerability detection, mitigation, and patch delivery in white-box repository settings.

To examine this question, the proposed framework decomposes security review into stages for discovery, triage, analysis, mitigation, verification, and delivery. Each stage produces structured artifacts that are consumed by later stages, making the workflow more inspectable than a single end-to-end repair agent. The design emphasizes repository-grounded evidence, explicit repair hypotheses, independent verification, and reviewer-oriented delivery units.

The evaluation combines issue-level repair experiments with repository case studies. On the PatchEval runtime-validated subset of 230 vulnerability-repair tasks, the multi-stage workflow achieves 78 successful repairs, compared with 73 for a matched single-agent baseline under the same model. This improvement is modest and requires higher token use and cost, but case-level analysis suggests that staging can influence repair strategy by clarifying vulnerability boundaries and providing independent feedback on patch completeness. In repository case studies, agentic discovery and triage cover more answer-key vulnerabilities than CodeQL, Semgrep, and OWASP ZAP, especially for issues that require cross-file or design-level reasoning. The workflow also improves reviewability by organizing patches and reports into structured delivery units.

Overall, the results indicate that the main value of multi-stage orchestration is not a large increase in repair success, but a more inspectable and controllable security-review process. The workflow records repository evidence, analysis assumptions, verification results, and delivery boundaries, helping reviewers understand why a vulnerability was found, how it was analyzed, whether the patch addresses the issue, and how the repair should be reviewed. These benefits remain constrained by model capability and cost.

Keywords: Attack Mitigation, Vulnerability Detection, Code Analysis, Cybersecurity, Artificial Intelligence

Acknowledgements

We would like to express our sincere gratitude to our supervisors, Atmane Ayoub Mansour Bahar and Mohamed Hashim Changrampadi, for their guidance, feedback, and encouragement throughout this thesis. Their advice helped us refine the research direction, structure the experimental evaluation, and think more critically about the practical limits of agentic security workflows.

We are also grateful to our examiner, Romaric Duvignau, and to the Chalmers University of Technology for providing the academic environment in which this work was carried out.

We would like to thank our friends for their discussions, feedback, and support during the development and writing of this thesis. Finally, we thank our families for their financial support and continuous encouragement throughout our studies.

Xuanhao Liu and Jiangzhao Xie, Gothenburg, 2026-06-19

Contents

List of Figures

List of Tables

List of Acronyms

AI	Artificial Intelligence
API	Application Programming Interface
APR	Automated Program Repair
AVR	Automated Vulnerability Repair
CSRF	Cross-Site Request Forgery
CSV	Comma-Separated Values
CVE	Common Vulnerabilities and Exposures
CVSS	Common Vulnerability Scoring System
CWE	Common Weakness Enumeration
DAST	Dynamic Application Security Testing
FIRST	Forum of Incident Response and Security Teams
FP	False Positive
GHSA	GitHub Security Advisory
GPT	Generative Pre-trained Transformer
HMAC	Hash-Based Message Authentication Code
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
JWT	JSON Web Token
LLM	Large Language Model
OS	Operating System
OSV	Open Source Vulnerabilities
OWASP	Open Worldwide Application Security Project
PHP	PHP: Hypertext Preprocessor
PoC	Proof of Concept
PR	Pull Request

RQ	Research Question
RSA	Rivest-Shamir-Adleman
SARIF	Static Analysis Results Interchange Format
SAST	Static Application Security Testing
SQL	Structured Query Language
SSRF	Server-Side Request Forgery
SWE	Software Engineering
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
USD	United States Dollar
WHATWG	Web Hypertext Application Technology Working Group
XML	Extensible Markup Language
XSS	Cross-Site Scripting
ZAP	Zed Attack Proxy

1

Introduction

Modern web applications have become increasingly complex, expanding their attack surface and leading to a surge in new vulnerabilities [1]. Current vulnerability detection already benefits from automation through static and dynamic tools, which can identify many candidate weaknesses at scale. By contrast, mitigation, validation, and final delivery still largely require human intervention, because acting on a finding requires repository-level analysis about root cause, patch scope, and behavioral side effects.

Traditional automated repair research has attempted to address software defects through automated program repair (APR). Early heuristic-based APR approaches, such as GenProg [2], relied on genetic programming and test-suite validation to search for patches, which often led to "plausible" but semantically incorrect fixes. Later learning-based APR systems, such as SequenceR [3], learned patch patterns from existing code changes. Despite these improvements, these approaches still depend heavily on task-specific training data and remain constrained by limited context.

However, general APR mainly targets functional bugs rather than security vulnerabilities. Automated vulnerability repair (AVR), especially for web applications, received comparatively less attention. Existing web-focused repair systems are often limited to pattern-based vulnerabilities such as SQL injection and cross-site scripting, and many are designed around specific languages or frameworks, particularly PHP-based web applications. As a result, they remain less suitable for complex vulnerabilities.

Large language models (LLMs) offer a different step forward. Given sufficiently relevant code and repository context, they can often generate plausible patch code without task-specific fine-tuning, making automated mitigation more accessible than in earlier repair pipelines. Yet this capability alone is not enough for dependable security review. LLM outputs remain prone to hallucination, false confidence, and patches that fail even simple tests or validation checks [4]. Agentic systems address these limitations by introducing a closed-loop process in which reasoning is interleaved with action [5]. Through using tools to inspect repository state and interact with an execution environment, agents can adapt their plan based on intermediate feedback, which can improve reliability and reduce hallucination.

Multi-agent orchestration can be organized into four common forms: Layered, De-

centralized, Centralized, and Shared Message Pool [6]. In a single-agent setup, discovery, analysis, patch generation, and validation are typically assigned to the same agent under a unified prompt and control loop. This design is simple and practical, but as observed in RepairAgent [7], in complex repository-level security tasks it places greater demands on responsibility separation, context management, and self-verification. Although multi-agent communication provides clearer role separation, open-ended interaction between agents can make the process harder to inspect, reproduce, and evaluate [8].

In this thesis, we choose to apply a multi-stage workflow for the proposed Agentic AI Framework. This framework covers discovery, triage, analysis, mitigation, verification, and delivery. Discovery collects candidate vulnerability evidence from the repository, and triage suppresses noise and groups related candidates into cases. This design is chosen for reliability and evaluability. Separating the stages reduces the risk that one agent both generates and validates its own conclusion. It also makes intermediate outputs easier to inspect, debug, and compare across experiments. Therefore, the multi-stage workflow is not only an implementation choice, but also a methodological choice for controlled evaluation of agentic security review.

1.1 Challenge

Automating security mitigation at the repository level introduces several challenges beyond patch generation.

Firstly, patches cannot be treated as independent units. In real-world scenarios, multiple vulnerabilities may be identified within the same repository, and their corresponding patches may modify overlapping files or code regions. Submitting a separate patch for each finding is often impractical, as it can lead to conflicts, redundant changes, or inconsistent behavior. Ensuring that multiple patches can be correctly composed into a coherent and functional update is therefore a key problem.

Secondly, security reasoning often depends on repository-level context [9]. An agent may need to trace how a function, variable, route, or data object is used across the codebase before it can understand the root cause and patch scope of a vulnerability. This is difficult because call relationships and data flows are expressed differently across programming languages and frameworks. In addition, functions may be invoked indirectly through reflection, callbacks, routing conventions, or configuration files, while variables and objects may be renamed, wrapped, or transformed across layers. As a result, generating a correct fix based only on local code snippets is often insufficient.

Thirdly, ensuring the correctness of generated patches remains difficult. A patch may appear syntactically valid and pass basic tests, but still fail to fully address the vulnerability or introduce unintended side effects. This is especially challenging for security fixes that depend on implicit assumptions, input validation rules, or business logic that are not explicitly captured in tests.

Finally, traditional static and dynamic tools remain useful sources of candidate

findings, but their outputs are imperfect. Missed detections can hide relevant cases from the workflow, while false positives can overload later stages with noise and unnecessary analysis.

1.2 Scope

This thesis focuses on code-level security review and repair for web application vulnerabilities in a blue-team setting. The study assumes white-box access to repository source code, allowing agents to inspect implementation details and execute code in a controlled development environment. The scope is limited to application-layer vulnerabilities that can be analyzed and mitigated through repository source code, such as injection flaws, authorization errors, unsafe data exposure, insecure input handling, and related web security weaknesses.

It does not address vulnerabilities whose primary cause lies outside the application codebase, such as operating-system, network-protocol, cloud-configuration, or deployment-infrastructure issues. Hardcoded secrets are considered only when they appear in the source code directly; comprehensive secret scanning, supply-chain vulnerability scanning, and dependency upgrade management are outside the scope.

1.3 Research Objectives and Questions

The primary objective is to design and evaluate a multi-stage agentic framework for repository-based security workflows. It should also make the tradeoff between effectiveness, coverage, and cost explicit, especially when different model capacities are assigned to different stages.

The thesis is organized around three research questions:

- **RQ1:** How does a multi-stage agentic framework for analysis, mitigation, and verification compare with a single-agent baseline on specified vulnerability repair tasks?
- **RQ2:** How do agent-based discovery and triage compare with traditional static and dynamic scanning tools in terms of vulnerability coverage and detectable issue scope?
- **RQ3:** How does the workflow improve the reviewability and delivery readiness of agent-generated security repairs?

1.4 Contributions

This thesis makes the following contributions:

- We propose a multi-stage agentic architecture for repository-based security review, covering the full workflow from vulnerability discovery to the final delivery of patches and review reports. The design separates vulnerability

analysis, mitigation, and verification into distinct stages, improving reliability and enabling controlled evaluation.

- We add discovery and triage to support repository review, complementing traditional static and dynamic analysis tools. Instead of replacing existing scanners, the framework integrates their outputs with agent-based discovery to reduce missed findings and support repository review.
- We address the problem of patch composition at the repository level. The framework considers how multiple patches interact when applied to the same codebase, and supports the merge of multiple patches based on their vulnerability types, modified files, and code regions.
- We conduct a systematic evaluation of stage-specific model allocation, analyzing the tradeoffs between effectiveness and cost under different configurations.

1.5 Thesis Outline

The remainder of this thesis is organized as follows. Chapter 2 introduces the background needed for the work, including web vulnerability repair, LLM agents, repository-grounded execution, and traditional security testing tools. Chapter 3 reviews related work in automated program repair, automated vulnerability repair, and repository-level LLM agents, and positions this thesis in relation to these areas. Chapter 4 describes the proposed multi-stage framework and its implementation. Chapter 5 presents the evaluation setup and results for both issue-driven repair and repository review. Chapter 6 discusses the findings, limitations, and practical implications. Chapter 7 concludes the thesis, and Chapter 8 outlines possible directions for future work.

2

Background

This chapter introduces the concepts needed to understand the proposed agentic framework. It first explains the characteristics of web vulnerability repair, then summarizes the basic principles of LLM agents, agentic workflows, security scoring standards, traditional scanning tools, and repository-grounded execution environments.

2.1 Characteristics of Web Vulnerability Repair

Web vulnerability repair focuses on exploitable behavior in web applications. The relevant security properties often involve authentication, authorization, input handling, session management, cross-user isolation, client-side output, and interaction with databases or external services [10]. A repair must therefore remove or reduce the vulnerable behavior while preserving intended application behavior for legitimate users.

The input evidence in this thesis mainly comes from three sources: issue reports, pull-request changes, and scanner alerts. An issue report may describe a suspected vulnerability and may include supporting evidence such as logs, screenshots, error messages, or proof-of-concept HTTP requests collected by developers. A pull request may introduce or modify security-relevant behavior, especially when it changes routes, authentication logic, authorization checks, input handling, or data access. Scanner alerts may point to suspicious routes, files, functions, or data-flow paths identified by static or dynamic analysis tools. However, these inputs are often uncertain and incomplete, and may not directly indicate whether a real vulnerability exists or identify its root cause. A repair workflow must therefore first determine whether the candidate represents a real vulnerability, then analyze its cause and scope before deciding what should be changed.

The relevant context depends heavily on how the specific application is structured and implemented. Web behavior is shaped by routes, controllers, templates, middleware, database models, framework conventions, dependency configuration, environment variables, and deployment settings. A vulnerability may be caused by the interaction between these layers rather than by one isolated line of code. For example, an access-control issue may involve route registration, authentication middleware, object ownership checks, and database queries. A repair must therefore

consider how the application behaves across the repository, rather than focusing on isolated code snippets.

Validation also needs to consider both security and functionality. A web vulnerability repair needs evidence that the relevant web interaction has been secured and that legitimate workflows still work. Passing functional tests is useful, but it may not prove that the vulnerability has been fully fixed. Conversely, a patch that blocks an exploit may still be unacceptable if it breaks valid user behavior, changes public API semantics, or bypasses framework conventions.

Finally, web vulnerability repair must be delivered as changes that can be reviewed, merged, and maintained within a development workflow. The output is therefore not only a patch, but also an inspectable change set. A repair may need to coexist with other security fixes, avoid overlapping edits, update tests or configuration, and provide an explanation that helps reviewers understand the security impact. These characteristics make web vulnerability repair a repository-level security workflow rather than only a program transformation problem.

2.2 LLM Agents, ReAct, and Context Engineering

The ReAct paradigm is a common foundation for this style of system. ReAct, short for reasoning and acting, interleaves reasoning traces with task-specific actions so that the model can update its plan based on observations from tools or environments [5]. In a repository setting, the action space may include reading files, searching for symbols, running tests, or applying patches. The key idea is that tool use connects the model to external evidence, while reasoning helps decide which tool result matters and what should be attempted next.

Tool use is especially important for security repair because the model should not rely only on the vulnerability description or on memorized programming patterns. A repair agent needs to inspect the actual implementation, identify the relevant data flow or control flow, modify the correct files, and validate the result in the repository. Tool calls therefore provide an execution mechanism: they allow the agent to test its assumptions against the actual codebase and its execution results.

Prompt engineering and context engineering are two supporting techniques for making such agents useful in practice. Prompt engineering concerns how the task is expressed to the model: the role definition, objectives, constraints, output schema, examples, and failure-handling instructions. In this thesis, prompts are used to separate responsibilities between stages and to make agent outputs structured enough for downstream processing.

Context engineering concerns what information is made available to the model and when. Repository-scale security review cannot place an entire codebase, all logs, all scanner alerts, and all previous reasoning into every model call. The system must select relevant files, issue text, pull-request diffs, scanner output, test results, and prior stage artifacts while excluding unrelated or misleading material. Context

engineering is therefore central to agents, because the quality of the model's reasoning depends not only on the model itself, but also on whether the supplied context represents the actual security problem.

2.3 Agentic Workflow

An *agentic workflow* is the control structure around one or more LLM agents. It defines how tasks are decomposed, which tools are available, what artifacts each step should produce, and how later steps use feedback from earlier steps. A simple agentic workflow may use one agent in a loop: observe the task, reason about the next action, invoke a tool, inspect the result, and continue until completion.

A multi-stage workflow applies the same principle with explicit role separation. For example, one stage may analyze the vulnerability, another may generate a patch, and a later stage may verify whether the patch is trustworthy. This separation matters because security repair involves different kinds of judgement: deciding whether a candidate is real, identifying the root cause, editing the repository, checking side effects, and packaging the result for review. Treating these as separate workflow steps makes intermediate outputs easier to inspect.

In the proposed framework, agentic workflow design is therefore not only an implementation detail. It is the mechanism that connects LLM reasoning, tool use, repository context, and developer-oriented delivery. The later methodology chapter describes the concrete stages used in the system.

2.4 Security Standards and Tooling Foundations

Security review workflows often need to prioritize findings before human review or delivery. The Common Vulnerability Scoring System (CVSS) provides a standardized way to describe and score the severity of vulnerabilities. CVSS v4.0 [11] is defined by FIRST as an open framework for communicating vulnerability characteristics and severity, with metric groups covering base, threat, environmental, and supplemental information. In this thesis, CVSS is relevant because the proposed workflow includes a scoring stage after analysis has retained a confirmed or scoreable case. The score is not used as proof that a vulnerability exists; instead, it provides guidance for developers on how to prioritize vulnerability mitigation.

Static Application Security Testing (SAST) and Dynamic Application Security Testing (DAST) are two traditional families of application security tools. SAST inspects source code or compiled code without running the application. OWASP describes source code analysis tools, also known as SAST tools, as tools that analyze source code or compiled code to help find security flaws [12]. In practice, SAST tools often use pattern matching, taint analysis, data-flow analysis, or language-specific rules to identify suspicious code. They are useful because they can run early in development and scale across large repositories, but they may miss runtime behavior and may produce false positives when they lack application context.

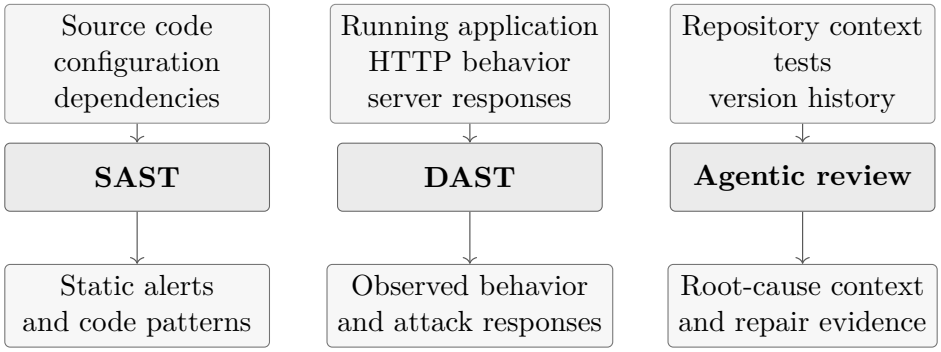


Figure 2.1: Complementary evidence views in web vulnerability review.

