International Journal of Scientific Research and Engineering Development-– Volume 9 Issue 4, July – August 2026
Available at www.ijsred.com

RESEARCH ARTICLE OPEN ACCESS

A Multi-Agent Architecture for Autonomous Security-Focused Code Review in GitHub Pull Requests

Coordination, Communication, and Memory in an Agentic AppSec Reviewer

Jahnavi Somaraju[1], Sreevathsa Ganesh P R[2]
Assistant Professor[1], Final Year UG Student[2]
Department of Artificial Intelligence and Data Science, Aditya College of Engineering, Madanapalle
Email: prsganesh1405@gmail.com

Abstract

Security review of pull requests (PRs) remains manual, inconsistent, and dependent on reviewer expertise, creating a bottleneck in modern DevSecOps pipelines. Single-agent LLM reviewers, while easy to deploy, conflate detection, explanation, and prioritization inside one undifferentiated reasoning pass, making them prone to confirmation bias and inconsistent output across repeated runs. This paper proposes a multi-agent architecture — the AppSec Review Agent — that decomposes PR security review across five cooperating agents: an Orchestrator, three deterministic specialist agents (static analysis, dependency-CVE lookup, secret scanning), and an LLM Reasoning Agent supported by a three-tier memory system. Agents communicate through the Model Context Protocol (MCP), a standardized client-server tool-calling layer, rather than bespoke point-to-point integrations. We present the complete system architecture, agent interaction sequence, communication protocol, memory architecture, deployment architecture, coordination algorithms, and a tagged case-study protocol for reporting agent behaviour consistently across evaluations. A worked case study demonstrates the pipeline end-to-end on a deliberately vulnerable code sample. Full benchmark-scale quantitative evaluation is scoped as an immediate next phase and is described here as a concrete, reproducible protocol rather than presented as completed results.

Keywords *— Multi-Agent Systems, Agentic AI, Application Security, Large Language Models, Model Context Protocol, Retrieval-Augmented Generation, DevSecOps, Vector Memory*

I. INTRODUCTION

A. Motivation

Modern software teams merge dozens of pull requests daily. Security review is frequently reduced to a checklist step or skipped entirely under delivery pressure, and Static Application Security Testing (SAST) tools apply fixed rule sets that generate high false-positive rates, causing developers to distrust or disable them [7].

B. Problem Statement

A single monolithic LLM tasked with reviewing a PR for security issues must simultaneously parse code, recall vulnerability patterns, reason about exploitability, and phrase a response — all inside one undifferentiated call. This conflation is a known source of failure: agentic reviewers have been shown to be susceptible to confirmation bias, trusting a PR's stated intent rather than independently verifying it [1], and to adversarial framing embedded directly in PR descriptions [4].

C. Why Multi-Agent Systems Are Needed

Splitting detection from reasoning across specialized agents keeps the parts of the pipeline that must be reliable (vulnerability detection) deterministic and auditable, while confining the LLM's role to the parts that benefit from language flexibility (explanation, prioritization, fix suggestion). This separation of concerns is the core architectural argument of this paper, and mirrors the broader industry shift toward coordinated multi-agent orchestration over single-agent designs [10].

D. Research Objectives

- Design a multi-agent pipeline that isolates deterministic vulnerability detection from LLM-based reasoning.
- Define a standardized inter-agent and agent-to-tool communication protocol suitable for CI environments.
- Design a memory architecture that lets the reasoning agent draw on prior review history without retraining.
- Propose a reproducible, tagged evaluation protocol for benchmarking agentic security-review reliability.

E. Contributions of This Paper

1. **1. Architecture:** A five-agent pipeline (Orchestrator, three deterministic specialists, LLM Reasoning Agent) with an explicit division of labor between detection and reasoning.
2. **2. Communication layer:** An MCP-based agent-to-tool protocol replacing bespoke API integrations, detailed in Section VI.
3. **3. Memory system:** A three-tier (short-term, long-term, vector) memory architecture supporting retrieval-augmented reasoning.
4. **4. Evaluation protocol:** A tagged, reusable case-study template and benchmark-based evaluation methodology suited to reliability reporting for agentic tools.

