A Multi-Agent DevSecOps Framework for Intelligent Vulnerability Detection and Auto-Remediation

Haining Fan[1], Lijie Zheng[1], Chenhao Han[1], Ji He[1,*], and Chunlin He[2]
[1]School of Computer Science and Technology, Xidian University, Xi'an, Shaanxi, 710071, China
[2]Chaoyue Technology Co., Ltd., Shandong Province Key Laboratory of Independent and Reliable Computing Technology and Equipment, Jinan, 250104, China

This paper presents a multi-agent DevSecOps framework that integrates static code scanning, large language model (LLM) based security reasoning, automated repair generation, policy-as-code enforcement, and runtime monitoring into a unified event-driven pipeline. Five specialized agents collaborate through LangGraph shared state graphs: a Code Security Agent combining Semgrep rule matching with LLM contextual review, a Fix Agent generating reviewable candidate patches, a Policy Agent producing OPA Rego and Kubernetes NetworkPolicy files, and an Enforcement Agent operating in both CI gate and runtime response modes. Evaluation on a test application containing 50 planted vulnerabilities across Python code and infrastructure-as-code demonstrates that the combined Semgrep+LLM detection achieves 92.0% recall (F1=95.8%), compared to 34.0% for Semgrep alone, with zero false positives under the manually labeled test oracle. The Fix Agent commits candidate patches for 98.4% of detected vulnerabilities at an average of 35.5 seconds each. The Enforcement Agent correctly blocks non-compliant configurations and completes CI gate decisions in under 34 seconds. Runtime monitoring detects injection attacks, brute-force attempts, and unauthorized access with risk-proportional automated response within 42 seconds.

Index Terms—**DevSecOps, multi-agent systems, vulnerability detection, auto-remediation, Security as Code, large language models.**

I. Introduction

DEVOPS has become the dominant paradigm for modern software delivery, unifying development and operations through continuous integration, continuous delivery (CI/CD), and Infrastructure as Code (IaC) [1], [2]. By automating build, test, and deployment pipelines, organizations now routinely push code changes to production multiple times per day, managing cloud-native microservice architectures at a scale and velocity that would be impossible with manual processes.

However, the speed and automation that make DevOps effective also introduce significant security risks. Rapid release cycles can outpace traditional security reviews, and infrastructure defined in code can propagate misconfigurations across entire environments within minutes. DevSecOps addresses this challenge by shifting security left—embedding security controls into every stage of the software development lifecycle rather than treating security as a post-development gate [3], [4]. A central practice within DevSecOps is Security as Code (SaC), which codifies security policies, compliance checks, and configuration hardening rules so that they are version-controlled and automatically enforced in CI/CD pipelines [5], [6]. By making security a shared responsibility among developers, operations, and security teams, DevSecOps aims to detect and remediate vulnerabilities before they reach production, thereby reducing the attack surface and the cost of late-stage fixes. Despite its promise, practical DevSecOps adoption remains challenging: organizations struggle with fragmented security tooling, complex policy management, configuration drift, and developer resistance to additional security processes [7].

Manuscript received March 17, 2026; revised June 10, 2026. Corresponding author: Ji He (email: jihe@xidian.edu.cn).

Traditional approaches to DevSecOps security automation rely primarily on rule-based static analysis tools and manual processes. Static Application Security Testing (SAST) tools such as Semgrep [8] and CodeQL [9] scan source code against predefined vulnerability patterns, detecting common issues like SQL injection, cross-site scripting, and hard-coded secrets. Open Policy Agent (OPA) [10] and HashiCorp Sentinel [2] enable policy-as-code enforcement for infrastructure and deployment configurations. While these tools provide reliable pattern matching and consistent enforcement, they suffer from well-known limitations: high false-positive rates due to lack of semantic understanding, inability to detect context-dependent vulnerabilities that require reasoning about application logic, and a fundamental disconnect between detection and remediation—findings are reported, but fixes must still be crafted manually by developers. Furthermore, security policies must be written and maintained by specialists, creating bottlenecks in organizations where security expertise is scarce.

Recent advances in large language models (LLMs) have opened new possibilities for DevSecOps automation [11]. LLMs can interpret code semantics, explain security findings in natural language, and generate code patches or policy configurations—capabilities that complement the pattern-matching strengths of traditional tools. Researchers have explored LLM-based vulnerability detection through prompt engineering techniques [12], automated vulnerability repair through natural-language advisory generation [13] and self-healing software pipelines [14], and security policy generation and compliance checking [15]. However, a single LLM struggles to cover the full breadth of security tasks—code-level analysis, policy reasoning, and runtime monitoring demand different expertise and context, and no single model excels at all of them. Multi-agent systems (MAS) address this by

decomposing complex workflows into specialized, collaborating roles, as demonstrated by ChatDev [16], MetaGPT [17], and AutoGen [18] in software engineering. In the security domain, multi-agent architectures further enable defense-in-depth through specialized agents that each focus on a distinct aspect of the security lifecycle [19], [20].

Despite these advances, several gaps remain that motivate our work. First, existing LLM-based security approaches typically address isolated tasks—detection, repair, or policy generation—without integrating them into a unified, end-to-end workflow that spans from code commit to production monitoring. Second, most approaches rely on a single LLM, missing the opportunity to leverage complementary strengths of different models (e.g., code generation versus logical reasoning). Third, the critical question of how to balance AI-driven automation with human oversight in security-critical decisions has received limited attention. Even the most complete multi-agent effort to date—Vakhula and Opirskyy's AI-driven Security as Code framework [6], which deploys four collaborative agents for code scanning, policy generation, enforcement, and monitoring—leaves the remediation stage entirely manual: once a vulnerability is detected, developers must still craft fixes by hand. More broadly, existing multi-agent security architectures lack automated repair capabilities that close the detection-to-remediation loop, rely on a single LLM without exploiting complementary model strengths, and use ad-hoc inter-agent communication rather than structured, auditable state management.