DAST works from the opposite direction. Instead of reading source code, DAST interacts with a running application from the outside. OWASP describes DAST as black-box testing that can find vulnerabilities in a running application by injecting malicious payloads and observing the application's behavior [13]. This makes DAST useful for detecting issues that depend on runtime behavior, HTTP interfaces, authentication flows, or server configuration. However, DAST usually has limited visibility into the root cause in source code and depends on application coverage, test data, and scanner configuration.

These tools are therefore complementary rather than interchangeable. Some vulnerabilities are difficult to identify from either static code patterns or runtime behavior alone, because they require connecting evidence from both source-code structure and observed application behavior. In such cases, an agent-based discovery stage can complement traditional scanners by inspecting repository context, linking static and dynamic evidence, and identifying potential vulnerabilities that may otherwise be missed.

Figure 2.1 summarizes these complementary evidence views. SAST mainly reasons from source-code structure, DAST mainly reasons from observed runtime behavior, and agentic review can combine these signals with project context before moving toward repair.

2.5 Repository-Grounded Execution

Repository-grounded execution means that the agent operates against the actual software repository rather than against isolated snippets. The repository is treated as the main source of truth: source files, tests, dependency manifests, configuration files, documentation, issue context, pull request diffs, and version-control history can all become evidence for analysis and repair. This is important because many web vulnerabilities are caused by interactions across files and layers, not by a single defective line.

A repository-centered execution environment usually combines three elements. First, it provides a materialized workspace, often a local clone or checked-out copy of the target repository. Second, it provides controlled tool access, such as file search, build

commands, test execution, static analysis, or patch application. Third, it isolates risky operations through a sandbox, container, or similar boundary. Docker-based sandboxes are common because they can package dependencies and services while limiting the effect of executed code on the host system.

Version-control interaction is also part of repository-grounded execution. A practical repair workflow must know which files changed, how a patch differs from the base version, whether several fixes conflict, and how results should be published back to a developer workflow. Git diffs, branches, commits, pull requests, and review comments are therefore not only delivery mechanisms; they are also inspection artifacts that make the agent's actions auditable.

3

Related Work

This chapter positions the proposed framework with respect to traditional application security testing, automated program repair, automated vulnerability repair, and LLM-based software engineering agents. Traditional SAST and DAST tools represent the established foundation for automated vulnerability discovery, but their role is primarily limited to identifying candidate findings rather than supporting mitigation, verification, and delivery. Repair-oriented research shows a clear evolution from search-based patch generation, to learning-based repair models, and then to repository-level agentic systems. Across this evolution, the scope of repair has gradually expanded from isolated patch generation toward broader, workflow-oriented problem settings. However, existing approaches still remain insufficient for multi-stage, end-to-end web vulnerability review that combines discovery, triage, analysis, mitigation, verification, and delivery.

3.1 Traditional Application Security Testing

Traditional automated discovery largely relies on Static and Dynamic Application Security Testing (SAST and DAST). Representative systems such as Pixy [14] and SecuBat [15] show that static taint analysis and black-box dynamic scanning can effectively identify common web vulnerability patterns at scale. However, these tools exhibit significant limitations when applied to complex web applications. SAST tools struggle with inter-procedural data flows and lack runtime context, often resulting in high false-positive rates and a tendency to miss sophisticated, real-world CVEs [16], [17]. DAST tools, on the other hand, evaluate running applications but lack source-code visibility. This inherently limits their ability to locate root causes or generate actionable patches, and prior studies show that DAST tools also struggle with stored vulnerabilities, active content, and tool-dependent detection coverage [18], [19]. Although combining SAST and DAST can improve detection coverage, traditional scanners still mainly produce isolated alerts rather than repository-level reasoning over the vulnerability, its root cause, and its potential fix. Therefore, they remain insufficient in the workflow, lacking the repository-level contextual understanding required to bridge the transition from vulnerability detection to end-to-end mitigation.

3.2 Automated Program Repair

APR aims to reduce the manual effort required to fix software defects. APR covers several families of repair techniques, including generate-and-validate and semantics-based approaches [20], as well as learning-based approaches [21]. Generate-and-validate methods search a predefined patch space and validate candidate patches with test suites, semantics-based methods use specifications or symbolic reasoning to synthesize repairs, and learning-based methods learn repair patterns from historical bug fixes. Recent neural and large-language-model-based APR systems further formulate repair as code generation or code transformation [21].

Early APR work was dominated by generate-and-validate approaches [2], [22], [23], where a test suite commonly serves as the behavioral oracle: a generated patch is considered plausible if it makes the failing test pass while preserving the passing tests. GenProg [2] is a representative example of this line of work, using genetic programming, test coverage information, existing program statements, and patch minimization to search for plausible repairs. However, the main limitation of such approaches is that test-suite adequacy does not imply semantic correctness [24], [25]. A patch may pass the available tests while overfitting to the test suite or changing intended behavior in untested cases [26]. This concern is especially important for security repair, where a patch must both remove exploitable behavior and preserve legitimate behavior.

Learning-based APR shifts repair from manually designed search spaces toward prediction over historical bug-fixing examples. Zhang et al. [21] describe the common workflow of learning-based APR in terms of fault localization, data preprocessing, patch generation, patch ranking, validation, and correctness assessment. This view is useful for this thesis because it frames patch generation as only one component within a broader repair pipeline, rather than the repair problem itself. SequenceR [3] is an early neural repair system that treats program repair as a sequence-to-sequence translation task and constructs an abstract buggy context around the suspicious line. However, its focus on one-line patches and fault-localized inputs illustrates a broader limitation of many learning-based approaches: repair quality depends heavily on accurate localization, suitable training data, and sufficient context. This limitation becomes more severe when extending these methods to vulnerability repair, where exploitability and cross-component interactions must be considered.

3.3 Automated Vulnerability Repair

Automated vulnerability repair (AVR) narrows the repair problem to security flaws. Compared with general APR, AVR must reason about exploitability, attacker-controlled inputs, security invariants, and the distinction between blocking an exploit and preserving valid behavior. VulRepair [27] is a representative neural AVR system that uses a pretrained code model to generate patches for vulnerable functions. Its evaluation over real-world vulnerability fixes shows that pretrained code representations improve perfect prediction accuracy compared with earlier

vulnerability repair models.

Despite these improvements, VulRepair remains primarily function-centered. The input is a localized vulnerable function, and the output is a candidate repair for that function. This formulation implicitly assumes that vulnerabilities can be isolated and repaired within a single function. While effective for known, localized vulnerabilities, it becomes less suitable for repository-level web vulnerabilities whose root cause may involve routes, middleware, templates, database models, authorization checks, and configuration spread across multiple files.

More recent LLM-based vulnerability patching systems reduce the dependence on task-specific fine-tuning. APPatch [28] uses adaptive prompting, vulnerability semantics reasoning, and semantics-aware scoping to guide LLMs toward vulnerability root causes and patches. Its results suggest that directly prompting an LLM to generate patches is weaker than structured reasoning with curated exemplars. PatchAgent [29] takes a more end-to-end view by integrating fault localization, patch generation, and validation in an autonomous agent assisted by a language server and patch verifier. These systems move closer to practical vulnerability mitigation, but their evaluations primarily target C/C++ memory-safety vulnerabilities or localized vulnerability instances. The web-application setting considered in this thesis additionally requires reasoning about application-layer behavior, framework conventions, and repository-level patch composition. While these neural and prompting-based approaches have shown promise, they remain largely optimized for the semantics of standalone functions or memory-safety issues in C/C++. In the context of web applications, vulnerabilities often emerge from the interaction between stateful components, middleware configurations, and framework-specific routing conventions. Such characteristics represent a higher-level "behavioral" complexity that is rarely captured by localized function-level repair models, thereby requiring an approach that can reason across the entire application.

3.4 LLM Agents for Repository-Level Software Engineering

LLM agents extend one-shot generation by allowing models to inspect files, search repositories, edit code, execute commands, and adapt based on feedback. To complete a complex task within a codebase, LLM agents must overcome challenges related to action execution, task decomposition, and information processing. By analyzing how recent studies address these challenges, existing work on repository-level agents can be classified along three key dimensions: interaction interfaces, role decomposition, and context management.

In terms of interaction interfaces, RepairAgent [7] applies LLMs to APR by equipping them with repair-specific tools for reading code, searching for repair ingredients, applying patches, and running tests. Unlike fixed feedback-loop systems, it allows the model to decide which action to take next through a dynamic prompt and a finite-state control mechanism. SWE-agent further argues that agent performance depends on interfaces designed specifically for language models rather than generic

shell access [30]. Its agent-computer interface provides structured commands for navigation, search, editing, execution, and context management, highlighting the importance of action space and feedback design.

In terms of role decomposition, multi-agent systems introduce explicit separation of responsibilities. MAGIS [31] proposes a framework with Manager, Repository Custodian, Developer, and Quality Assurance agents for GitHub issue resolution. AutoPatch [32] similarly combines multiple agents with retrieval over recent CVE data to verify and patch vulnerable code generated by LLMs. These systems demonstrate that role separation can improve coordination in repository-level tasks.

In terms of context management, repository context remains a central bottleneck. RepoGraph [33] addresses this by constructing a repository-level code graph that captures definitions and references at line level, and retrieving relevant subgraphs to guide LLM reasoning. This line of work is important because many failures in repository-level repair arise from incomplete or noisy context. For vulnerability repair, graph- or retrieval-based methods can help locate related files, reveal call relationships, and trace data dependencies.

While these directions expand repair from isolated patch generation toward repository-level workflows, they do not explicitly model a full defensive security-review lifecycle that spans discovery, analysis, mitigation, verification, and delivery.

3.5 Position of This Thesis

The proposed framework builds on these research directions but targets a different workflow boundary. It is broader than issue-only repair agents because it explicitly includes discovery and triage for repository review, and narrower than general-purpose coding assistants because it focuses on defensive security review for web vulnerabilities. It also differs from traditional scanners by extending beyond detection to include mitigation and verification, and from patch-generation systems that implicitly treat repair as a code-editing task without modeling repository-scale discovery.

Existing issue-to-patch agents typically assume that a reasonably well-formed problem description already exists, and therefore concentrate on downstream repair. In AVR and repair-agent settings, the input is usually treated as a true repair task rather than as a claim whose validity must first be established. Traditional scanners operate at a different stage of the workflow: they identify candidate findings but do not provide repository-grounded mitigation, patch verification, or repository-level delivery. Traditional APR and learning-based repair similarly focus on patch generation under assumptions such as available failing tests, accurate fault localization, or localized vulnerable functions. Recent LLM agents expand the workflow through repository interaction and tool use, while multi-agent systems introduce role separation for issue resolution and patching.

The contribution of this thesis is therefore not the generic use of multiple agents, nor the proposal of a new model. Instead, it is a multi-stage and auditable security-

review workflow with explicit handoffs between discovery, triage, analysis, mitigation, verification, and delivery. Candidate generation and case formation are separated from finding assessment, root-cause analysis, patch generation, validation, and developer-oriented output.

This design addresses three gaps in the reviewed work. Firstly, it does not assume that a vulnerability has already been clearly described, validated, or localized for repair, but instead supports discovery, triage, and analysis before mitigation. Secondly, it emphasizes repository-level context and patch composition, which are necessary when multiple web vulnerabilities affect overlapping files or shared application logic. Thirdly, it enables controlled evaluation through matched single-agent baselines for repair tasks, scanner-supplied baselines for repository review, and ablations that test whether discovery and triage, independent verification, and stage-specific model allocation provide measurable value.

4

Methodology

This chapter describes how the thesis operationalizes the proposed multi-stage agentic framework. The framework focuses on workflow design instead of task-specific fine-tuning. It defines how repository evidence is collected, how responsibility is separated, how prompt design differs between stages, and how patches and review artifacts are produced and checked.

4.1 System Overview

Figure 4.1 presents the overall design of the proposed framework and shows how the security review process is organized into multiple agentic stages.

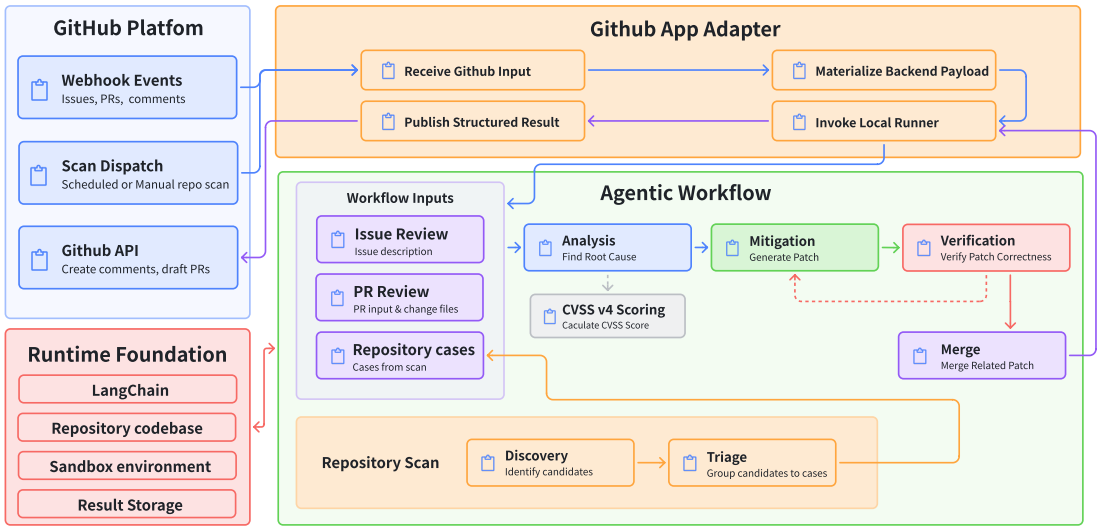


Figure 4.1: System overview of the proposed multi-stage agentic framework.

The system is organized into two decoupled layers: a platform adapter and a core agentic workflow. The adapter is responsible for receiving repository events, materializing the relevant context, and publishing workflow outputs. The core workflow performs the security-review logic and is independent of the specific repository hosting platform.

The workflow can be triggered by issue reports, pull request changes, or repository scan requests. In this context, an issue report is a developer-submitted description of a suspected vulnerability, while a pull request is a proposed code change represented by its changed files and related metadata. For issue- and pull-request-triggered review, the input already points to a concrete object under review. The workflow therefore starts from analysis, proceeds to mitigation when a repair is warranted, and ends with verification of the generated patch and remaining risks.

Repository review starts from the repository itself. It first performs discovery and triage: discovery collects candidate vulnerability evidence, and triage suppresses noise and groups related candidates into cases. These cases are not treated as confirmed vulnerabilities. Analysis then confirms or rejects each retained case by checking whether it is supported by repository evidence and whether it should proceed to mitigation. This is similar to traditional SAST and DAST outputs, which produce alerts or findings that still require confirmation before repair. After analysis, CVSS v4 scoring can run as a read-only prioritization branch, while mitigation and verification handle patch generation and patch audit. Only verified cases proceed to merge and delivery, where the workflow packages reports and patches into artifacts for human review.

4.2 Research Strategy

This thesis follows a design-and-evaluation research strategy. The design component defines a repository-grounded agentic workflow for web vulnerability review. The evaluation component compares this workflow against simpler alternatives and measures the tradeoffs between effectiveness, coverage, and cost under different model-allocation settings.

The research is organized around two scenarios. The first scenario is repair-task evaluation. In this setting, the vulnerability or repair task is already specified, and the main question is whether separating analysis, mitigation, and verification improves repair performance compared with a matched single-agent baseline. This scenario primarily addresses RQ1.

The second scenario is repository review. In this setting, the workflow cannot assume that a reliable vulnerability report already exists. It must first do discovery and triage, then use analysis to confirm or reject the retained cases before mitigation. This scenario addresses RQ2 by evaluating the value of the discovery and triage stages relative to traditional scanning tools, and RQ3 by examining how analysis, mitigation and verification outputs and final merge improve reviewability and delivery readiness for agent-generated repairs.

4.3 Stage Responsibilities and Artifacts

This section defines the responsibilities of each workflow stage and the artifacts passed to downstream stages.

4.3.1 Discovery

Discovery is the first repository-review stage. It is a file-level candidate generation stage: repository files are processed one by one, file content is presented individually, and the stage outputs zero or more candidates for each file. Its output is discovery candidates that contain title, category, confidence, source locations, and local evidence that may require further inspection.

Discovery is designed for coverage. It should identify potentially relevant clues quickly while avoiding expensive whole-repository reasoning at this early point. It is the stage most suitable for cost-efficient or faster model deployments.

4.3.2 Triage

Triage is the second repository-review stage. It takes discovery candidates and decides whether they should be kept as cases or suppressed, while also grouping related candidates when they should be represented as one case. It does not confirm vulnerabilities; its purpose is to reduce noise and form a smaller set of cases, which reduces the repository scope that must be processed by the downstream analysis stage.

The output of triage is a set of cases and suppressed candidates. Each retained case has a title, category, affected paths, key locations, and grouping rationale.