II. LITERATURE REVIEW

A. Existing Multi-Agent Architectures

The multi-agent LLM ecosystem has consolidated rapidly around a small set of orchestration philosophies: LangGraph's explicit stateful graph, CrewAI's role-based team abstraction, and AutoGen's conversational multi-turn pattern, among others [10]. Each answers the coordination question differently — a graph, a team, or a conversation — and this choice ramifies through state management, token cost, and failure recovery [10].

B. Single-Agent vs. Multi-Agent Approaches

Single-agent LLM reviewers are simpler to deploy but conflate all responsibilities into one reasoning pass. Bugdar augments a single reviewing agent with Retrieval-Augmented Generation over project documentation [2]. RepoAudit instead adopts a multi-agent, repository-level auditing pipeline built on tree-sitter parsing and inter-procedural data-flow analysis [3]. AutoReview studies security-issue-oriented multi-agent review directly and finds that confirmation bias measurably lowers detection accuracy for both interactive and autonomous review agents [1].

C. Limitations of Current Systems

- Systems that rely on the LLM for both detection and explanation inherit the LLM's inconsistency across repeated runs on identical input [1].
- Point-to-point API integrations (one-off wrappers per tool) do not generalize across code hosts and complicate maintenance as the toolset grows.
- Few systems report run-to-run reliability or adversarial robustness; SEVRA-BENCH is among the first to test whether narrative framing of a PR can induce an agent to approve code that reintroduces known CVEs [12].

D. Research Gap

No system reviewed combines (i) a hard separation between deterministic detection and LLM reasoning, (ii) a standardized, protocol-based communication layer between agents and tools, and (iii) a structured, reusable reporting template for case studies and benchmark runs. This paper addresses that combined gap.

TABLE I COMPARISON WITH CLOSELY RELATED MULTI-AGENT CODE-REVIEW SYSTEMS

System	Coordination Style	Key Limitation Noted
Bugdar [2]	Single agent + RAG over project docs	Depends on quality/availability of project documentation
RepoAudit [3]	Multi-agent, tree-sitter + data-flow analysis	Designed for full-repo audits, not lightweight PR gating
AutoReview [1]	Multi-agent, security-issue-oriented review	Confirmation bias from PR framing lowers detection
Proposed System	5-agent pipeline; deterministic detection + MCP-mediated LLM reasoning	Deterministic core limits novel / zero-day pattern detection

III. BACKGROUND

A. Multi-Agent Systems

A multi-agent system (MAS) decomposes a task across multiple autonomous, communicating agents rather than one monolithic process. Coordination primitives — how agents discover each other, share state, and decide who acts next — are the core engineering problem in MAS design, distinct from single-agent prompt/tool design [10].

B. Large Language Models as Reasoning Agents

In this architecture, the LLM is used strictly as a reasoning component: given structured findings and retrieved context, it produces explanation, severity, and remediation text. It is not used as the primary vulnerability detector, which keeps the highest-stakes decision (is this exploitable?) deterministic and testable.

C. Agent Roles and Responsibilities

Five roles are defined in this system: Orchestrator (task decomposition and routing), Static Analysis Agent, Dependency CVE Agent, Secret Scanning Agent (all deterministic specialists), and the LLM Reasoning Agent (explanation and prioritization). Each has a single, narrowly scoped responsibility, consistent with the role-based design philosophy used in frameworks such as CrewAI [10].

D. Communication Protocols

Two coordination layers are relevant to agentic systems: an orchestration layer (how agents within one system hand off tasks) and a tool-access layer (how any agent calls external tools/data). This paper uses the Model Context Protocol (MCP) for the latter — a JSON-RPC-based client-server standard that decouples an agent from any specific tool implementation [6], [8]. Current MAS literature treats these as two architecturally distinct layers that should not be collapsed into one [10].