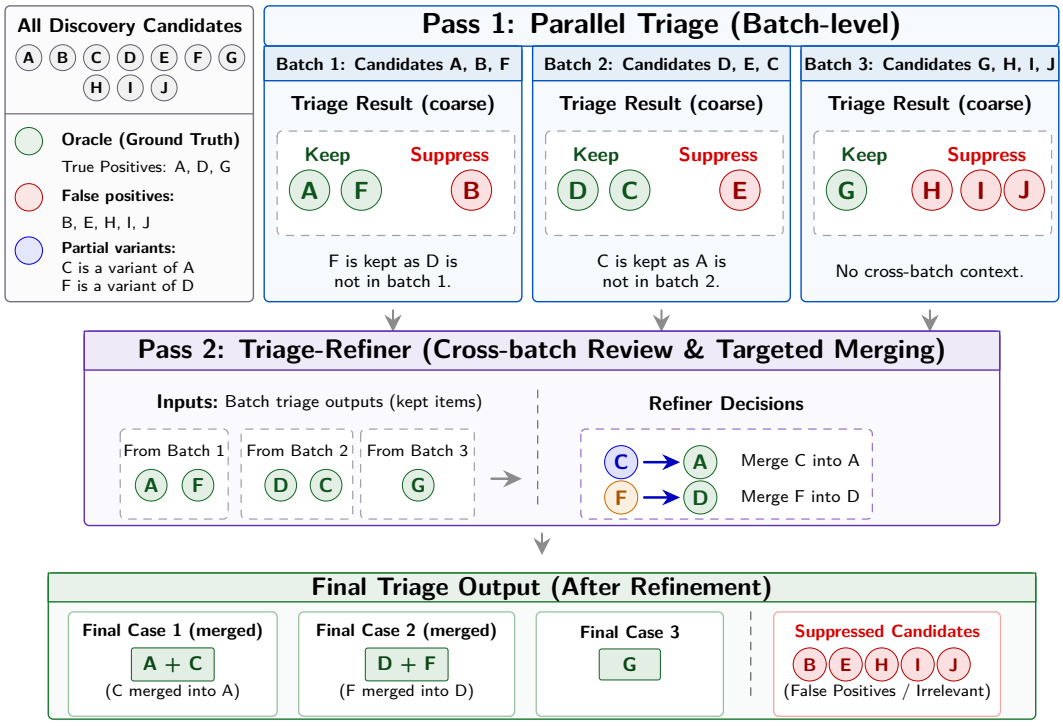


Figure 4.2: Two-pass repository triage.

In practice, triage may receive too many candidates to process reliably in a single

agent execution. A single large triage execution requires a larger prompt size and increases reasoning difficulty, and may lead to unstable grouping decisions or incomplete candidate coverage. A straightforward batching strategy reduces the cognitive load, but it can compromise cross-batch merging because related candidates may be separated early. To address these problems, our implementation uses the two-pass design shown in Figure 4.2. Discovery candidates are first triaged in parallel batches to produce coarse keep/suppress/group results. A subsequent triage-refiner stage then performs a second round of case-level refinement (limited to suppressions or merges) over the surviving batch-level cases. This design preserves system stability while recovering part of the cross-batch coherence lost by batching.

4.3.3 Analysis

Analysis turns a scoped input into a repository-grounded security judgement. In issue review, the input is the issue content; in pull-request review, it is the pull-request content together with the changed files; and in repository review, it is a triaged case.

Its first task is to determine whether the repository actually contains the vulnerability suggested by the input. If so, the analyzer formulates one or more evidence-grounded hypotheses that explain how the vulnerability arises, where it is located in the codebase, and what patch scope it implies. These are treated as hypotheses rather than final truths, as the current evidence may be incomplete, the root cause could be interpreted in different ways, and multiple interconnected bugs might exist simultaneously.

In all settings, the analyzer has access to the repository workspace. Unlike the discovery and triage stages, analysis is allowed to perform deeper repository-level reasoning. In particular, once a case has been formed, the analyzer can trace cross-file relationships such as call chains, data flow, routing behavior, and configuration-dependent control paths. This means that the discovery stage does not need to prove the full vulnerability within a single file. It only needs to collect enough local evidence for a suspicious case to be retained. If such evidence is present, the analyzer can reconstruct a complex, multi-file attack path that remains hidden when looking at individual files.

This design does leave a boundary case: some vulnerabilities may provide such a weak local signal for discovery even to identify any useful candidate at all. However, the framework treats this as an acceptable tradeoff. Allowing the analyzer to search the repository for possible vulnerabilities itself without the discovery and triage stages would greatly increase cost, reduce stability, and weaken inspectability. The workflow therefore prioritizes limited deep analysis over unconstrained whole-repository reasoning during discovery and triage.

The analyzer produces the main reasoning artifact of the workflow. Its output includes an overall verdict, confidence, residual risks, and a set of prioritized hypotheses. Each hypothesis is tied to concrete repository evidence such as locations, source facts, sink facts, supported inferences, and proof gaps. This structure driven

by hypotheses gives a clear target for downstream stages: the mitigator can focus on confirmed hypotheses, and the verifier can check whether the report overstated or underspecified the evidence.

4.3.4 CVSS v4 Scoring

CVSS v4 scoring is a post-analysis stage for confirmed repository cases. It follows the CVSS v4.0 specification and official calculator published by FIRST [11], [34]. It does not decide whether a vulnerability exists. Instead, it maps facts about the original analyzer report onto the standardized CVSS v4 base metrics, producing a vector, a base score, and a severity label. The score is used as a reporting and prioritization aid for human reviewers. It is not the main workflow gate for whether a patch should be produced or delivered.

CVSS v4 is used as a standardized technical severity measure so that the score can support reporting and prioritization of confirmed cases. Compared with application-specific approaches such as the OWASP Risk Rating Methodology [35], CVSS relies less on organization-specific assumptions about threat actors, asset value, and business impact. However, repository evidence may not fully capture deployment context, so the score may still involve contextual assumptions and is treated as a repository-grounded severity estimate rather than a complete business or operational risk assessment.

Compared with analysis or mitigation, this stage is intentionally lightweight. Once the analyzer has produced sufficiently explicit facts, CVSS scoring becomes a constrained interpretation task rather than an open-ended security reasoning task, because the CVSS v4 dimensions and scoring rules are predefined. In practice, the implementation uses the analyzer's output as its main input, and CVSS v4 scoring guidance and a small number of few-shot examples help keep the metric mapping consistent.

4.3.5 Mitigation

Mitigation starts from the analyzer output. It runs only when the analysis produces a reparable, validated hypothesis. The mitigator works in a writable workspace and attempts to produce a focused patch that addresses the scoped issue.

Quality in this stage is not just about the size of the patch. While the mitigator aims for small, practical edits, fixing the entire issue is more important than minimizing the number of lines changed. The goal is a precise and thorough fix, not just the smallest patch.

After producing the initial patch, the default mitigate agent performs a lightweight self-check. This check evaluates two key aspects: whether the vulnerability is successfully fixed, and whether any new regressions have been introduced. Although this self-check is not fully independent and inherently contains the mitigate agent's own bias, it provides an important early filter. Without it, even superficial issues in the patch would have to be caught by the verifier, which could consume more time

and token cost.

Despite the self-check, the verifier stage is still useful. The self-check may miss some issues due to its bias or limited perspective. The verifier acts as a useful secondary check to catch these remaining issues.

The mitigation artifact contains the mitigation status, changed files, a workspace patch, summaries of applied changes, and residual risks. The status may be `applied`, `partial`, or `skipped`. This status model is intentionally non-binary: when only partial repair is feasible, the stage may return a partial result together with explicit residual risks rather than overstating patch completeness.

4.3.6 Verification

Verification is an independent audit stage. It receives the analysis and mitigation report **summary** and the patch as input, and it produces an independent judgement about whether the vulnerability is completely fixed and no regressions are introduced.

The verifier receives a concise summary rather than the full analyzer report. Based on preliminary system trials, the full report made the verifier more likely to inherit the analyzer's framing instead of assessing the patch independently. However, giving the verifier no analyzer or mitigator context would force it to re-analyze the vulnerability from scratch. The summary therefore includes only objective findings and actions taken, while excluding subjective reasoning.

In patch audit, the verifier checks whether the mitigation covers that same review target with sufficient scope rather than only fixing a single instance of the vulnerability. It also examines similar behaviors, fallback paths, and related entry points when they remain relevant to the claim. Ideally, verification should gather evidence in two directions: whether the vulnerability path is now blocked, and whether legitimate or similar behavior is still preserved. In practice, this involves a mix of exploit simulation and regression testing.

4.3.7 Merge and Delivery

Merge and delivery are used only after repository cases have passed the workflow gate. This stage does not decide whether a case is valid; it decides how already-retained cases should be packaged for review. The implementation separates this responsibility into a `merge-planner`, which creates an exact-once delivery plan, and a `merge-executor`, which materializes that plan into delivery artifacts.

The merge-planner policy addresses a tension between patch atomicity and delivery efficiency. Patch atomicity favors small deliveries with a narrow repair intent, because this makes responsibility, testing, and rollback easier to understand. Delivery efficiency, however, favors grouping cases when separate deliveries would require reviewers to inspect the same files, code paths, or security control repeatedly. The planner therefore keeps cases separate by default and merges them only when there is a concrete patch-surface reason to do so, such as overlapping edits, shared helper

logic, or dependence on the same security control. Cases are not merged merely because they belong to the same vulnerability category or were discovered together.

After the plan is created, the executor synthesizes each delivery from a clean workspace. Single-case deliveries can be produced deterministically from their retained patches, while combined deliveries may use a delivery-level agent to reconcile overlapping edits into one coherent repair. If a combined delivery cannot be produced safely, the workflow falls back to smaller deliveries or marks the affected cases as blocked. The final result is a standardized, reviewer-oriented artifact that records the delivery strategy, changed files, final file snapshots, verification context, and residual risks.

4.4 Bounded Verifier Feedback

We introduce a bounded verifier-to-mitigator feedback mechanism that allows targeted patch revision without collapsing the multi-stage workflow into an open-ended agent loop. In a linear workflow (`analysis -> mitigation -> verification`), verification can identify patch defects but cannot trigger revision. This is limiting because many generated patches are partially correct but too narrow, local-only, misaligned with the target claim, or risky for nearby behavior.

The feedback path, shown in Figure 4.3, is triggered when the verifier reports patch-level problems such as incomplete coverage, unresolved vulnerability paths, regression risks, or mismatch with the scoped input. The retry does not reopen the analysis stage. Instead, the mitigator receives the verifier's report and the previous mitigation attempt, then performs a targeted revision.

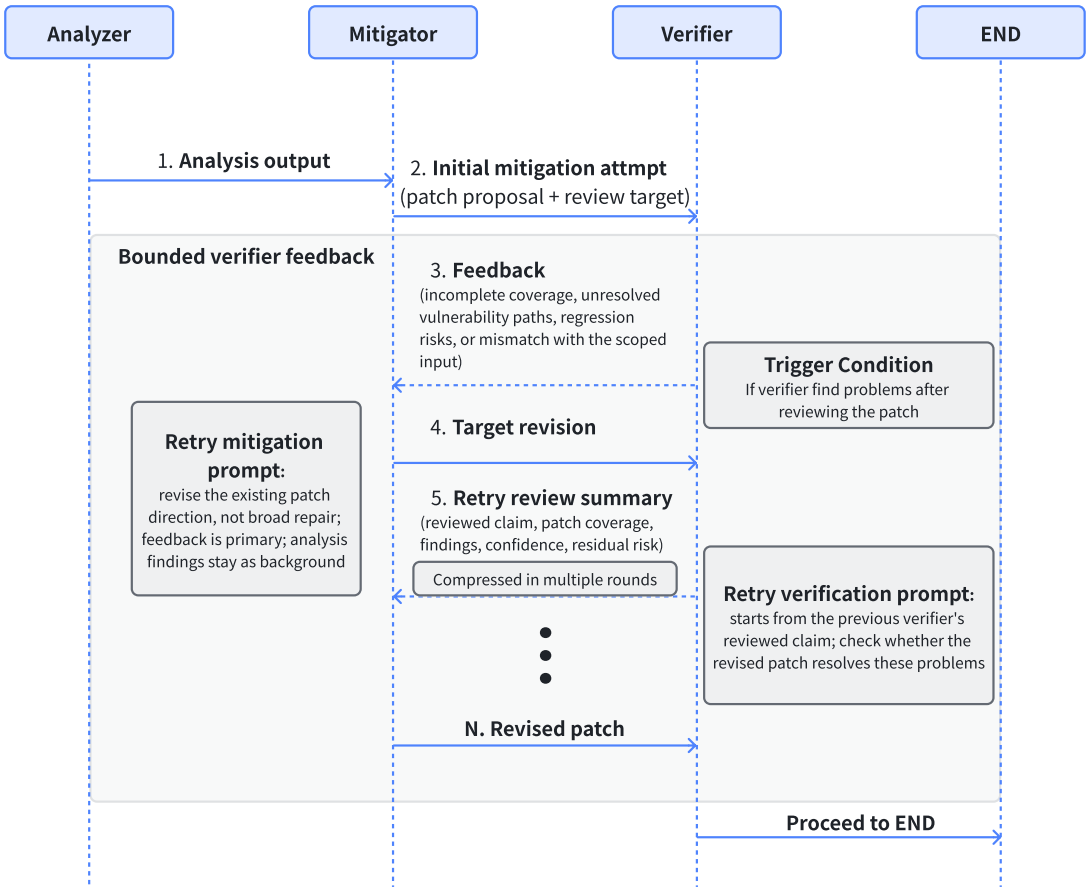


Figure 4.3: Bounded verifier-to-mitigator feedback loop.

The retry prompts are also different from the initial-stage prompts. On the mitigation side, the retry prompt treats verifier feedback as the primary corrective signal, while preserving the analyzer's valid findings as background context. It asks the mitigator to revise the existing patch direction rather than restart broad repair. On the verification side, the retry verifier starts from the previous verifier's reviewed claim, and asks whether the revised patch resolves the problems without introducing new regressions. This preserves continuity rather than reopening broad vulnerability analysis.

The retry context can also carry history across multiple feedback rounds, but this history is deliberately compressed. Instead of replaying full previous transcripts or complete stage outputs, the workflow passes forward concise verifier summaries such as the reviewed claim, patch coverage, findings, confidence, and residual risks. The prompt rendering further projects older history to the retry index, patch coverage, and key patch findings. This keeps later retries aware of the correction trajectory without allowing accumulated history to dominate the current patch review.

The mechanism is parameterized by a maximum retry count. In the current implementation and evaluation, this bound is set to one retry. A larger bound could support multiple verifier-mitigator feedback rounds, but additional retries should

be used cautiously: repeated failure is itself evidence that the repair target may be difficult, ambiguous, or poorly served by further local revision. The key constraint is therefore not only that the loop remains finite, but also that it remains a small, explicitly configured correction mechanism rather than an open-ended agent loop.

This design provides limited help when executable validation, such as proof-of-concept scripts or unit tests, is unavailable. If reliable tests are available, the mitigator can use them directly for revision. In realistic security-review settings, however, such validation is often missing or difficult to reproduce; bounded verifier feedback therefore provides a general fallback for improving patch quality.

4.5 Evidence- and Execution-Grounded Reasoning

The workflow is designed to move beyond prompt-only reasoning by grounding agent decisions in repository evidence and, when useful, lightweight execution. Across stages, agents are expected to inspect concrete materials such as source files, code changes, tests, configuration, and prior stage artifacts before forming conclusions. As a result, findings are tied to explicit repository locations and evidence items rather than free-form intuition.

Execution is used as a secondary grounding mechanism rather than as the default starting point. When command execution can significantly reduce uncertainty, agents may run focused checks such as lightweight reproduction commands or small validation scripts. These checks help confirm assumptions, expose false positives, and reduce the risk that a stage simply accepts its own earlier reasoning.

This design does not assume that runtime validation is always available. Many security-review tasks lack stable proof-of-concept scripts or directly runnable fix tests. The workflow therefore prioritizes repository-grounded reasoning first, and uses execution selectively when it provides meaningful additional evidence. In this sense, execution-guided validation complements evidence-grounded reasoning rather than replacing it.

4.6 Execution Model and Parallelism

Repository review is the workflow path where parallelism matters most. Compared with issue- or pull-request-triggered review, it involves a longer stage chain and a much larger number of work items, including many files during discovery and many cases after triage. Without some degree of concurrency, the end-to-end process would become unacceptably slow. For this reason, the implementation introduces parallelism at several points in this path.

Repository review contains four forms of parallelism. First, discovery can scan multiple files concurrently. Second, after triage, multiple cases can be processed concurrently because each case starts from the same clean workspace and writes to

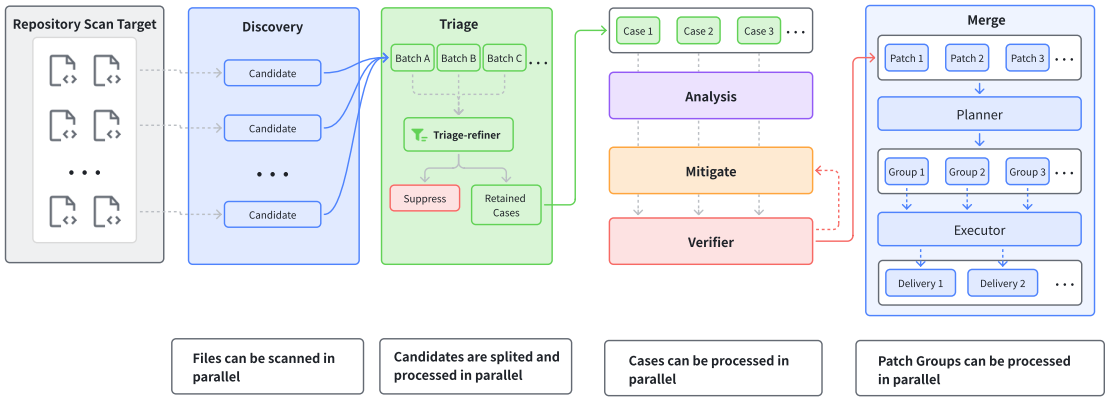


Figure 4.4: Parallel execution points in the repository review workflow.

separate artifacts. Third, CVSS v4 scoring is read-only and can run in parallel with the mitigation and verification path for the same case after analysis has completed. Fourth, after merge planning, independent deliveries can be synthesized concurrently by the merge executor.

Parallelism is bounded by runtime stability, local resource limits, and model-provider rate limits. Within a single case, the main dependency chain remains ordered: analysis precedes mitigation, and mitigation must produce a patch before full patch verification can be completed. If the verifier triggers the bounded retry path, the retry remains sequential inside that case. Similarly, each delivery is synthesized as a controlled unit even when multiple deliveries run at the same time. Parallelism is therefore used only where work items are independent or where a stage is read-only with respect to the patching path.

4.7 Implementation and Execution Flow

This section describes how the multi-stage workflow is operationalized in the implementation, focusing on workspace isolation, artifact persistence, and execution order.

4.7.1 Workspace and Execution Environment

Most stages operate on a unified local repository clone rather than isolated snippets, providing agents with full access to source files and prior stage outputs. Analysis-centric stages are restricted to read-only views, while the mitigation stage is granted a writable workspace for patch generation.

In repository reviews, each case is isolated within a clean baseline workspace. This prevents cross-case interference and ensures that patches remain independent until the final delivery stage, while also simplifying the inspection of failed tasks. For further security, an optional Docker sandbox can be enabled to explicitly constrain command execution, resource usage, and network access, providing a strictly controlled environment for agent operations.

4.7.2 Artifacts and Auditability

Each stage saves its result in a structured form, including stage outputs, workspace patches, messages, checkpoints, and final workflow summaries. These saved results make it possible to inspect what the workflow produced at each step instead of relying only on free-form model explanations.

This is important to the multi-stage design because later stages read earlier saved outputs rather than relying only on conversational memory. Discovery results, triage cases, analyzer hypotheses, mitigation patches, verification judgments, merge decisions, and delivery manifests are all preserved as explicit artifacts. As a result, runs are easier to inspect, review, and audit after execution.

5

Evaluation

This chapter presents the evaluation setup and results. The evaluation is organized into two settings. The first is an issue-driven benchmark comparison. The second is a repository case-study comparison.

5.1 Evaluation Setup

The evaluation is organized around two kinds of settings. The first is an issue-driven benchmark setting, where the input already specifies a vulnerability description and the task is to localize and repair it within a repository. The second is a repository case-study setting, where the system must first discover and triage before deeper analysis and mitigation can begin. These two settings correspond to different parts of the workflow and are therefore evaluated separately.

Figure 5.1 shows how the evaluation settings are used to answer the research questions.

5.1.1 Issue Scenario Evaluation

Issue scenario evaluation starts with a given vulnerability description and access to the repository. This makes it possible to compare the multi-stage agentic workflow (`analysis -> mitigation -> verification` + bounded feedback) against representative single agent.

PatchEval [36] is used as the main benchmark framework for this setting. It is a benchmark for automated vulnerability repair that provides 1,000 real-world vulnerabilities from CVEs reported between 2015 and 2025, covering 65 CWE categories across Go, JavaScript, and Python. PatchEval also evaluates multiple agent and model configurations under standardized repair settings, which makes it a suitable experimental basis for the issue-driven comparison in this thesis.

PatchEval is selected because it matches the issue-driven automated vulnerability repair task studied in RQ1. General APR benchmarks such as SWE-bench [37] focus on software issue resolution rather than vulnerability-specific repair, while AVR benchmarks such as VUL4J [38] are limited to Java and provide little coverage of web vulnerability repair. The issue-driven experiment is therefore limited to PatchEval to keep the task formulation, oracle, and comparison protocol consistent.

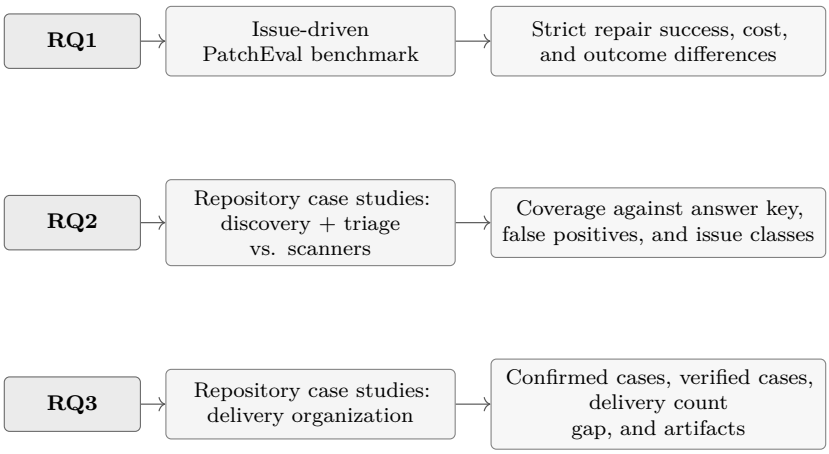


Figure 5.1: Evaluation design used to answer the research questions.

In addition, many AVR benchmarks are derived from public CVE datasets, which can lead to overlapping vulnerability sources rather than fully independent test distributions. PatchEval is used as a broad, task-aligned benchmark, and the results should be interpreted as benchmark-specific evidence rather than proof of general performance across all vulnerability-repair benchmarks.

A subset of 230 cases includes Docker-based runtime sandboxes with at least proof-of-concept (PoC) tests. In this study, we use that runtime-validated subset and follow PatchEval's task formulation: the system receives the vulnerability description and repository access but does not receive feedback from unit tests or PoC execution during generation.

Within that formulation, PatchEval distinguishes between two settings:

- **End-to-End Patch Generation**: the system must localize and repair the vulnerability from the description and repository alone.
- **Patch Generation with Location Oracle**: the system also receives vulnerable-location guidance in addition to the vulnerability description.

These two settings are the most relevant external references for this thesis because they match the issue-driven task family while separating the effect of localization assistance from the broader end-to-end repair problem.

The experiment uses the 230 PatchEval runtime-validated subset and compares the following four agent workflows under the same model setting. For compact reporting, the short names below are used throughout the Results and Discussion chapters.

- Single-Agent: a single agent receives the same input and performs analysis and repair within one agent.
- Default: the full multi-stage workflow, `analysis` → `mitigation` → `verification`, with one bounded verifier-to-mitigator retry.
- No-Verifier: a two-stage workflow, `analysis` → `mitigation`, without independent verification or feedback retry. It is measured using the initial miti-

gation output of the DEFAULT run before retry.

- NO-VERIFIER+DEEP-SELF-CHECK: the same two-stage workflow as NO-VERIFIER, but the mitigator is instructed to perform an additional self-audit of the final patch before finishing. It is measured as a separate analyzer–mitigator run with the stronger mitigation self-check prompt.

The SINGLE-AGENT baseline versus DEFAULT provides the most direct contrast between multi-stage and single-agent designs. NO-VERIFIER removes verification and retry from DEFAULT, while NO-VERIFIER+DEEP-SELF-CHECK tests whether a mitigator-side self-audit can replace part of the benefit of an independent verifier. In this self-audit, the mitigator reviews its own patch against the repair target, checks for incomplete coverage or regressions, and either revises the patch or records remaining uncertainty. Because the task set, repository access, and model family are fixed, the comparison focuses on the effect of workflow design rather than differences between models.

5.1.2 Repository Case-Study Evaluation

Repository evaluation starts only from repository access. The evaluated path is `discovery` → `triage` → `analysis` → `cvss-v4-scoring` → `mitigation` → `verification` → `merge-planner` → `merge-executor`, with CVSS v4 scoring treated as a post-analysis reporting branch.

This setting is evaluated through three repository case studies: Book Shop [39], Raddit [40], and WorldPress [41]. Together, these benchmarks cover SQL injection, path traversal, arbitrary upload, command injection, server-side request forgery (SSRF), authorization flaws, JSON Web Token (JWT) design or validation flaws, sensitive-data exposure, cross-site scripting (XSS), cross-site request forgery (CSRF), parser vulnerabilities, and missing preventive controls. Table 5.1 summarizes the repository stacks and evaluation reference sizes.

Table 5.1: Repository benchmark overview.

Repository	Main stack	Answer-key items	Ideal deliveries
Book Shop	Next.js / Prisma	19	9
Raddit	Go / React	22	9
WorldPress	Flask / Vue	21	10

Note. Answer-key items are manually injected benchmark vulnerabilities. Ideal deliveries are reviewer-oriented repair groups defined for each repository.

For each repository, we construct a ground-truth reference package consisting of two components: a Vulnerability Answer Key Document and a set of ideal deliveries. Together, these two artifacts serve as the standard answer against which generated review outputs are evaluated.

The Vulnerability Answer Key Document is the vulnerability-level ground truth. It enumerates the confirmed vulnerabilities in the repository and records, for each

vulnerability, its vulnerability type, affected path or code location, trigger input or condition, and validation evidence. This document defines what vulnerabilities exist in the repository and provides the evidence used to judge whether a generated review has correctly identified and explained them.

The ideal deliveries define the delivery-level ground truth. Each ideal delivery specifies a reviewer-oriented repair boundary for evaluation: it groups one or more answer-key vulnerabilities into a single coherent delivery that can be assessed largely independently. Vulnerabilities are grouped only when doing so improves reviewability, for example because the repairs share a code path, trust boundary, authentication/session flow, rendering or sanitization strategy, or another cross-cutting control. In practice, an ideal delivery often corresponds to one self-contained patch, but the definition is intentionally tool-agnostic and does not assume any specific code-review or version-control mechanism.

The first comparison in this setting addresses RQ2. Traditional scanning tools' output (Semgrep [42], CodeQL [43], and OWASP ZAP [44]) is compared with the `discovery + triage` workflow. The purpose is to test whether discovery and triage stages find more true vulnerabilities, suppress more false leads, or expose complementary findings that traditional scanning tools do not produce.

We evaluate detection against a Vulnerability Answer Key Document. Case-level recall counts an answer-key vulnerability as covered when the workflow keeps a case for the same vulnerable surface after the discovery and triage stages. For scanner baselines, the same measure is computed over alerts or findings.

For agentic runs, we also track stage-level funnel counts: discovery candidates, triaged cases, analyzer-confirmed cases, verified cases, and final deliveries. These counts describe how review outputs move through the workflow and support the delivery-readiness analysis.

Manual review records confirmed false positives after duplicate grouping. Duplicate, overlapping, defense-in-depth, non-security, deployment-hardening-only, and non-covering related observations are recorded as review load or related signals, but are not counted as confirmed false positives.

The second comparison in this setting addresses RQ3. It evaluates whether stage-level reports and the merge-planner and merge-executor improve reviewability and delivery readiness by producing coherent, scoped artifacts for human review.

We evaluate delivery organization by comparing the manually defined ideal delivery count with the number of deliveries produced by the workflow. We use this comparison as a proxy for reviewer burden: when semantically related cases are split into too many deliveries, reviewers must inspect more items and manually reconstruct cross-case relationships. The main numeric measure is the delivery count gap:

$$\text{delivery count gap} = \text{actual deliveries} - \text{ideal deliveries}.$$

A large positive gap indicates fragmented repair packaging and increased review-organization overhead. A negative gap indicates fewer deliveries than ideal, often

because unrelated repairs were bundled together or because expected repairs were missed. A small or zero gap is not sufficient evidence of good delivery organization, because it can also result from missed cases or overly broad bundling. Therefore, delivery count gap is interpreted together with answer-key coverage and manual review of delivery contents.

5.1.3 Reproducibility Details

The issue-review runs used `deepseek-v4-pro`. The repository case-study runs used `mimo-v2.5-pro`, `minimax-m2.7`, and `doubao-seed-2.0-lite`. In the implementation, these models are routed through an Anthropic-compatible chat interface.

The evaluations were run in May 2026. We could not determine more specific version-pinned model IDs, such as `doubao-seed-2-0-lite-260215`, because the provider did not expose this information.

All chat model calls used temperature 0.0. No explicit `top_p` or random seed was configured, so any remaining decoding or backend nondeterminism is provider-side behavior.

For scanner baselines, CodeQL results were taken from GitHub code-scanning alerts, and Semgrep results were taken from Semgrep Platform Code findings. ZAP combines a frontend passive scan with an authenticated OpenAPI-seeded backend/API scan; the reported ZAP numbers use manually reviewed cleaned alert groups.

The repository answer keys and ideal delivery groups described above were finalized before evaluating agent outputs. The subsequent mapping from agent and scanner outputs to answer-key items was manually checked.

5.2 Issue-Review Benchmark Results

We evaluate the proposed workflow on the PatchEval [36] 230-task subset with container image and PoC tests under a strict criterion, where a repair is counted as successful only if it passes both PoC and unit-test validation. To isolate the effect of workflow design, we keep the model fixed as Deepseek V4 Pro across all workflow variants, so that performance differences can be attributed to the workflow rather than to model choice.

This section is organized as follows. We first provide external PatchEval reference results for comparison. We then present the results of all workflow variants in both the end-to-end and location-oracle settings. Finally, we examine the end-to-end success-set overlap between Default and Single-Agent to show the difference at the case level.

5.2.1 State-of-the-art Performance

Table 5.2 places the proposed workflow result alongside the strongest and most relevant external PatchEval reference results reported by the benchmark leaderboard.

In this table, *System* refers to the repair setup that performs the task. Some rows use a general-purpose coding agent, such as Claude-Code, SWE-Agent, or OpenHands, with the PatchEval task prompt; the GPT-5 row uses GPT-5 directly with a customized prompt. The DEFAULT row refers to the proposed multi-stage workflow defined in the evaluation setup. *Model* refers to the LLM used by each setup, such as Gemini 2.5 Pro, GPT-5, or Deepseek V4 Pro.

The external systems are not controlled baselines for this study because the agent framework, model family, or both differ from the fixed-model workflow variants evaluated later. Therefore, they are reported only as references rather than used for workflow attribution [36], [45].

Table 5.2: PatchEval issue-review results compared with external references.

System	**Model**	**Setting**	**Overall**	**Tokens**	**Price**
Claude-Code	Gemini 2.5 Pro	End-to-end	12.61% (29/230)	2492.83	6.401
SWE-Agent	Gemini 2.5 Pro	End-to-end	13.91% (32/230)	1190.35	2.387
OpenHands	Gemini 2.5 Pro	End-to-end	18.70% (43/230)	1817.37	4.640
DEFAULT (ours)	Deepseek V4 Pro	End-to-end	**33.91% (78/230)**	1938.38	**0.216**
GPT-5	GPT-5	Location-oracle	29.1% (67/230)	–	–
SWE-agent	GPT-5	Location-oracle	33.0% (76/230)	–	–
ClaudeCode	GPT-5	Location-oracle	34.4% (79/230)	–	–
OpenHands	GPT-5	Location-oracle	**36.1% (83/230)**	–	–
Claude-Code	Gemini 2.5 Pro	Location-oracle	18.70% (43/230)	333.06	0.853
OpenHands	Gemini 2.5 Pro	Location-oracle	21.30% (49/230)	1141.62	2.933
SWE-Agent	Gemini 2.5 Pro	Location-oracle	23.04% (53/230)	561.07	1.128
DEFAULT (ours)	Deepseek V4 Pro	Location-oracle	33.91% (78/230)	1430.24	**0.181**

Note. Average token usage is reported in thousands, and average price is reported in USD when available. Bold indicates the best reported overall result within a setting, or the lowest reported price when price is available.

For the listed Gemini 2.5 Pro-based agent references, end-to-end results range from 29/230 to 43/230 successful repairs, while location-oracle results range from 43/230 to 53/230, corresponding to gains of 6 to 21 repairs. The DEFAULT result of 78/230 is above these Gemini 2.5 Pro references in both settings, but below the strongest GPT-5 location-oracle references listed in the table. Within our own workflow, DEFAULT shows no success-count difference between the end-to-end and location-oracle configurations.

5.2.2 Fixed-Model Workflow Variant Results

Table 5.3 reports the absolute results for the workflow variants defined in the evaluation setup.

Table 5.3: Absolute issue-review results by workflow variant with Deepseek V4 Pro fixed as the model.

Variant	Setting	Overall	Tokens	Price
Single-Agent	End-to-end	31.74% (73/230)	1619.09	0.142768
No-Verifier+Deep-Self-Check	End-to-end	31.74% (73/230)	1632.86	0.180573
No-Verifier	End-to-end	33.04% (76/230)	1618.15	0.177376
Default	End-to-end	**33.91% (78/230)**	1938.38	0.215955
Single-Agent	Location-oracle	31.74% (73/230)	963.37	0.110590
No-Verifier+Deep-Self-Check	Location-oracle	32.61% (75/230)	1248.71	0.153615
No-Verifier	Location-oracle	33.48% (77/230)	1190.65	0.149150
Default	Location-oracle	**33.91% (78/230)**	1430.24	0.180557

Note. Average token usage is reported in thousands, and average price is reported in USD when available.

Table 5.4: Workflow-variant deltas relative to the matched Single-Agent baseline with Deepseek V4 Pro.

Variant	Setting	Overall	Δ Overall	Δ Price
Single-Agent	End-to-end	31.74%	–	–
No-Verifier+Deep-Self-Check	End-to-end	31.74%	+0.00 pp	+26.48%
No-Verifier	End-to-end	33.04%	+1.30 pp	+24.24%
Default	End-to-end	**33.91%**	**+2.17 pp**	+51.26%
Single-Agent	Location-oracle	31.74%	–	–
No-Verifier+Deep-Self-Check	Location-oracle	32.61%	+0.87 pp	+38.90%
No-Verifier	Location-oracle	33.48%	+1.74 pp	+34.87%
Default	Location-oracle	**33.91%**	**+2.17 pp**	+63.27%

Note. For each row, Δ values are computed against the Single-Agent row with the same setting. Accuracy changes are reported in percentage points; price changes are reported as relative changes.

Tables 5.3 and 5.4 show that Default has the highest Strict success count among the workflow variants in both settings, with 78/230 successful repairs. Relative to Single-Agent, none of the evaluated workflow variants has a lower success count.

Default achieves the largest improvement, at +2.17 percentage points in both settings, corresponding to 5 additional successful repairs. The two no-verifier variants remain close: No-Verifier improves over Single-Agent by +1.30 percentage points end-to-end and +1.74 percentage points with the location oracle, while No-Verifier+Deep-Self-Check matches Single-Agent end-to-end and improves by +0.87 percentage points with the location oracle.