E. Memory: Short-Term, Long-Term, and Vector Memory

Short-term memory holds the active PR diff and conversation buffer for the current run. Long-term memory persists developer-confirmed outcomes (accepted fixes, dismissed false positives) across runs. Vector memory stores dense embeddings of past vulnerability patterns and coding-standard documents, retrieved via similarity search and injected into the reasoning agent's context — the retrieval-augmented generation (RAG) pattern introduced by Lewis et al. [11].

F. Planning and Reasoning

The Orchestrator performs lightweight planning — deciding which specialist agents a given PR requires based on changed file

types — rather than open-ended multi-step planning. This is a deliberate scope restriction: broader planning is delegated to the LLM Reasoning Agent only after deterministic findings already exist, limiting the blast radius of any single planning error.

IV. PROPOSED ARCHITECTURE

Figure 1 shows the overall multi-agent pipeline. The system triggers on every pull_request event and executes as a stateless GitHub Action composed of five cooperating agents.

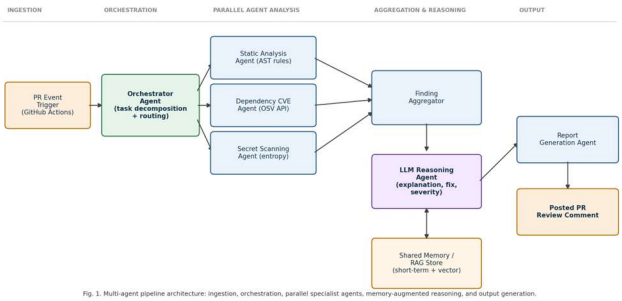

Fig. 1. Multi-agent pipeline architecture: ingestion, orchestration, parallel specialist agents, memory-augmented reasoning, and output generation.

Fig. 1. Multi-Agent Pipeline Architecture — Ingestion, Orchestration, Parallel Specialist Agents, Memory-Augmented Reasoning, and Output Generation

A. Agent Definitions

- **Orchestrator Agent —** fetches the PR diff via an MCP-exposed GitHub server, classifies changed files, and dispatches only relevant specialist agents.
- **Static Analysis Agent —** parses changed files into an Abstract Syntax Tree (AST) and matches insecure patterns (SQL string concatenation, eval/exec, insecure deserialization).
- **Dependency CVE Agent —** extracts manifest dependencies and queries the OSV database for known vulnerabilities [9].
- **Secret Scanning Agent —** applies regex and Shannon-entropy checks to flag likely hardcoded credentials.
- **LLM Reasoning Agent —** consumes aggregated findings plus retrieved memory context and produces explanation, fix suggestion, and confidence-adjusted severity.

B. Agent Workflow and Orchestrator Design

The Orchestrator implements task decomposition as a routing table: file extension and path patterns map to a subset of specialist agents, so a PR that only touches a requirements file never invokes the static analyzer, and vice versa. This keeps per-PR latency and LLM token cost proportional to what actually changed, rather than re-running the full pipeline unconditionally.

C. Routing Mechanism

Routing is rule-based rather than LLM-decided at this stage, so that which detectors run on a given diff is deterministic and auditable — an LLM is not in the loop until candidate findings already exist.

D. Tool Usage

All external actions — reading repository content, posting comments, querying CVE data, reading/writing memory — are exposed as MCP tools rather than being hard-coded into agent logic. This means an agent's code does not change if, for example, the CVE data source is swapped from OSV to a different provider; only the corresponding MCP server changes.

E. Shared Memory and Knowledge Retrieval (RAG)

The LLM Reasoning Agent is the only agent with read/write access to the three-tier memory system detailed in Section VII. Before generating an explanation, it issues a similarity-search query against vector memory to retrieve prior instances of the same vulnerability class, which is used to keep severity ratings and fix suggestions consistent across PRs over time.

F. Feedback Loop

When a developer's fix commit resolves a flagged finding, the outcome (fixed / dismissed-as-false-positive / disputed) is written back to long-term memory, closing the loop shown in Figure 2 and gradually improving the relevance of future retrieval.