The success counts show small differences between end-to-end and location-oracle settings, but the location oracle reduces cost and token use. Averaged across the four workflow variants, token use decreases from 1702.12k to 1208.24k tokens, a reduction of 493.88k tokens or 29.0%. Average price decreases from 0.179168 USD to 0.148478 USD, a reduction of 17.1%. Default and Single-Agent have identical success counts across the two settings, while No-Verifier differs by 1 successful repair and No-Verifier+Deep-Self-Check differs by 2. All workflow variants have higher average price than Single-Agent in both settings. Among the workflow variants, No-Verifier has the lowest average price in both settings, while Default has the highest average token use and average price.

5.2.3 Default vs. Single-Agent Outcome Differences

The aggregate end-to-end success counts show the overall gap between Default and Single-Agent, but not how their successful cases overlap. Table 5.5 reports the overlap and differences between their Strict success sets on the full PatchEval 230-task subset.

Table 5.5: End-to-end success-set overlap between Default and Single-Agent on the 230-task subset.

	Single-Agent **pass**	Single-Agent **fail**
Default **pass**	66	12
Default **fail**	7	145

The off-diagonal cases show that Default and Single-Agent succeed on partly different tasks. The case-level difference sets, including the separately marked runtime-oracle outlier, are reported in Appendix A.3, and Table 5.6 lists one representative case from each off-diagonal direction.

Table 5.6: Representative off-diagonal cases in the end-to-end success-set comparison.

Aspect	`CVE-2021-21360`	`CVE-2023-50726`
Outcome	DEFAULT pass; SINGLE-AGENT fail.	SINGLE-AGENT pass; DEFAULT fail.
Vulnerability	Information disclosure in Generic Setup snapshot objects.	Authorization bypass in application creation with local manifests.
Oracle check	Allowed roles of generated snapshot folders and files.	Created application must have no retained operation.

Note. The examples are selected to illustrate repair-pattern differences rather than differences by CVE year.

The two examples show differences in the vulnerability-analysis conclusions and repair directions selected with and without the analyzer stage.

For `CVE-2021-21360` [46], the relevant creation paths are `writeDataFile()` for snapshot files and `_ensureSnapshotsFolder()` for snapshot folders. The SINGLE-AGENT leaves files created by `writeDataFile()` unchanged, and assigns only the `Manager` role to the folder. By contrast, **all** other workflow variants repair both paths by setting explicit `View` permissions on snapshot folders and snapshot files for `Manager` and `Owner`.

For `CVE-2023-50726` [47], the vulnerable endpoint is `Create()`, which accepts an `Application` object that may contain `Operation.Sync.Manifests`. The SINGLE-AGENT patch directly sets `a.Operation = nil` during application creation, so any operation supplied in the creation request is removed before the application is stored. By contrast, **all** other workflow variants select a permission-check repair and add an `ActionOverride` check for local-manifest sync rather than setting `a.Operation` to `nil`. This enforces the override-permission condition, but the oracle checks for a `nil` operation in the created application.

5.2.4 Verifier Retry Outcomes

We collect retry statistics from DEFAULT. A retry is triggered when the verifier returns feedback to the mitigator. We record how many cases trigger retry, how many triggered cases pass final validation, and how many cases are converted from an initial failure to a final success, as summarized in Table 5.7.

Table 5.7: Verifier retry outcomes in Default.

Outcome group	Count
Total evaluated cases	230
Retry not triggered	184
Retry triggered by verifier	46
Retry: Fail → Pass	3
Retry: Pass → Pass	6
Retry: Pass → Fail	1
Retry: Fail → Fail	36
Retry-triggered final pass	9
Retry-triggered final fail	37

All triggered cases used one retry attempt. Across these triggered cases, the verifier output records residual paths or bypasses without access to the hidden benchmark oracle. The resulting retry maps to four observed outcome transitions: Fail-to-Pass, Pass-to-Pass, Pass-to-Fail, and Fail-to-Fail.

Table 5.8 lists only the cases where the outcome differs between the initial mitigation output and the final DEFAULT output. The corresponding vulnerability records are `CVE-2015-1326` [48], `CVE-2020-15084` [49], `CVE-2021-21321` [50], and `CVE-2022-29822` [51].

Table 5.8: Cases where verifier retry changes the outcome.

Case	Outcome change	Verifier feedback	Retry modification
`CVE-2015-1326`	Fail → Pass	`module.__file__` check misses malicious cached `.pyc` loads that still report the source `.py` path.	Replaced `importlib.import_module()` with explicit source reading, `compile()`, and `exec()` through `types.ModuleType`.
`CVE-2020-15084`	Fail → Pass	Defaulting JWT algorithms to `HS256` creates an RSA-public-key-as-HMAC-secret confusion path.	Changed behavior to reject configuration without an explicit `algorithms` setting.
`CVE-2021-21321`	Fail → Pass	URL-encoded path separators such as `%2f` survive WHATWG URL parser path normalization.	Added multi-pass percent decoding, re-resolution, and a stricter base-path prefix check.
`CVE-2022-29822`	Pass → Fail	One `_get` path can overwrite sanitized `[Op.and]` data with the original unsanitized query object.	Moved validation into `filterQuery` and added broader query-option sanitization.

5.2.5 Mitigator Self-Check Outcomes

We next compare NO-VERIFIER and NO-VERIFIER+DEEP-SELF-CHECK at the case level. Both variants remove the verifier and retry mechanism, following the definitions in the evaluation setup. We report the overlap of their strict success sets.

Table 5.9: Success-set overlap between no-verifier variants.

	Deep-Self-Check pass	Deep-Self-Check fail
No-Verifier pass	63	13
No-Verifier fail	10	144

The off-diagonal cases show that the two no-verifier variants also succeed on partly different tasks. The complete strict-outcome difference sets are reported in Appendix A.3, and Table 5.10 lists one representative case from each off-diagonal direction.

Table 5.10: Representative off-diagonal cases in the mitigator self-check comparison.

Aspect	`CVE-2021-32796`	`CVE-2024-49750`
Outcome	No-Verifier pass; No-Verifier+Deep-Self-Check fail.	No-Verifier+Deep-Self-Check pass; No-Verifier fail.
Vulnerability	XML serialization output escaping in `xmldom`.	Sensitive credential exposure in Snowflake Connector Python logging.
Oracle check	Synthesized `xmlns` namespace URI values must be XML-escaped during serialization.	Sensitive tokens and credentials must be masked in the checked logging/redaction paths.

For `CVE-2021-32796` [52], both no-verifier variants edit `lib/dom.js`, but they modify different serialization paths. The analyzer identifies unescaped namespace URI values in synthesized `xmlns` declarations. The No-Verifier patch changes the two `buf.push(ns, '="', uri, '"')` paths to encode `uri` with `_xmlEncoder`. The No-Verifier+Deep-Self-Check patch instead changes the regexes used for ordinary attribute-value and text-node escaping by adding `>` to those character classes, but does not modify the synthesized `xmlns` namespace URI emission paths. Under the strict oracle, the first patch passes and the second patch fails.

For `CVE-2024-49750` [53], both no-verifier variants modify the authentication debug-log filter, the secret detector, and storage-client retry logging. The No-Verifier patch updates `PASSCODE` filtering, several `SecretDetector` patterns, and Azure SAS-token masking in storage retry logs. The No-Verifier+Deep-Self-Check patch additionally masks URLs in `network.py` error logging and several vendored `urllib3` connection-pool debug or warning logs. Under the strict oracle, the narrower No-Verifier patch fails and the broader No-Verifier+Deep-Self-Check patch passes.

5.3 Repository Case-Study Results

This section reports results for the repository case-study setting defined in Section 5.1.2. Unlike the file- or task-level experiments, this setting measures whether a workflow can recover a benchmark answer key from an entire codebase, triage findings into security cases, and package generated repairs into reviewable deliveries.

5.3.1 Vulnerability Coverage Compared with Traditional Scanners

Table 5.11 summarizes repository detection coverage on the three case-study repositories. Coverage is measured at case level for the proposed repository-review workflow and at alert/finding level for the scanner baselines. For the workflow, an answer-key item is counted as covered when discovery and triage retain a case for the same vulnerable surface.

Table 5.11: Repository detection coverage across the three benchmark repositories.

Run/tool	Outputs	Answer-key recall	Review notes
Book Shop			
Ours (mimo)	42 cases	19/19 (100.0%)	–
Ours (MiniMax)	71 cases	17/19 (89.5%)	–
Ours (Doubao)	75 cases	18/19 (94.7%)	–
CodeQL	4 alerts	1/19 (5.3%)	–
Semgrep Code	12 findings	3/19 (15.8%)	–
ZAP	17 groups	1/19 (5.3%)	–
Raddit			
Ours (mimo)	45 cases	21/22 (95.5%)	–
Ours (MiniMax)	70 cases	21/22 (95.5%)	1 FP
Ours (Doubao)	43 cases	15/22 (68.2%)	3 FP
CodeQL	11 alerts	7/22 (31.8%)	–
Semgrep Code	10 findings	8/22 (36.4%)	–
ZAP	16 groups	3/22 (13.6%)	1 FP
WorldPress			
Ours (mimo)	39 cases	20/21 (95.2%)	–
Ours (MiniMax)	52 cases	20/21 (95.2%)	–
Ours (Doubao)	33 cases	17/21 (81.0%)	–
CodeQL	20 alerts	9/21 (42.9%)	–
Semgrep Code	25 findings	8/21 (38.1%)	–
ZAP	12 groups	2/21 (9.5%)	–

Note. Review notes list only confirmed false positives after duplicate grouping; duplicate, non-security, frontend-only, product-defect, and non-covering related observations are not marked as false positives. WorldPress Semgrep findings are counted with a CSV parser because quoted fields contain embedded newlines.

The scanner baselines produced smaller reviewed finding sets and lower answer-key recall than the agentic runs. Their coverage was concentrated in localized or externally observable issue classes, while the agentic runs also covered repository-level logic and design flaws. After duplicate grouping, confirmed false positives were concentrated in Raddit ZAP and the more permissive Raddit agentic runs.

Figure 5.2 shows the same pattern by issue class. The main shared false negative was missing login rate limiting, which all three agentic runs, CodeQL, Semgrep, and ZAP missed.

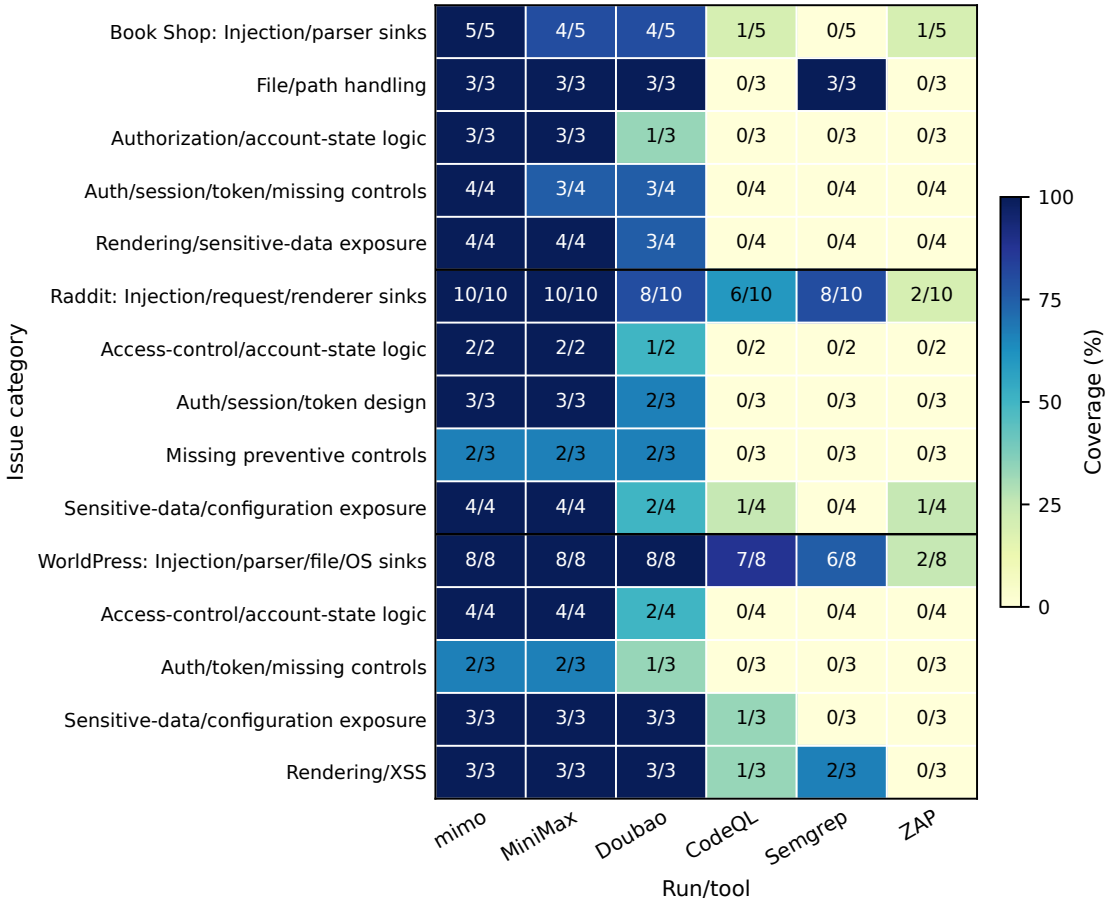


Figure 5.2: Answer-key coverage heatmap by vulnerability category across repository case studies. Cell color denotes the percentage of answer-key items covered within each repository-specific category; cell text reports covered items over category total.

5.3.2 Reviewability and Delivery Organization

Table 5.12 summarizes repository review and delivery results for each workflow run. These counts are diagnostic rather than standalone evaluation metrics: they show how candidates become cases, how analyzer judgment filters or classifies those cases, how many cases survive mitigation and verification, and how final merge packaging turns verified cases into deliveries. The rightmost column reports the delivery count gap defined in the methodology.

Table 5.12: Repository review and delivery results.

Run	Candidates	Cases	Confirmed	Verified	Deliveries	Gap
Book Shop (9 ideal deliveries)						
Ours (mimo)	119	42	40	39	22 (10 combined + 12 single)	+13
Ours (MiniMax)	136	71	61	61	53 (5 combined + 48 single)	+44
Ours (Doubao)	103	75	74	73	44 (11 combined + 33 single)	+35
Raddit (9 ideal deliveries)						
Ours (mimo)	87	45	35	34	14 (7 combined + 7 single)	+5
Ours (MiniMax)	105	70	62	60	23 (15 combined + 8 single)	+14
Ours (Doubao)	49	43	43	43	43 (0 combined + 43 single)	+34
WorldPress (10 ideal deliveries)						
Ours (mimo)	85	39	36	36	20 (9 combined + 11 single)	+10
Ours (MiniMax)	94	52	49	49	49 (0 combined + 49 single)	+39
Ours (Doubao)	60	33	33	33	10 (7 combined + 3 single)	0

Note. Ours denotes the proposed repository-review workflow with the model named in parentheses: mimo is mimo-v2.5-pro, MiniMax is MiniMax M2.7, and Doubao is Doubao Seed 2.0 Lite. Funnel columns and delivery count gap follow the definitions in Section 5.1.2.

For example, in the Book Shop workflow run with Doubao Seed 2.0 Lite, discovery produced 103 candidates. Triage kept and merged candidates, finally producing 75 cases; the analyzer confirmed 74 of them, and 73 cases survived mitigation and verification. The merge stage then packaged these 73 verified cases into 44 reviewable deliveries, consisting of 11 combined deliveries and 33 single-case deliveries.

The largest positive gaps were observed with MiniMax M2.7 on Book Shop (+44) and WorldPress (+39), followed by Doubao Seed 2.0 Lite on Book Shop (+35) and Raddit (+34). With Doubao Seed 2.0 Lite, the workflow had a zero delivery count gap on WorldPress.

Table 5.13 reports token and cost diagnostics for the same agentic runs.

Table 5.13: Agentic repository-review token and cost diagnostics.

Run	Input tokens	Output tokens	Cache-read tokens	Observed cost
Book Shop				
Ours (mimo)	27.0M	926.6K	24.9M	$9.835646
Ours (MiniMax)	24.0M	907.0K	0	$8.280616
Ours (Doubao)	51.7M	705.5K	26.9M	$3.088899
Raddit				
Ours (mimo)	30.8M	867.6K	28.4M	$10.691960
Ours (MiniMax)	36.2M	1.1M	0	$12.127009
Ours (Doubao)	37.9M	503.2K	17.5M	$2.416956
WorldPress				
Ours (mimo)	21.7M	1.0M	19.8M	$8.891914
Ours (MiniMax)	18.1M	706.3K	0	$6.277521
Ours (Doubao)	18.8M	303.6K	8.9M	$1.216244

Note. The MiniMax runs report zero cache-read tokens because the provider-reported artifacts did not expose cache-hit usage; this is a provider-side reporting limitation rather than a workflow setting that can be corrected in our implementation.

Token counts are total workflow usage; cache-read tokens are reported separately because cached input is priced differently by the workflow pricing profile. Observed costs are computed from provider-reported token usage and do not include human review effort outside the agent run.

Figures 5.3 and 5.4 show one generated delivery artifact from the Raddit workflow run with mimo-v2.5-pro. The overview page groups publishable deliveries by CVSS v4 severity and reports the number of cases in each delivery. The delivery page records the changed files and expandable per-case reports. The expanded case view includes metadata such as category, analyzer verdict, CVSS v4 score, CVSS scoring rationale, and verification information.

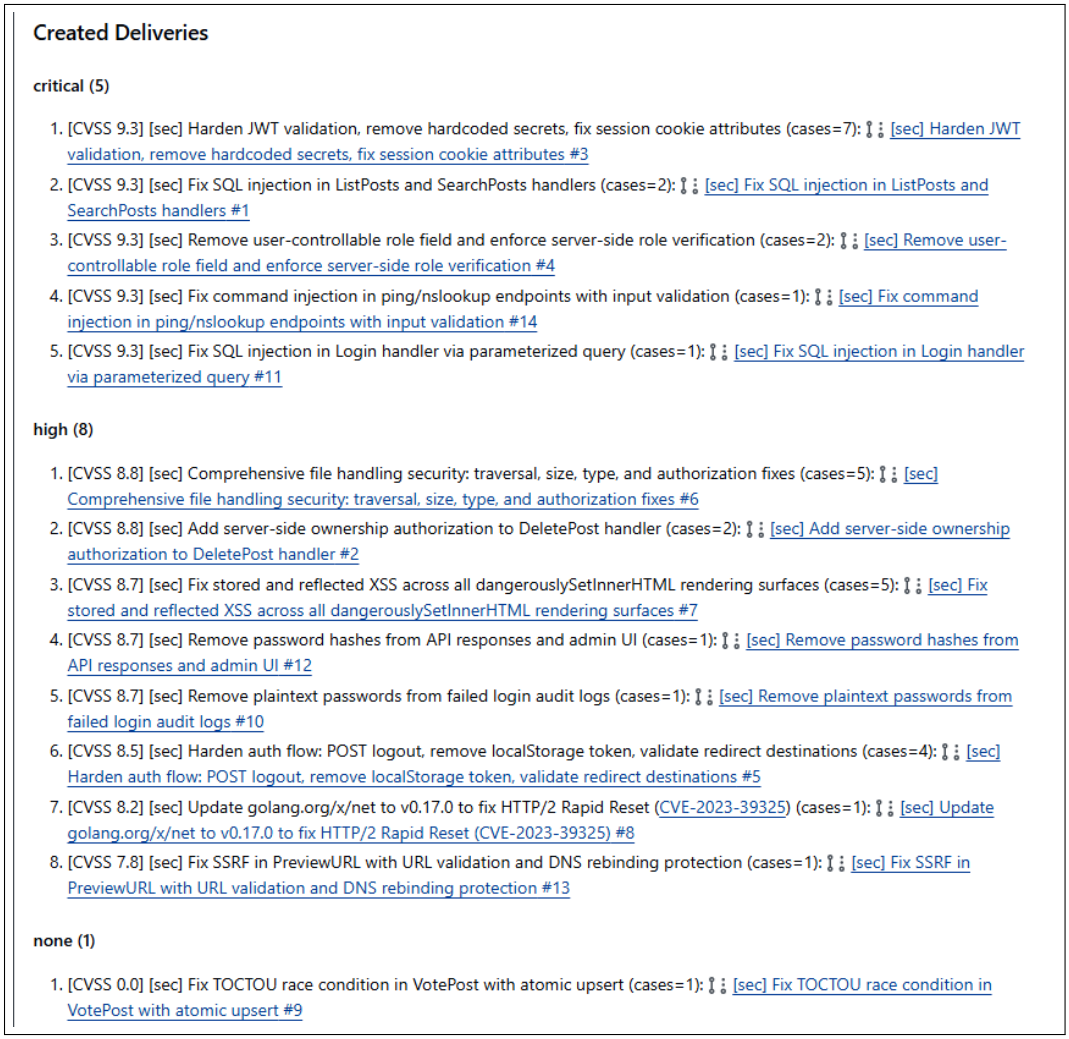
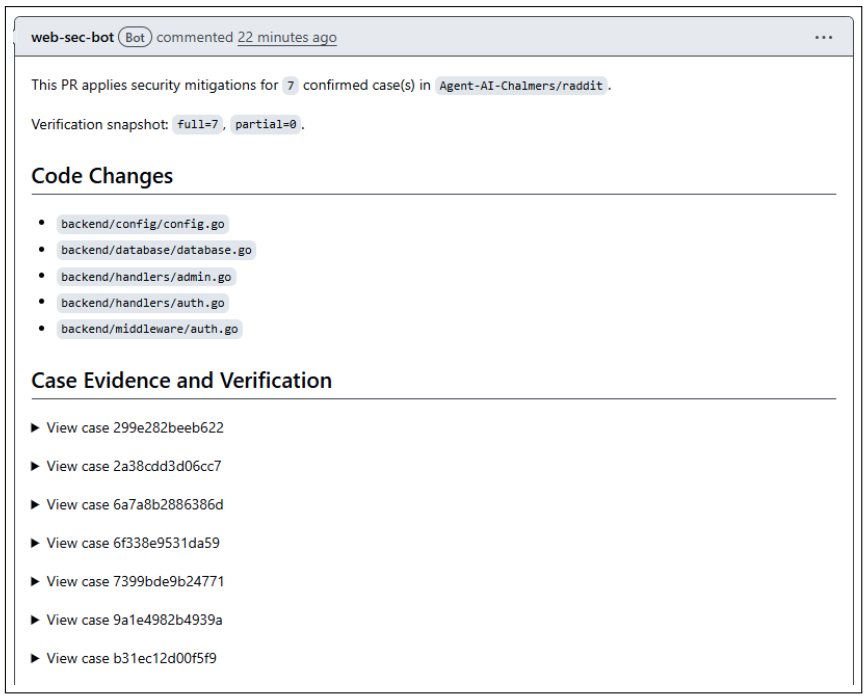


Figure 5.3: Severity-ranked delivery overview from the Raddit workflow run with mimo-v2.5-pro.



web-sec-bot Bot commented 22 minutes ago ···

This PR applies security mitigations for `7` confirmed case(s) in `Agent-AI-Chalmers/raddit`.

Verification snapshot: `full=7`, `partial=0`.

Code Changes

- `backend/config/config.go`
- `backend/database/database.go`
- `backend/handlers/admin.go`
- `backend/handlers/auth.go`
- `backend/middleware/auth.go`

Case Evidence and Verification

▶ View case 299e282beeb622

▶ View case 2a38cdd3d06cc7

▶ View case 6a7a8b2886386d

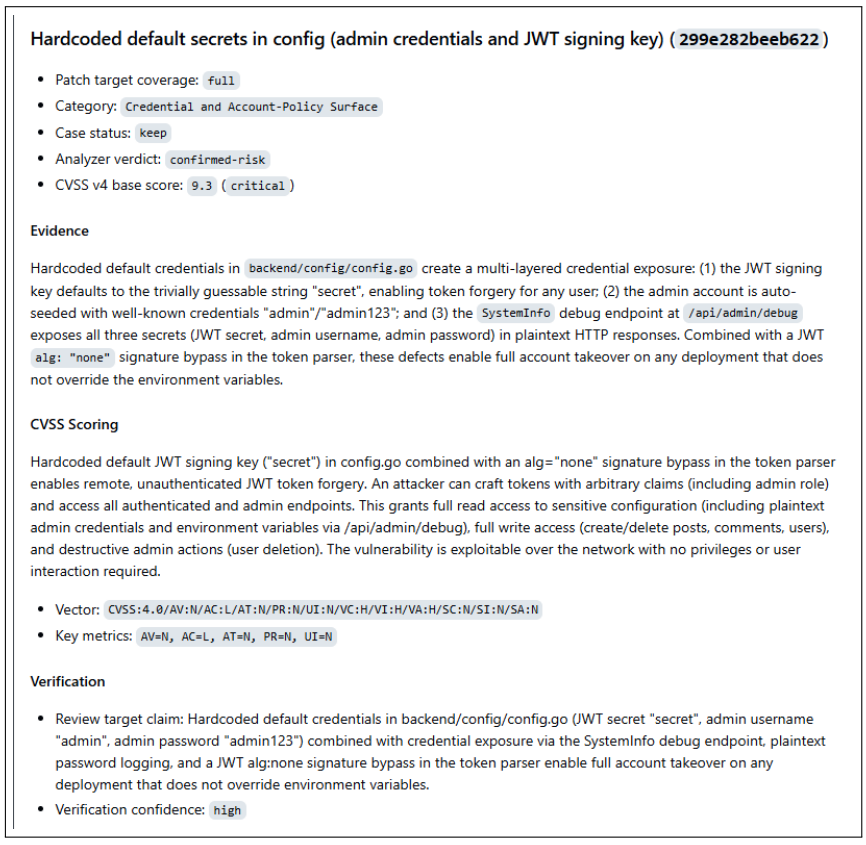
▶ View case 6f338e9531da59

▶ View case 7399bde9b24771

▶ View case 9a1e4982b4939a

▶ View case b31ec12d00f5f9

(a) Delivery page summary.



Hardcoded default secrets in config (admin credentials and JWT signing key) (`299e282beeb622`)

- Patch target coverage: `full`
- Category: `Credential and Account-Policy Surface`
- Case status: `keep`
- Analyzer verdict: `confirmed-risk`
- CVSS v4 base score: `9.3` (`critical`)

Evidence

Hardcoded default credentials in `backend/config/config.go` create a multi-layered credential exposure: (1) the JWT signing key defaults to the trivially guessable string "secret", enabling token forgery for any user; (2) the admin account is auto-seeded with well-known credentials "admin"/"admin123"; and (3) the `SystemInfo` debug endpoint at `/api/admin/debug` exposes all three secrets (JWT secret, admin username, admin password) in plaintext HTTP responses. Combined with a JWT `alg: "none"` signature bypass in the token parser, these defects enable full account takeover on any deployment that does not override the environment variables.

CVSS Scoring

Hardcoded default JWT signing key ("secret") in config.go combined with an alg="none" signature bypass in the token parser enables remote, unauthenticated JWT token forgery. An attacker can craft tokens with arbitrary claims (including admin role) and access all authenticated and admin endpoints. This grants full read access to sensitive configuration (including plaintext admin credentials and environment variables via /api/admin/debug), full write access (create/delete posts, comments, users), and destructive admin actions (user deletion). The vulnerability is exploitable over the network with no privileges or user interaction required.

- Vector: `CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:H/VA:H/SC:N/SI:N/SA:N`
- Key metrics: `AV=N, AC=L, AT=N, PR=N, UI=N`

Verification

- Review target claim: Hardcoded default credentials in backend/config/config.go (JWT secret "secret", admin username "admin", admin password "admin123") combined with credential exposure via the SystemInfo debug endpoint, plaintext password logging, and a JWT alg:none signature bypass in the token parser enable full account takeover on any deployment that does not override environment variables.
- Verification confidence: `high`

(b) Expanded case evidence.

Figure 5.4: Generated delivery details and expandable case evidence from the Raddit workflow run with mimo-v2.5-pro.

6

Discussion

This chapter interprets the evaluation, summarizes the answers to the research questions, and discusses limitations, ethical considerations, and future work.

6.1 RQ1: Multi-Stage Workflow vs. Single-Agent Baseline, and Ablations

The discussion of RQ1 follows the three evidence blocks reported in the Results chapter: outcome differences between DEFAULT and SINGLE-AGENT, verifier retry, and mitigator self-check. Taken together, these results point to a common pattern: the agent workflow does not merely change the aggregate pass rate, but changes how the agent models the vulnerability boundary, the repair target, and patch completeness. When this repository-grounded problem model aligns with the hidden benchmark oracle, the workflow can improve the success outcome. When the two diverge, a more structured or deeper workflow may leave the strict outcome unchanged, or even lower the benchmark outcome. Even so, in the current fixed-model comparison, DEFAULT still achieves the highest strict success count among all workflow variants.

6.1.1 Default vs. Single-Agent Outcome Differences

First we compare the overall behavior of DEFAULT with the SINGLE-AGENT baseline. The results show that DEFAULT outperforms SINGLE-AGENT in repair success rate, but the improvement is limited. At the same time, it introduces additional token consumption and execution cost. From a performance perspective, the gain of DEFAULT is therefore real, but it must be considered together with its substantially higher computational cost.

The success-set comparison further shows that the multi-stage workflow does not simply cover all the successful cases of the SINGLE-AGENT baseline. Both systems have unique successes, although the multi-stage workflow has more of them. This indicates that the two designs can form different vulnerability-analysis conclusions and repair directions.

The off-diagonal cases suggest that this behavioral difference is largely related to the analysis stage. Before the mitigator produces a patch, the analyzer explicitly

constructs a vulnerability explanation, relevant locations, call-chain or data-flow reasoning, and repair boundaries. These structured analysis outputs can substantially shape how the mitigator understands the issue and selects the repair scope. By contrast, although the SINGLE-AGENT baseline can also be prompted to analyze the issue, its analysis, localization, and repair decisions are made inside a single agent, without a separate and explicit analyzer stage. As a result, the multi-stage workflow often forms a different repair strategy from the SINGLE-AGENT baseline and produces a different patch.

This mechanism can have different effects in different cases. When the original input is underspecified, localization is incomplete, or the vulnerability involves multiple related paths, the analyzer's expansion of the problem boundary may help the mitigator avoid under-repair. For example, if the vulnerability is not limited to the explicitly mentioned entry point but also appears in adjacent data paths or call chains, the analyzer may include these locations in the repair scope and guide the mitigator toward a more complete patch. However, the same mechanism can also introduce risk. If the analyzer's repair direction is semantically plausible but diverges from the PatchEval hidden oracle or expected patch direction, the workflow may generate a broader or different repair and fail the benchmark evaluation.

Another relevant observation is the small gap between the end-to-end setting and the location-oracle setting. In our experiments, DEFAULT has nearly identical results in the end-to-end and location-oracle settings. The SINGLE-AGENT baseline also has the same success count in both settings. This suggests that the location oracle provides limited benefit in our setup. Our hypothesis is that this pattern is not solely due to the presence of an independent analyzer agent, but to a more general analysis-oriented design: In DEFAULT, this design appears as a separate analyzer agent. In the SINGLE-AGENT baseline, it appears as a prompt that emphasizes analysis, vulnerable-location search, call chains, and repair boundaries. Both may encourage the agent to perform localization reasoning even without explicit location information, reducing the additional value of the oracle and making the end-to-end and location-oracle results converge.

This explanation can be compared with the PatchEval paper and leaderboard results. In the PatchEval results, the same SWE-Agent + Gemini 2.5 configuration shows a larger gap between the end-to-end and location-oracle settings, suggesting that the location oracle provides a substantial repair-success benefit for that setup. By contrast, our results show only a small gap between the two settings. One possible explanation is that the PatchEval end-to-end prompt places less explicit emphasis on vulnerable locations, call chains, and data-flow analysis. As a result, an agent without oracle location information may move into patch generation without performing sufficient localization and boundary analysis. The prompt mainly asks the agent to inspect relevant code, create and rerun a reproduction script, edit the source code, and consider edge cases, but it does not explicitly require systematic analysis of vulnerable locations, call chains, data-flow paths, or patch boundaries. The prompt excerpt is provided in Appendix A.4 [54].

For this reason, we interpret the small end-to-end versus location-oracle gap as

potentially related to analysis-oriented prompting or staging, rather than simply to benchmark difficulty or to the location oracle being intrinsically unhelpful. This interpretation remains a hypothesis rather than a strict causal conclusion, because we do not perform a controlled ablation of the PatchEval prompt and do not vary only the analysis emphasis within an otherwise identical agent framework. Still, the comparison suggests that explicitly emphasizing vulnerable locations, call chains, and repair-boundary analysis may strengthen localization reasoning even in the end-to-end setting.

6.1.2 Verifier Retry

The next comparison concerns Default and No-Verifier, the two-stage variant that removes the verifier and feedback retry. Default is the full analyzer–mitigator–verifier workflow with feedback retry, while No-Verifier keeps only the analyzer and mitigator. The aggregate results show that Default is slightly better than No-Verifier, but the difference is small. The current results therefore do not prove that verifier + retry provides a stable and large success-rate improvement.

The verifier does not directly know the PatchEval hidden oracle. It cannot know whether a patch would already pass benchmark validation. Instead, it judges from the analyzer output, patch, and repository context whether the repair still leaves a residual path, bypass, or regression risk. The value of verifier + feedback retry therefore should not be understood only through final pass/fail counts. The case-level retry results show that verifier feedback sometimes turns a failing patch into a passing one. They also show cases where the initial patch already passes the oracle, but the verifier still requests a stronger patch. This suggests that the verifier acts as an independent patch-audit stage: it reviews patch completeness according to repository-grounded evidence rather than according to the benchmark oracle.

This also explains the verifier's limitation in the benchmark setting. On the one hand, when a mitigation is clearly too narrow, misses boundary conditions, or leaves a visible bypass, verifier feedback may provide a useful external perspective and lead to a more complete repair through bounded retry. On the other hand, due to the lack of an executable PoC for reference, even when the verifier identifies residual concerns, these issues might be irrelevant to the PoC's core principles. Consequently, resolving such concerns does not guarantee a pass on the benchmark. Thus, verifier + feedback retry is better considered as an observable quality-control mechanism that changes patch revision, rather than as a strong oracle substitute that has been proven to reliably improve the benchmark pass rate.

6.1.3 Mitigator Self-Check

Similarly, the comparison between No-Verifier and No-Verifier+Deep-Self-Check should not be treated as a direct measurement of whether deeper self-check is closer to the hidden oracle. Both variants are analyzer–mitigator workflows without an independent verifier. The self-check does not add a new information source: it asks the mitigator to review its own patch using the same repository evidence and

the same benchmark input, without access to the PoC, unit-test feedback, or the expected patch. Therefore, it can make the patch more internally consistent, but it cannot reliably correct a wrong repair direction or infer hidden behavioral contracts. This explains why the deeper self-check does not increase the overall score of NoVerifier. The off-diagonal cases in Table 5.10 show the same pattern: in one case, self-check moves toward a different but oracle-failing repair path, while in another it broadens the repair and succeeds. The more cautious conclusion is therefore that the current results do not prove a stable benefit from deeper self-check in this setting. This does not mean that the self-check changes are semantically useless or worse in general.