G. Error Handling

Each specialist agent call is wrapped with a timeout and a fallback: if the Dependency CVE Agent's external query fails, the Orchestrator marks that check as "unavailable" in the final report rather than blocking the pipeline, and the LLM Reasoning Agent is explicitly instructed not to fabricate a finding for a check that did not complete.

V. AGENT INTERACTION WORKFLOW

Figure 2 traces one complete review cycle from commit push to developer remediation, including the memory read/write calls and the loop that re-triggers the pipeline on the fix commit.

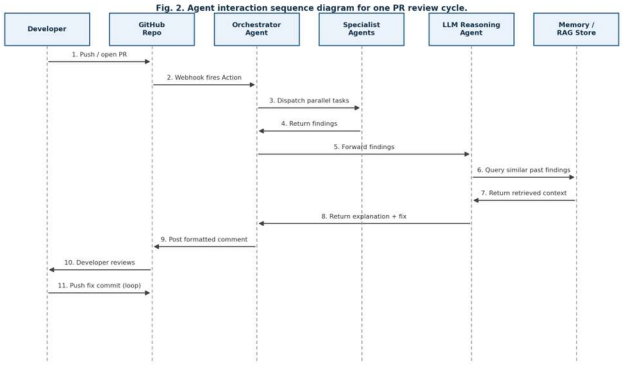

Fig. 2. Agent Interaction Sequence Diagram for One PR Review Cycle

VI. AGENT COMMUNICATION SYSTEM

This section describes, in detail, how the agents defined in Section IV exchange messages and tool calls, since this is the

layer that most differentiates the proposed design from bespoke, per-tool integrations common in earlier systems [2], [3].

A. Protocol Choice: Model Context Protocol (MCP)

All agent-to-tool communication in this system uses MCP, a JSON-RPC 2.0-based client-server protocol originally introduced by Anthropic [8] and since analyzed at the ecosystem level for its security and coordination properties [6]. Each agent acts as an MCP client; each external capability (GitHub API, OSV lookup, memory store, LLM provider) is exposed behind an MCP server. Figure 3 shows the resulting message topology.

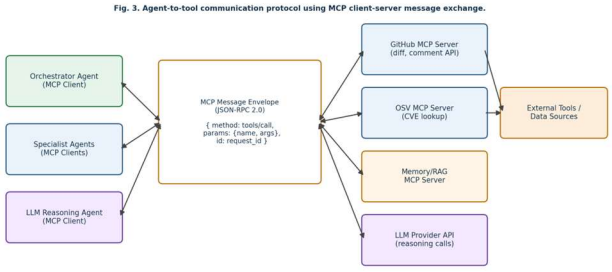

Fig. 3. Agent-to-Tool Communication Protocol Using MCP Client-Server Message Exchange

B. Message Format

Every call follows the JSON-RPC envelope: a method field (tools/call), a params object naming the target tool and its arguments, and a request id used to match asynchronous responses. This uniform envelope is what allows the Orchestrator, specialist agents, and the LLM Reasoning Agent to share one client implementation rather than three.

C. The Agents Used in This System and Their Communication Roles

- **Orchestrator Agent —** MCP client to the GitHub MCP server only; issues read (diff, file contents) and write (post comment) calls.
- **Static Analysis, Dependency CVE, and Secret Scanning Agents —** stateless MCP clients invoked by the Orchestrator with a file/diff payload; the CVE agent additionally calls the OSV MCP server, and all three return structured JSON findings, not natural language.
- **LLM Reasoning Agent —** the only agent that calls both the Memory/RAG MCP server (read and write) and the LLM Provider MCP server; it is the sole point where retrieved context and generative reasoning combine.
- **Memory/RAG MCP Server —** not an autonomous agent, but a passive MCP server providing similarity search over vector memory and key-value access to long-term memory, described further in Section VII.

D. Synchronous vs. Asynchronous Exchange

Specialist-agent calls (Section IV-A) are issued in parallel and awaited synchronously by the Orchestrator, since PR review latency is dominated by the slowest specialist rather than by sequential accumulation. The LLM Reasoning Agent's memory-retrieval call and subsequent reasoning call are sequential, since retrieved context is a direct input to the reasoning prompt.

E. Failure and Retry Semantics

MCP calls are retried once with exponential backoff on transport-level failure; a second failure is surfaced as a structured error to the Orchestrator rather than silently dropped, consistent with the error-handling design in Section IV-G.

VII. MEMORY ARCHITECTURE

Figure 4 details the three-tier memory system introduced in Section IV-E. Separating memory by volatility and access pattern — rather than using one undifferentiated context window — keeps the LLM Reasoning Agent's prompt small while still giving it access to a much larger effective history.

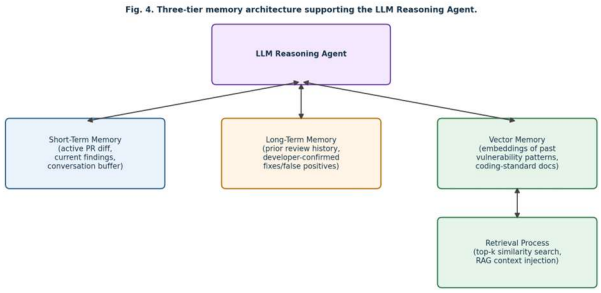

Fig. 4. Three-Tier Memory Architecture Supporting the LLM Reasoning Agent

A. Short-Term Memory

Holds the active PR diff, the current run's findings, and the in-progress reasoning conversation buffer. Discarded at the end of each run.

B. Long-Term Memory

A structured key-value store of prior review outcomes per repository: which findings were fixed, which were dismissed as false positives, and any developer-supplied justification. Persists indefinitely and is queried by exact repository/rule-id key.

C. Vector Memory

Dense embeddings of historical vulnerability findings and relevant coding-standard excerpts, queried by similarity rather than exact key, implementing the retrieval-augmented generation pattern [11]. This is what allows the reasoning agent to say, in effect, "this matches a pattern already resolved in this repository three weeks ago."

VIII. ALGORITHMS

A. Agent Selection Algorithm

Given a PR's changed-file list F, the Orchestrator computes the specialist set S = { a in Agents : match(a.filePattern, F) != empty }, so only agents relevant to the diff are invoked.

```
def select_agents(changed_files):
    selected = set()
    for agent in SPECIALIST_AGENTS:
        if any(agent.pattern.match(f)
               for f in changed_files):
            selected.add(agent)
    return selected
```

B. Task Scheduling

Selected specialist agents are scheduled concurrently (asyncio.gather-equivalent), bounded by a per-agent timeout; the Orchestrator proceeds to aggregation once all agents have either returned or timed out.

C. Inter-Agent Communication

As detailed in Section VI, all inter-agent data exchange is mediated through MCP tool calls rather than direct function calls or shared mutable state, which keeps agents independently testable and replaceable.

D. Conflict Resolution

Conflicts arise when two specialist agents report overlapping findings on the same line with different severities (e.g., the static analyzer flags a line as High while a future linting agent flags it as Low). The Finding Aggregator resolves this by severity-max: the highest reported severity for a given (file, line) pair is retained, and all contributing findings are passed to the LLM Reasoning Agent for a unified explanation rather than being deduplicated silently.

E. Consensus Mechanism

Because the current specialist set is deterministic (rule-based) rather than independently-sampled LLM judges, a voting consensus mechanism is not required for detection. Consensus becomes relevant only in the proposed evaluation protocol (Section X), where run-to-run agreement of the LLM Reasoning Agent's severity assignment is itself a measured reliability metric.

IX. IMPLEMENTATION

A. Technologies Used

- Language/runtime: Python 3.11 for all agent logic.
- Orchestration: a lightweight custom asyncio scheduler; LangGraph or CrewAI are viable drop-in alternatives given their native support for the graph/role patterns described in Section III-A [10].
- Communication: MCP (JSON-RPC 2.0) for all agent-to-tool calls [6], [8].
- Static analysis: Python's built-in ast module.
- Vector memory: any embedding-similarity store (e.g., a managed vector database or self-hosted FAISS-backed service).
- CI integration: GitHub Actions YAML workflow, triggered on pull_request.