6.2 RQ2: Agent-Based Discovery and Triage vs. Traditional Scanners

The repository results suggest that agentic discovery and traditional scanners are better seen as complementary mechanisms than as direct substitutes. CodeQL, Semgrep, and ZAP produced smaller and more localized finding sets. Their covered answer-key items were concentrated in issues with recognizable syntactic, data-flow, or externally observable signals, such as SQL injection, command injection, SSRF, path traversal, password logging, React XSS, password-hash disclosure, and some error-exposure surfaces. These findings are useful because the scanner outputs contained few confirmed false positives, and scanner alerts can provide high-signal entry points for a broader review workflow.

However, the category-level coverage results show a broader advantage for the agentic runs. Across Book Shop, Raddit, and WorldPress, the agentic workflow covered not only localized sink-style issues but also repository-level logic and design flaws. These included access-control and account-state logic, token and session design, configuration and credential exposure, missing preventive controls, and multi-surface rendering or file-boundary problems. Such cases often require connecting route registration, middleware behavior, model fields, frontend assumptions, configuration defaults, and repair boundaries across files. Traditional scanners may report local problems, but they usually do not assemble those signals into a security case with repository-level context.

This difference is visible in the heatmap in Figure 5.2. For injection, parser, file, and OS-sink categories, scanners sometimes covered a substantial fraction of the answer key, especially on WorldPress and Raddit. In contrast, for access-control, account-state, auth/token, and missing-control categories, scanner coverage was often zero while the agentic runs retained at least partial coverage. This supports the main answer to RQ2: the advantage of agentic discovery is not simply that it produces more findings, but that it expands the scope of detectable issue classes toward cross-file and design-level vulnerabilities.

The results also show why agentic discovery should not be treated as a replacement for traditional tools. Scanner outputs are cheaper, deterministic, and often precise for known bug patterns. They can provide strong candidate evidence for localized

vulnerabilities and can be used as input signals to an agentic workflow. Conversely, agentic discovery can investigate whether scanner alerts correspond to broader cases, group related symptoms, identify adjacent vulnerable surfaces, and reason about issue classes that scanners do not model well. The practical interpretation is therefore a complementary one: scanners provide useful low-level signals, while agentic review adds repository-grounded triage, case construction, and broader vulnerability-class coverage.

This complementarity also extends to day-to-day integration with repository security platforms. Platforms such as GitHub Code Scanning expect findings to arrive in a stable, structured format. In practice, third-party tools commonly upload SARIF reports, a standard JSON format for static-analysis results [55]. Each finding is associated with fields such as a rule identifier, source location, severity, and fingerprint. These fields are not merely shown in the user interface. They allow the platform to filter alerts, track the same finding across repeated scans, and avoid reporting duplicate alerts.

This structure fits traditional scanners more naturally than agentic discovery. A scanner such as CodeQL is backed by a maintained rule set: the same query usually produces the same rule identifier and a similar result shape across runs. By contrast, an agentic review may describe the same vulnerability in different terms across runs. One run may describe an issue as missing authorization on a route, another may classify it as account-state logic, and another may merge it into a broader finding about user-profile data exposure. These outputs may all be useful to a human reviewer, but they are not automatically recognizable as the same alert by a code-scanning platform.

As a result, it is difficult to automatically classify, merge, and deduplicate agentic findings in the same way as scanner alerts. Two descriptions that are equivalent to a human reviewer may not share the same title, category, primary location, or grouping boundary. Repeated runs may therefore create separate alerts for the same vulnerability, or split and merge cases differently across runs. Supporting stable code-scanning alerts would require the system to constrain these descriptions into a fixed set of categories and identifiers, which is closer to maintaining a scanner rule than to free-form repository review. For this reason, agentic discovery is better treated as a semantic review and case-construction layer, or as a complement to scanner outputs, rather than as a replacement for stable code-scanning alert feeds.

6.3 RQ3: Reviewability and Delivery Readiness

RQ3 concerns what happens after vulnerabilities are found. The review workflow is designed to transform raw discovery output into artifacts that a human reviewer can inspect, prioritize, and merge. The results show this process at two levels: a stage-level funnel and the final delivery artifacts.

The funnel in Table 5.12 shows that the workflow does not expose all discovery candidates directly to reviewers. Discovery candidates are first kept and merged into cases during triage. Analyzer judgment then marks which cases are confirmed,

and verification records whether generated patches cover the reviewed claims. This sequence improves reviewability by giving reviewers a structured chain from initial candidate evidence to a case, from case to analyzer, mitigation, and finally verification. Even when the final patch is not perfect, the intermediate artifacts make the state of the case easier to audit than a flat list of raw findings or a one-shot patch.

The delivery stage adds a second layer of organization. Verified cases are packaged into delivery units, and each delivery records its cases, changed files, verification status, residual risks, and synthesis rationale. This matters because repository repairs are often not one vulnerability to one pull request. Some vulnerabilities share the same code path or control boundary and should be reviewed together; others are better kept separate. The merge stage decides which cases should appear in the same delivery, and the reviewer should inspect them together.

Delivery count gap is a useful but incomplete signal for this boundary. Large positive gaps, such as MiniMax M2.7 on Book Shop and WorldPress or Doubao Seed 2.0 Lite on Book Shop and Raddit, indicate fragmented packaging: reviewers would need to inspect many more units than the ideal delivery plan. A small or zero gap is not automatically better, because it can arise from missed cases or overly broad bundling. This is why the gap must be read together with answer-key coverage and manual inspection of the delivery contents. The WorldPress Doubao run illustrates this point: it matched the ideal delivery count, but its answer-key coverage was lower than the two higher-gap agentic runs.

The Raddit delivery artifact shown in Figures 5.3 and 5.4 provides one example of such manual delivery inspection. The overview ranks deliveries by CVSS v4 severity and shows how many cases are included in each delivery, giving reviewers a prioritization view. The delivery page then exposes changed files, and expandable per-case reports. In the expanded case view, the reviewer can inspect the category, analyzer verdict, CVSS v4 score, evidence, scoring rationale, and verification information. This turns a generated repair from an opaque patch into a review packet with evidence and audit context.

The main answer to RQ3 is therefore that the workflow improves delivery readiness by producing scoped, evidence-bearing deliveries. It does not guarantee that the delivery organization is always ideal, and the delivery gap results show that over-fragmentation remains a practical challenge. Its contribution is that it makes this organization explicit: the reviewer can see which cases were grouped, why they were grouped, what changed, which claims were verified, and what residual risks remain. This is the difference between agent-generated code changes and agent-generated security-review deliveries.

6.4 Model Capability, Cost, and Review Load

The comparison among the MIMO, MiniMax, and Doubao configurations shows a common issue that spans discovery, triage, cost, and delivery.

Under the same discovery-and-triage workflow, all three model configurations cov-

ered issue classes that the traditional scanner baselines largely missed, but their answer-key recall and case counts differed. A high-recall discovery stage can be useful even if it produces extra candidates. The important question is whether triage can manage a large volume of candidates by removing weak findings, merging duplicates, and grouping related observations into cases. When triage is weaker, extra discovery output becomes a larger retained case set, which increases the number of downstream analyzer inputs even before patch generation begins. Theoretically, this should increase later-stage token usage, and the token-cost diagnostics in Table 5.13 show that this did happen in some Doubao runs: they used substantially more total input tokens than the other model configurations. Even so, Doubao's lower per-token price kept its observed cost low.

The same behavior appears later in delivery organization. Some runs combined related cases into fewer deliveries, while others directly output many single-case deliveries.

If we only consider the workflow cost, this is economically attractive. However, it does not mean that the overall review process is cheap. Delivery fragmentation can move cost outside the measured workflow: reviewers must inspect more pull requests or delivery pages and reconstruct relationships between related fixes. Thus, the practical value of agentic discovery depends not only on recall or measured API cost, but also on whether triage and delivery can reduce many raw candidates into fewer reviewable deliveries.

6.5 Limitations and Observed Challenges

When interpreting the experimental results, it is necessary to consider both the methodological limitations of the evaluation design and the practical challenges observed during system execution.

First, the study is limited to web application vulnerabilities in white-box defensive review settings. It does not cover black-box testing, broader software assurance, or entirely new vulnerability classes. Repair-oriented benchmarks also do not fully capture repository review workflows that include discovery, triage, and delivery, while repository case studies may remain too small to support broad generalization.

Second, issue-driven benchmarks such as PatchEval have a relatively narrow task scope, focusing primarily on vulnerability localization and repair. They cannot fully simulate the real-world workflow of discovering vulnerabilities from scratch in large repositories. The low strict success counts should also be interpreted with care: failed cases are not all identical unresolved vulnerabilities. Some failures reflect genuine agent limitations, such as incomplete localization, under-repair, over-broad repair, or failure to preserve existing behavior. Others are affected by benchmark-specific factors, including underspecified repair contracts, hidden behavioral or API expectations, and runtime-oracle problems such as script errors, missing test dependencies, or mismatches between the test harness and the checked-out repository. Appendix A.2 summarizes a manual audit of these quality risks. Therefore, the

PatchEval results should be read as strict benchmark outcomes rather than as a complete taxonomy of practical repair success and failure.

Third, the workflow depends on the reasoning quality, context handling, and failure modes of externally provided LLMs. Results may vary with model-provider behavior, prompt design, runtime limits, and external service variance. Wall-clock time is also an imperfect measure of system efficiency because it is affected by network fluctuations and queueing delays. Token consumption and API cost therefore provide more objective indicators of computational workload. Relatedly, the experimental scope is constrained by computational budget. Each additional workflow, model, or prompt variant requires rerunning many repository-level repair tasks, which is expensive in token usage, API cost, and wall-clock time. Therefore, the evaluation emphasizes controlled comparisons under matched model conditions rather than exhaustive exploration of all possible model-agent combinations.

Fourth, the system makes strategic compromises when processing large repositories. To control computational cost, deep cross-file reasoning is triggered only when initial suspicious signals are detected within an individual file. This strategy makes repository-scale review more practical, but it also risks missing subtle cross-file vulnerabilities that do not expose localized clues.

Finally, automated validation cannot replace human review. Existing tests and automated checks cannot provide perfect conclusions about actual exploitability, deeper business-logic impact, or whether a repair preserves the original behavior of the program. These judgments still require manual review before deployment.

6.6 Ethics and Governance

The system has a dual-use character because automated security tools can be misused to identify vulnerabilities. This thesis positions the work as a blue-team defensive system for authorized repository review. It does not aim to build offensive exploit capabilities.

Practical deployment should enforce tiered access, continuous monitoring, and clear governance. Agent actions should remain isolated from unauthorized external targets, and evaluation should remain confined to local or containerized environments. The long-term defensive benefit lies in reducing mitigation cost and improving the reliability of security review, but that benefit depends on responsible deployment boundaries.

7

Conclusion

This thesis studies a multi-stage agentic framework for web vulnerability detection, mitigation, and patch delivery. The core problem is that vulnerability resolution is not only patch generation: it also requires discovery, analysis, validation, prioritization, and delivery. The proposed workflow decomposes these responsibilities into explicit stages and evaluates whether this structure improves repair effectiveness, value in repository review, and delivery readiness.

On one hand, in the issue-level repair setting, the fixed-model PatchEval experiment shows a slight but consistent advantage for the full multi-stage workflow. DEFAULT achieves the highest strict success count among the evaluated variants, but the improvement over SINGLE-AGENT is small and comes with higher token use and price. The case-level comparisons show that staging changes the repair behavior and produces more explicit explanations than the single-agent baseline, rather than simply adding successful cases on top of it. Analyzer output can help define repair boundaries and avoid incomplete repairs, while verifier feedback can catch residual paths and sometimes improve the initially failing patch. At the same time, these stages are not substitutes for the benchmark oracle, because even broader analysis or stronger self-check can also move the patch away from the expected repair.

On the other hand, for repository review, the case studies show that agentic review can complement traditional scanners by covering issue classes that are difficult for localized static or dynamic tools. CodeQL, Semgrep, and ZAP remained useful for high-signal localized findings, but the agentic runs covered more repository-level logic and design issues, including access control, account-state behavior, token/session design, configuration exposure, and missing controls. This advantage depends on model capability. The discovery stage benefits from recall with both coverage and candidate quality; in the triage stage, a capable model can merge overlapping candidates and apply light filtering, reducing candidates into a smaller set of cases.

For delivery readiness, the workflow turns retained and verified cases into reviewable deliveries rather than leaving reviewers with raw findings or patches. The stage-level records, analyzer judgments, verification summaries, and CVSS v4 prioritization make generated repairs easier to inspect. However, delivery organization remains a practical challenge. Large delivery-count gaps show that some runs fragmented related fixes into too many units, while a small gap can also hide missed coverage or overly broad grouping. The core tradeoff is between patch atomicity and delivery

efficiency: deliveries should remain narrow enough to review safely, but related fixes that share a patch surface or control boundary should not force reviewers to reconstruct the same repair context repeatedly.

Overall, the main conclusion is that multi-stage orchestration provides a practical way to make agentic security review more inspectable and controllable. Its value should not be judged only by raw patch success. A useful agentic security workflow must also expose why a case was kept, how the vulnerability was analyzed, whether the patch covers the reviewed claim and how the repair should be delivered for human review. The current results support this direction, while also showing that workflow quality remains largely dependent on the model: model choice affects discovery recall, triage compression, case grouping, and delivery organization. Besides, the verifier has structural limits. Although it can audit the patch based on repository evidence, it cannot replace the benchmark oracle or human security judgment.

8

Future Work

We should broaden repository evaluation across more repositories, vulnerability types, and development workflows. Additional integrations with static and dynamic scanners, dependency analysis tools, and repository automation platforms would make the workflow more realistic and easier to compare against existing security engineering practices.

We could also incorporate a memory module to support reuse of operational experience across runs. Such a module could help the agent retain recurring repair patterns, tool-use strategies, and project-specific constraints, thereby reducing redundant exploration in later tasks.

Another promising direction is to develop a dedicated benchmark for AVR agents. Unlike conventional vulnerability-fixing datasets that often rely on official patches as implicit ground truth, such a benchmark should provide the relevant application-specific semantics and context needed to understand the vulnerability, while avoiding over-specification of the expected fix. In particular, benchmark instances should include carefully constructed prompts, vulnerability-triggering proof-of-concepts, and robust unit tests that capture the intended security properties rather than patch-specific implementation details. This would allow the evaluation to accommodate diverse yet correct fixes and better reflect practical settings in which agents must reason from incomplete but relevant contextual information.

Bibliography

[1] CVE Program. "Cve metrics." (2026), [Online]. Available: https://www.cve.org/About/Metrics (visited on 04/27/2026).

[2] 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, 2011.

[3] Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "SequenceR: Sequence-to-Sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1943–1959, 2019. DOI: 10.1109/TSE.2019.2940179.

[4] Z. Xu, S. Jain, and M. Kankanhalli, *Hallucination is inevitable: An innate limitation of large language models*, 2024. arXiv: 2401.11817 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2401.11817.

[5] S. Yao, J. Zhao, D. Yu, *et al.*, "ReAct: Synergizing reasoning and acting in language models," in *The eleventh international conference on learning representations*, 2022.

[6] T. Guo, X. Chen, Y. Wang, *et al.*, *Large language model based multi-agents: A survey of progress and challenges*, 2024. arXiv: 2402.01680 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2402.01680.

[7] I. Bouzenia, P. Devanbu, and M. Pradel, "RepairAgent: An autonomous, LLM-based agent for program repair," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, IEEE, 2025, pp. 2188–2200.

[8] X. Li, S. Wang, S. Zeng, Y. Wu, and Y. Yang, "A survey on LLM-based multi-agent systems: Workflow, infrastructure, and challenges," *Vicinagearth*, vol. 1, no. 1, p. 9, 2024.

[9] N. Le Hai, D. M. Nguyen, and N. D. Bui, "On the impacts of contexts on repository-level code generation," in *Findings of the Association for Computational Linguistics: NAACL 2025*, 2025, pp. 1496–1524.

[10] OWASP Foundation. "Owasp top 10:2025." (2025), [Online]. Available: https://owasp.org/Top10/2025/ (visited on 04/27/2026).

[11] Forum of Incident Response and Security Teams. "Common Vulnerability Scoring System Version 4.0: Specification Document." (2023), [Online]. Available: https://www.first.org/cvss/v4.0/specification-document (visited on 04/25/2026).

[12] OWASP Foundation. "Source Code Analysis Tools." (2026), [Online]. Available: https://owasp.org/www-community/Source_Code_Analysis_Tools (visited on 04/25/2026).

[13] OWASP Foundation. "Dynamic Application Security Testing." (2026), [Online]. Available: https://owasp.org/www-project-devsecops-guideline/latest/02b-Dynamic-Application-Security-Testing (visited on 04/25/2026).

[14] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A static analysis tool for detecting web application vulnerabilities," in *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, IEEE, 2006, pp. 258–263. DOI: 10.1109/SP.2006.29.

[15] S. Kals, E. Kirda, C. Kruegel, and N. Jovanovic, "SecuBat: A web vulnerability scanner," in *Proceedings of the 15th International Conference on World Wide Web*, ACM, 2006, pp. 247–256. DOI: 10.1145/1135777.1135817.

[16] K. Li, S. Chen, L. Fan, *et al.*, "Comparison and evaluation on static application security testing (SAST) tools for Java," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 921–933.

[17] W. Charoenwet, P. Thongtanunam, V.-T. Pham, and C. Treude, *An empirical study of static analysis tools for secure code review*, 2024. DOI: 10.48550/arXiv.2407.12241. arXiv: 2407.12241 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2407.12241.

[18] J. Bau, E. Bursztein, D. Gupta, and J. Mitchell, "State of the art: Automated black-box web application vulnerability testing," in *2010 IEEE symposium on security and privacy*, IEEE, 2010, pp. 332–345.