B. APIs

- GitHub REST API (diff retrieval, comment posting) — wrapped as an MCP server.
- OSV API (osv.dev) for dependency CVE lookup [9] — wrapped as an MCP server.
- LLM provider API for the reasoning agent — wrapped as an MCP server.

C. Databases

- Structured findings log — append-only JSON store, functioning as the system's audit trail.
- Long-term memory store — key-value store keyed by (repository, rule-id).
- Vector memory store — embedding index over historical findings and coding-standard text.

D. Deployment Architecture

Figure 5 shows the full deployment: the GitHub Actions runner hosts the agent container for the duration of the PR check; all persistent state (findings log, memory stores) lives outside the ephemeral runner, so state survives across runs even though the compute does not.

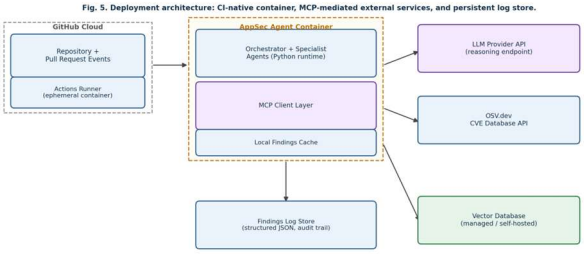

Fig. 5. Deployment Architecture — CI-Native Container, MCP-Mediated External Services, and Persistent Log Store

X. EXPERIMENTAL SETUP

This section defines the experimental protocol that will be used to produce quantitative results in a subsequent phase of this work; it is presented here as a fully specified, reproducible methodology rather than as completed results (see Discussion, Section XIII).

A. Datasets

Two intentionally vulnerable benchmark applications with documented ground-truth vulnerabilities: OWASP Juice Shop and DVWA (Damn Vulnerable Web Application). Both provide

a known, enumerable vulnerability set against which agent findings can be scored.

B. Test Scenarios

- Direct injection — vulnerable code introduced in a single-commit PR (as in the Section XI case study).
- Obfuscated injection — the same vulnerability class introduced with variable renaming and added indirection, to test pattern-matching robustness.
- Adversarial framing — a PR description that argues the change is safe or intentional, to test susceptibility to confirmation bias, following the methodology used in SEVRA-BENCH [12].
- Clean PR (negative control) — no injected vulnerability, to measure the false-positive rate.

C. Baselines

- Rule-only baseline — the deterministic specialist agents with no LLM reasoning layer.
- Single-agent LLM baseline — one LLM call given the raw diff, no specialist decomposition.
- Published reference systems — Bugdar [2] and AutoReview [1] as reported in their respective papers, for indirect comparison.

D. Evaluation Methodology

Figure 6 outlines the evaluation loop: running the full pipeline repeatedly against each test scenario, logging every finding, and comparing against the documented ground truth to compute the metrics defined in Section XI.

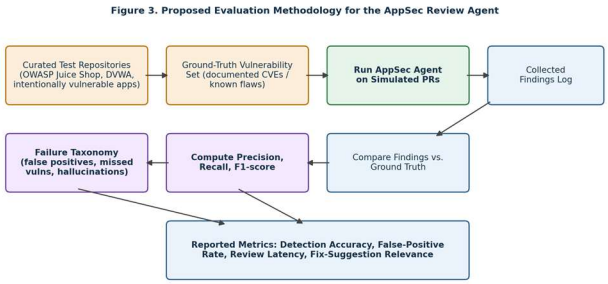

Fig. 6. Proposed Evaluation Methodology for the AppSec Review Agent

XI. EVALUATION METRICS

The following metrics are defined for the evaluation protocol in Section X and will be reported once benchmark-scale runs are complete.

TABLE III EVALUATION METRICS FOR THE PROPOSED MULTI-AGENT PIPELINE

Metric	Definition	Purpose
Accuracy	Percentage of correctly identified vulnerability instances	Overall performance
Precision	True positives / total predicted positives	Detection reliability
Recall	True positives / total actual positives	Detection completeness
F1-score	Harmonic mean of precision and recall	Balanced summary
Task Completion Rate	% of PRs for which the pipeline returns a complete report without error	Robustness
Latency	End-to-end time from PR event to posted comment	CI feasibility
Cost	LLM tokens consumed per PR review	Operating expense
Scalability	Latency/cost trend as PR size and repo count increase	Production readiness
Resource Utilization	CPU/memory of the CI runner during a review	CI runner sizing

XII. RESULTS

Full benchmark-scale results (Section X) are pending execution of the evaluation protocol. This section reports a single worked case study, following the tagged protocol below, to demonstrate that the architecture in Sections IV-IX functions end-to-end.

A. Tagged Case Study Protocol

Each case study is recorded against a fixed tag template — [OBJECTIVE], [ENVIRONMENT], [INPUT ARTIFACT], [DETECTED OUTPUT], [INTERPRETATION] — so that future case studies remain directly comparable as the system and benchmark corpus grow.

[OBJECTIVE] Verify end-to-end detection and explanation of a classic SQL-injection pattern in a simulated PR (Test Scenario: Direct Injection, Section X-B).

[ENVIRONMENT] Local GitHub Actions runner, Python 3.11, Flask 3.x sample application.

[INPUT ARTIFACT] A deliberately vulnerable Flask route submitted as a test PR:

```
@app.route('/user')
def get_user():
    uid = request.args.get('id')
    query = "SELECT * FROM users WHERE id = " + uid
    return db.execute(query)
```


TABLE IV [DETECTED OUTPUT] — DETERMINISTIC FINDINGS FOR CASE STUDY CS-01

Agent	Finding	Severity
Static Analysis Agent	Unsanitized user input concatenated directly into SQL query (line 4)	High
Static Analysis Agent	No input validation on 'id' query parameter	Medium
Secret Scanning Agent	No secrets detected in diff	—
Dependency CVE Agent	No new/changed dependencies in this PR	—

[INTERPRETATION] The vulnerability was caught deterministically before any LLM call. The LLM Reasoning Agent then retrieved a similar prior finding from vector memory and produced an explanation plus a parameterized-query fix suggestion. This confirms the intended division of labor: detection is deterministic (Sections IV-A, VIII); the LLM contributes explanation and remediation only (Section IV-E).

B. Comparative Analysis (Planned)

Once Section X is executed, Table 5 (planned) will report precision/recall/F1 for the proposed system against the rule-only and single-agent LLM baselines across all four test scenarios, isolating the contribution of the multi-agent decomposition itself.

C. Ablation Study (Planned)

The ablation plan removes one architectural component at a time and re-measures the metrics in Section XI: (i) remove the LLM Reasoning Agent (detection-only pipeline) to isolate its contribution to fix-suggestion quality; (ii) remove vector memory to isolate RAG's contribution to explanation consistency; (iii) remove the Orchestrator's routing logic (run all specialists on every PR) to measure the latency/cost cost of unconditional execution. Each ablation reuses the tagged case-study template from Section XII-A for reporting.

XIII. DISCUSSION

A. Strengths

- Deterministic detection core limits the LLM's influence to explanation and triage, reducing exposure to the framing-based bias reported in agentic reviewers [1], [12].
- MCP-based communication (Section VI) decouples every agent from any single tool implementation, matching current agentic-tooling conventions [6], [8], [10].
- The three-tier memory system (Section VII) gives the reasoning agent effectively unbounded history without growing its prompt.

B. Limitations

- Rule-based static analysis cannot detect novel or logic-level vulnerabilities outside its predefined pattern set.
- Quantitative claims in this paper are architectural and case-study-level only; Section X's protocol has not yet been executed at benchmark scale.
- LLM-generated fix suggestions require human verification before merge; the system is a review aid, not an autonomous remediation authority.

C. Failure Cases

Anticipated failure modes, to be measured against the taxonomy in Section X: missed vulnerabilities outside the static analyzer's rule set; hallucinated findings introduced by the LLM Reasoning Agent despite its structured-output constraint; and severity misclassification when the aggregator's severity-max rule (Section VIII-D) over- or under-weights a low-confidence specialist finding.