[19] I. Chorell and C. Ekberg, "A comparative analysis of open source dynamic application security testing tools," M.S. thesis, Linköping University, 2024.

[20] M. Monperrus, "Automatic software repair: A bibliography," *ACM Computing Surveys*, vol. 51, no. 1, pp. 1–24, 2018. DOI: 10.1145/3105906.

[21] Q. Zhang, C. Fang, Y. Ma, W. Sun, and Z. Chen, "A survey of learning-based automated program repair," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 2, pp. 1–69, 2023.

[22] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *2013 35th international conference on software engineering (ICSE)*, IEEE, 2013, pp. 802–811.

[23] F. Long and M. Rinard, "Staged program repair with condition synthesis," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 166–178.

[24] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *Proceedings of the 2015 international symposium on software testing and analysis*, 2015, pp. 24–36.

[25] F. Long and M. Rinard, "An analysis of the search spaces for generate and validate patch generation systems," in *Proceedings of the 38th International Conference on Software Engineering*, ACM, 2016, pp. 702–713. DOI: 10.1145/2884781.2884872.

[26] X.-B. D. Le, D.-H. Chu, D. Lo, C. L. Goues, and W. Visser, "Overfitting in semantics-based automated program repair," *Empirical Software Engineering*, vol. 23, pp. 3007–3033, 2018. DOI: 10.1007/s10664-017-9577-2.

[27] M. Fu, C. Tantithamthavorn, T. Le, V. Nguyen, and D. Phung, "VulRepair: A T5-based automated software vulnerability repair," in *Proceedings of the 30th ACM joint european software engineering conference and symposium on the foundations of software engineering*, 2022, pp. 935–947.

[28] Y. Nong, H. Yang, L. Cheng, H. Hu, and H. Cai, "APPATCH: Automated adaptive prompting large language models for real-world software vulnerability patching," in *34th USENIX Security Symposium (USENIX Security 25)*, 2025, pp. 4481–4500.

[29] Z. Yu, Z. Guo, Y. Wu, *et al.*, "PATCH AGENT: A practical program repair agent mimicking human expertise," in *34th USENIX Security Symposium (USENIX Security 25)*, 2025, pp. 4381–4400.

[30] J. Yang, C. E. Jimenez, A. Wettig, *et al.*, "SWE-agent: Agent-computer interfaces enable automated software engineering," *Advances in Neural Information Processing Systems*, vol. 37, pp. 50 528–50 652, 2024.

[31] W. Tao, Y. Zhou, Y. Wang, W. Zhang, H. Zhang, and Y. Cheng, "MAGIS: LLM-based multi-agent framework for GitHub issue resolution," *Advances in Neural Information Processing Systems*, vol. 37, pp. 51 963–51 993, 2024.

[32] M. Seo, W. Choi, S. Shin, and M. You, "AutoPatch: Multi-agent framework for patching real-world CVEs generated by outdated LLMs," SSRN, SSRN Working Paper 6339960. DOI: 10.2139/ssrn.6339960. [Online]. Available: https://doi.org/10.2139/ssrn.6339960.

[33] S. Ouyang, W. Yu, K. Ma, *et al.*, *RepoGraph: Enhancing AI software engineering with repository-level code graph*, 2024. arXiv: 2410.14684 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2410.14684.

[34] Forum of Incident Response and Security Teams. "Common Vulnerability Scoring System Version 4.0 Calculator." (2026), [Online]. Available: https://www.first.org/cvss/calculator/v4.0 (visited on 06/06/2026).

[35] OWASP Foundation. "OWASP Risk Rating Methodology." (2026), [Online]. Available: https://owasp.org/www-community/OWASP_Risk_Rating_Methodology (visited on 06/06/2026).

[36] Z. Wei, J. Zeng, M. Wen, *et al.*, *PatchEval: A new benchmark for evaluating LLMs on patching real-world vulnerabilities*, 2025. arXiv: 2511.11019 [cs.CR]. [Online]. Available: https://arxiv.org/abs/2511.11019.

[37] C. E. Jimenez, J. Yang, A. Wettig, *et al.*, "SWE-bench: Can language models resolve real-world GitHub issues?" In *The Twelfth International Conference on Learning Representations*, 2024. [Online]. Available: https://openreview.net/forum?id=VTF8yNQM66.

[38] Q.-C. Bui, R. Scandariato, and N. E. D. Ferreyra, "Vul4J: A dataset of reproducible Java vulnerabilities geared towards the study of program repair techniques," in *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, 2022, pp. 464–468. DOI: 10.1145/3524842.3528482.

[39] Agent-AI-Chalmers. "Book Shop." (2026), [Online]. Available: https://github.com/Agent-AI-Chalmers/book_shop (visited on 06/06/2026).

[40] Agent-AI-Chalmers. "Raddit." (2026), [Online]. Available: https://github.com/Agent-AI-Chalmers/raddit (visited on 06/06/2026).

[41] Agent-AI-Chalmers. “WorldPress.” (2026), [Online]. Available: https://github.com/Agent-AI-Chalmers/worldpress (visited on 06/06/2026).
[42] Semgrep. “Semgrep Documentation.” (2026), [Online]. Available: https://semgrep.dev/docs/ (visited on 06/06/2026).
[43] GitHub. “CodeQL Documentation.” (2026), [Online]. Available: https://codeql.github.com/docs/ (visited on 06/06/2026).
[44] OWASP Foundation. “Zed Attack Proxy.” (2026), [Online]. Available: https://www.zaproxy.org/ (visited on 06/06/2026).
[45] PatchEval. “Patcheval leaderboard.” (2026), [Online]. Available: https://patcheval.github.io/ (visited on 05/07/2026).
[46] Open Source Vulnerabilities. “CVE-2021-21360.” (2026), [Online]. Available: https://osv.dev/vulnerability/CVE-2021-21360 (visited on 06/06/2026).
[47] Open Source Vulnerabilities. “CVE-2023-50726.” (2026), [Online]. Available: https://osv.dev/vulnerability/CVE-2023-50726 (visited on 06/06/2026).
[48] Open Source Vulnerabilities. “CVE-2015-1326.” (2026), [Online]. Available: https://osv.dev/vulnerability/CVE-2015-1326 (visited on 06/06/2026).
[49] Open Source Vulnerabilities. “CVE-2020-15084.” (2026), [Online]. Available: https://osv.dev/vulnerability/CVE-2020-15084 (visited on 06/06/2026).
[50] Open Source Vulnerabilities. “CVE-2021-21321.” (2026), [Online]. Available: https://osv.dev/vulnerability/CVE-2021-21321 (visited on 06/06/2026).
[51] Open Source Vulnerabilities. “CVE-2022-29822.” (2026), [Online]. Available: https://osv.dev/vulnerability/CVE-2022-29822 (visited on 06/06/2026).
[52] Open Source Vulnerabilities. “CVE-2021-32796.” (2026), [Online]. Available: https://osv.dev/vulnerability/CVE-2021-32796 (visited on 06/06/2026).
[53] Open Source Vulnerabilities. “CVE-2024-49750.” (2026), [Online]. Available: https://osv.dev/vulnerability/CVE-2024-49750 (visited on 06/06/2026).
[54] Agent-AI-Chalmers. “SWE-Agent End-to-End Prompt Template without Feedback.” (2026), [Online]. Available: https://github.com/Agent-AI-Chalmers/PatchEval/blob/0876897b69426d82ea46130a3e11dbe917d5366b/patcheval/exp_agent/sweagent/configs/template_without_feedback.yaml (visited on 06/06/2026).
[55] OASIS. “Static Analysis Results Interchange Format (SARIF) Version 2.1.0.” (2019), [Online]. Available: https://docs.oasis-open.org/sarif/sarif/v2.1.0/os/sarif-v2.1.0-os.html (visited on 05/10/2026).

A

Appendix 1

A.1 Full Issue-Review Benchmark Results

Table A.1 and Table A.2 report the full issue-review benchmark results. For the fixed-model workflow variants, the model is Deepseek V4 Pro and the variant names follow the definitions in the methodology chapter. Gemini2.5-based agents and GPT-5-based PatchEval leaderboard systems are included as reference comparisons because they use different model families or agent frameworks [36], [45].

Table A.1: Full issue-review benchmark results in the end-to-end setting.

System/Variant	Model	Overall	Avg. Token (K)	Avg. Price (USD)
SWE-Agent	Gemini2.5	13.91% (32/230)	1190.35	2.387
OpenHands	Gemini2.5	18.70% (43/230)	1817.37	4.640
Claude-Code	Gemini2.5	12.61% (29/230)	2492.83	6.401
Default	Deepseek V4 Pro	**33.91% (78/230)**	1938.38	0.215955
No-Verifier	Deepseek V4 Pro	33.04% (76/230)	1618.15	0.177376
Single-Agent	Deepseek V4 Pro	31.74% (73/230)	1619.09	0.142768
No-Verifier+Deep-Self-Check	Deepseek V4 Pro	31.74% (73/230)	1632.86	0.180573

Table A.2: Full issue-review benchmark results in the location-oracle setting.

System/Variant	Model	Overall	Avg. Token (K)	Avg. Price (USD)
OpenHands	GPT-5	**36.1% (83/230)**	–	–
ClaudeCode	GPT-5	34.4% (79/230)	–	–
SWE-agent	GPT-5	33.0% (76/230)	–	–
–	GPT-5	29.1% (67/230)	–	–
SWE-Agent	Gemini2.5	23.04% (53/230)	561.07	1.128
OpenHands	Gemini2.5	21.30% (49/230)	1141.62	2.933
Claude-Code	Gemini2.5	18.70% (43/230)	333.06	0.853
Default	Deepseek V4 Pro	33.91% (78/230)	1430.24	0.180557
No-Verifier	Deepseek V4 Pro	33.48% (77/230)	1190.65	0.149150
Single-Agent	Deepseek V4 Pro	31.74% (73/230)	963.37	0.110590
No-Verifier+Deep-Self-Check	Deepseek V4 Pro	32.61% (75/230)	1248.71	0.153615

A.2 PatchEval Quality Audit Summary

This section summarizes a manual audit used to interpret strict PatchEval failures. The audit does not replace the benchmark metric used in Chapter 5. Instead, it identifies quality risks that affect how failed cases should be interpreted.

Two risk categories were considered. The first is input-side specification risk: whether the CVE description and vulnerable-location information provide enough behavioral detail to infer an equivalent repair. The second is runtime-oracle risk: whether the PoC, unit test, harness, image, and checked-out repository reliably judge the generated patch.

Table A.3: Representative input-side specification risks in PatchEval.

Case	Missing repair specification	Interpretation risk
`CVE-2023-25173`	The primary GID must also appear in `AdditionalGids`, and the PoC expects a specific ordering.	A plausible patch can reason about supplementary groups correctly in security terms but still miss the exact invariant expected by the oracle.
`CVE-2021-46561`	Changing a user's organization must require the Secretariat role and use a specific error helper.	The vulnerable code path is visible, but the required authorization policy and API error contract are not fully specified in the input.
`CVE-2021-41246`	Login handling must distinguish same-user, different-user, anonymous-session, and custom-session-store cases.	A direct session-regeneration patch can fix the intuitive risk while breaking application state or missing branch-specific behavior.
`CVE-2021-37712`	Path reservation must use a precise normalization strategy, including slash stripping, Unicode normalization, lowercasing, and Windows collision handling.	The input points toward the right vulnerability class, but the exact normalization order and concurrency behavior are hidden.
`CVE-2020-26299`	Windows separators must be converted before normalization while preserving existing path semantics.	A security-plausible containment check can fail because the oracle expects behavioral equivalence rather than simple rejection.

Note. These cases are representative examples from the audit, not a complete classification of all failed tasks.

The runtime-oracle audit identified 22 cases with potential validation reliability issues, listed in Table A.4. These include syntax errors in benchmark scripts, missing functions or constants referenced by tests, dependency drift, mismatches between the test harness and the checked-out repository, baseline tests that already pass, and one flaky unit-test outcome. These cases do not mean that the full 230-case subset should be discarded; the main results retain the benchmark's strict protocol for comparability. However, they show why strict failures should be interpreted as benchmark outcomes rather than as a direct count of simple vulnerabilities left unfixed.

Table A.4: Runtime-oracle validation reliability risks in PatchEval.

Case	Risk type	Observed issue
CVE-2020-7764	Script syntax error	`vul-run.sh` has a quote syntax error before the repair is evaluated.
CVE-2021-21354	Dependency drift	The official fix test fails because `test_nightly_archive` depends on a Mozilla hg URL that returns HTTP 404.
CVE-2024-52309	Checkout mismatch	The vulnerable baseline does not compile because the test references `EventManager`-related symbols absent from the pre-fix checkout.
CVE-2024-22199	Missing API	The test calls `Engine.SetAutoEscape`, which is not present in the repository.
CVE-2023-45128	Missing constants	The test references `ErrMissingParam` and `ErrMissingHeader`, which are not present in the repository.
CVE-2021-36157	Missing constant	The test depends on `errInvalidTenantID`, which is not present in the repository.
CVE-2024-52010	Missing types	The test depends on `Instance.Manager` and `MockManager`, which are not present in the repository.
CVE-2023-41891	API mismatch	The test uses a `NewSortParameter` signature that does not match the checked-out code.
CVE-2022-23536	Missing constant	The test depends on `errOpsGenieAPIKeyFileNotAllowed`, which is not present in the repository.
CVE-2024-24579	Missing symbol	The test depends on `tarVisitor`, which is not present in the repository.
CVE-2020-4053	Missing helper	The test depends on `cleanJoin`, which is not present in the repository.
CVE-2025-24366	Missing helper	The test depends on `canAcceptRsyncArgs`, which is not present in the repository.
CVE-2021-32783	API mismatch	The test uses a `dag.EnsureService` signature that does not match the checked-out code.
CVE-2021-41803	Missing constant	The test depends on `invalidSegmentName`, which is not present in the repository.
CVE-2023-23947	Missing API	The test depends on `common.PermissionDeniedAPIError`, which is not present in the repository.
CVE-2022-35936	Baseline already passes	The vulnerable baseline passes directly, so the runtime oracle does not distinguish the vulnerable and fixed states.
CVE-2022-31130	Missing helper	The test depends on `contexthandler.WithAuthHTTPHeader`, which is not present in the repository.
CVE-2024-47616	Missing helper	The test depends on `validateJWT`, which is not present in the repository.
CVE-2023-49736	Test selection mismatch	Pytest cannot find the specified test node.
CVE-2023-30172	Missing import target	The test imports `validate_path_is_safe`, which is not present in the repository.
CVE-2024-21542	Missing import target	The test imports `luigi.safe_extractor`, which is not present in the repository.
CVE-2018-3778	Flaky unit-test outcome	The original run produced asynchronous Tape failures, while rerunning the same final patch made both PoC and unit phases pass.

A.3 Case-Level Outcome Difference Sets

This section lists the case-level difference sets used to interpret the off-diagonal overlap results in Table 5.5 and Table 5.9. Runtime-oracle outliers are marked separately when they are part of the strict overlap count but are less suitable for qualitative interpretation.

Table A.5: End-to-end case-level differences between Default and Single-Agent.

Difference set	Count	Cases
Single-Agent only	6	CVE-2020-11053, CVE-2023-34233, CVE-2023-50726, CVE-2024-56362, CVE-2025-23042, and CVE-2025-24806.
Single-Agent only, runtime-oracle outlier	1	CVE-2022-23536.
Default only	12	CVE-2015-1326, CVE-2015-8213, CVE-2019-10788, CVE-2021-21360, CVE-2021-21384, CVE-2021-26921, CVE-2021-32796, CVE-2021-32803, CVE-2021-3583, CVE-2023-25168, CVE-2023-5122, and CVE-2025-27154.

Note. The first two rows together account for the 7 Single-Agent-only cases in Table 5.5; the outlier row is separated to avoid treating a patch-specific oracle dependency as an ordinary behavioral difference.

CVE-2022-23536 is separated because the validation test depends on the patch introducing `errOpsGenieAPIKeyFileNotAllowed`, a constant absent from the vulnerable checkout. The Single-Agent patch introduced that expected error path and also checked `GlobalConfig.OpsGenieAPIKeyFile`. The Default patch introduced the error path but did not add the global configuration check. This makes the case useful as runtime-oracle evidence, but less suitable as an ordinary qualitative example of a behavioral difference between the two workflows.

Table A.6: No-verifier strict outcome case-level differences.

Outcome difference	Count	Cases
No-Verifier only	13	CVE-2018-12976, CVE-2018-3733, CVE-2019-10792, CVE-2020-28494, CVE-2021-21411, CVE-2021-26921, CVE-2021-32796, CVE-2021-3583, CVE-2021-43798, CVE-2022-1883, CVE-2022-29822, CVE-2022-39286, and CVE-2023-29159.
No-Verifier+Deep-Self-Check only	10	CVE-2017-1001004, CVE-2019-10787, CVE-2020-8132, CVE-2021-21291, CVE-2022-0155, CVE-2022-23536, CVE-2023-34233, CVE-2024-0243, CVE-2024-49750, and CVE-2025-23042.

A.4 PatchEval End-to-End Prompt Excerpt

The following excerpt is the generic repair instruction discussed in Chapter 6, taken from the PatchEval SWE-Agent end-to-end prompt template without feedback [54].

```
Follow these steps to resolve the issue:
1. As a first step, it might be a good idea to find and read code relevant
↪  to the <pr_description>
2. Create a script to reproduce the error and execute it using the bash
↪  tool, to confirm the error
3. Edit the sourcecode of the repo to resolve the issue
4. Rerun your reproduce script and confirm that the error is fixed!
5. Think about edgecases and make sure your fix handles them as well
Your thinking should be thorough and so it's fine if it's very long.
```