D. Security and Privacy Considerations

Because the agent runs inside a GitHub Actions workflow and communicates via MCP, it inherits both the CI injection-vulnerability surface documented for agentic workflows generally [5] and the tool-poisoning risks documented for MCP deployments specifically [6]. Long-term memory (Section VII-B) persists code-derived content across runs, so access to the memory store must be scoped per repository to avoid cross-tenant leakage in a multi-repository deployment.

XIV. FUTURE WORK

- **Self-learning agents** — using long-term memory outcomes (Section VII-B) as a training signal to automatically adjust specialist agent confidence over time.
- **Dynamic agent creation** — spawning a narrowly scoped specialist agent on demand for a language or framework not covered by the current static analyzer, rather than pre-registering every possible specialist.
- **Adaptive planning** — letting the Orchestrator's routing (Section IV-C) be learned from historical agent-selection outcomes rather than fixed rule-based patterns.
- **Human-in-the-loop collaboration** — an explicit approve/dismiss action on posted findings that writes directly into long-term memory (Section VII-B), tightening the feedback loop described in Section IV-F.

XV. CONCLUSION

This paper presented a multi-agent architecture for security-focused pull request review that separates deterministic vulnerability detection from MCP-mediated, memory-augmented LLM reasoning. The research gap addressed is the absence, in prior systems, of a combined design offering (i) hard separation between detection and reasoning, (ii) a standardized communication protocol across agents and tools, and (iii) a reusable tagged protocol for reporting case studies and benchmark runs consistently. The architectural contribution — five cooperating agents, a three-tier memory system, and an MCP-based communication layer — is presented with a

complete system architecture diagram, agent interaction sequence diagram, communication protocol diagram, memory architecture diagram, and deployment architecture diagram (Figures 1-5), together with one worked case study. Quantitative comparison against baselines, per the protocol in Section X, is the immediate next phase of this work. Potential applications extend beyond GitHub PR review to any CI-adjacent review task — infrastructure-as-code review, dependency-update triage, or license-compliance checking — that benefits from the same deterministic-detection-plus-LLM-reasoning split.

REFERENCES

[1] "AutoReview: An LLM-based Multi-Agent System for Security Issue-Oriented Code Review," 2025.

[2] "Bugdar: AI-Augmented Secure Code Review for GitHub Pull Requests," arXiv:2503.17302, 2025.

[3] J. Guo, C. Wang, X. Xu, Z. Su, and X. Zhang, "RepoAudit: An Autonomous LLM-Agent for Repository-Level Code Auditing," in Proc. 42nd Int. Conf. Machine Learning (ICML), 2025.

[4] "Security in the Age of AI Teammates: An Empirical Study of Agentic Pull Requests on GitHub," arXiv:2601.00477, 2026.

[5] "Demystifying and Detecting Agentic Workflow Injection Vulnerabilities in GitHub Actions," arXiv:2605.07135, 2026.

[6] X. Hou, Y. Zhao, S. Wang, and H. Wang, "Model Context Protocol (MCP): Landscape, Security Threats, and Future Research Directions," arXiv:2503.23278, 2025.

[7] Snyk, "DeepCode AI: AI-Powered Static Analysis Engine," Snyk Documentation, accessed 2026.

[8] Anthropic, "Introducing the Model Context Protocol," Anthropic News, Nov. 2024.

[9] Google Open Source Security Team, "OSV — Open Source Vulnerability Database," osv.dev, accessed 2026.

[10] "LLM-Based Multi-Agent Orchestration: A Survey of Frameworks, Communication Protocols, and Emerging Patterns," Future Internet, 2026, doi:10.3390/fi18060326.

[11] P. Lewis et al., "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," in Proc. NeurIPS, 2020, pp. 9459-9474.

[12] "SEVRA-BENCH: Social Engineering of Vulnerabilities in Review Agents," arXiv:2606.13757, 2026.