To address these gaps, we propose a multi-agent DevSecOps framework that unifies vulnerability detection, automated repair, policy generation, enforcement, and runtime monitoring into a single event-driven pipeline. Agents collaborate through a shared state graph managed by LangGraph [21], where each agent reads from and writes to a centralized typed state that captures the full lifecycle of a security analysis run, eliminating the need for external message queues or databases. The detection stage combines LLM-based semantic reasoning with Semgrep's rule-based pattern matching, allowing the system to catch both well-known vulnerability patterns and context-dependent issues that pure pattern matching would miss. A tiered human-in-the-loop mechanism routes decisions through different approval levels based on risk severity, enabling low-risk actions to proceed automatically while escalating high-impact changes for human review. Specifically, this paper makes the following contributions:

1) We propose a unified multi-agent DevSecOps framework that integrates the full security lifecycle—from static analysis through policy enforcement to runtime monitoring—into a cohesive, automated Security-as-Code workflow, eliminating the manual handoffs that fragment existing approaches.
2) We design a shared-state collaboration mechanism based on LangGraph, where agents coordinate through a centralized typed state graph rather than point-to-point messaging, and a hybrid detection approach that combines LLM semantic reasoning with rule-based static analysis to improve vulnerability identification.
3) We introduce a tiered automation model that classifies security decisions by risk level and applies graduated human oversight accordingly, balancing automation throughput with accountability for high-impact changes.
4) We evaluate the framework on a test application with 50 planted vulnerabilities across Python code and infrastructure-as-code, demonstrating 92.0% detection recall with zero false positives under a manually labeled oracle, 98.4% candidate patch commit rate, and end-to-end pipeline completion within CI/CD time budgets.

II. Related Work

This section reviews existing research along the three technical pillars that underpin our framework: vulnerability detection and automated repair, security policy automation, and multi-agent systems for software engineering.

A. Vulnerability Detection and Automated Repair

Static Application Security Testing (SAST) tools such as Semgrep [8] and CodeQL [9] form the first line of defense in DevSecOps pipelines, scanning source code against predefined vulnerability patterns. However, empirical studies have revealed fundamental limitations of rule-based approaches. Charoenwet et al. [22] found that at least 76% of SAST warnings in vulnerable functions are irrelevant to actual vulnerabilities, and that a single tool detects warnings in only about half of vulnerable functions. These high false-positive rates and limited coverage stem from the inability of pattern-matching rules to reason about application-specific semantics.

Recent work has turned to large language models to address these limitations. Wen et al. [23] proposed SCALE, which constructs structured natural language comment trees to capture cross-function data flows, improving detection accuracy over traditional static analysis. Du et al. [24] developed a multi-task instruction fine-tuning approach that generalizes vulnerability detection across programming languages. Bae et al. [12] demonstrated that carefully designed prompts enable LLMs to identify security weaknesses that pattern-matching tools overlook. Yet LLM-based detection introduces its own challenges: Zibaeirad and Vieira [25] showed that individual LLMs struggle with complex vulnerability identification, and that while ensemble approaches improve detection by 10–12%, they introduce trade-offs between false positives and false negatives that require careful threshold calibration.

On the remediation side, Zhang et al. [13] introduced VulAdvisor, which generates natural language repair suggestions, and Charalambous et al. [14] explored self-healing pipelines that combine LLMs with formal verification. Li et al. [26] proposed PatchFinder for tracing security patches to disclosed vulnerabilities. Despite these advances, a fundamental disconnect persists: SAST tools report findings but leave remediation to developers, while LLM-based repair systems operate on pre-identified vulnerabilities rather than discovering them. Detection and repair remain separate activities with no automated handoff between them. Our framework bridges this gap by integrating detection and repair into a continuous pipeline where identified vulnerabilities are automatically routed to a specialized repair agent.

B. Security Policy Automation

Security as Code (SaC) codifies security policies so that they can be version-controlled and automatically enforced in CI/CD pipelines [5], [6]. Policy-as-code frameworks such as Open Policy Agent [10] and HashiCorp Sentinel [2] provide the enforcement layer, while industry best practices recommend embedding policy checks at multiple pipeline stages [7], [27], [28]. Organizations adopting SaC report fewer misconfigurations and more streamlined deployments [3].

However, these frameworks only automate enforcement—the policies themselves must still be authored manually in domain-specific languages by engineers who understand both the security requirements and the target platform [1], [4]. This creates bottlenecks in organizations where such expertise is scarce, and policies are prone to drift as environments evolve. Recent work has explored using LLMs to bridge this gap. Gong et al. [15] investigated LLMs as end-to-end secure code agents, but found that models struggle with context-specific security requirements without human guidance. Gupta and Sreenivasamurthy [29] developed Prose2Policy, an LLM pipeline for translating natural-language access policies into executable Rego code, achieving a 95.3% compile rate but only an 82.2% test pass rate—highlighting the persistent gap between natural language intent and correct executable policy.

Existing approaches thus address either enforcement of manually authored rules or generation of individual policies in isolation. Neither integrates policy generation with the broader security workflow: policies are not informed by detected vulnerabilities, and enforcement results do not feed back into detection. Our framework closes this loop by using an LLM agent to generate policies from application context and detected vulnerabilities, then automatically enforcing them through OPA and Sentinel, with results feeding back to the monitoring stage.

C. Multi-Agent Systems for Software Engineering

Multi-agent systems decompose complex tasks into specialized, collaborating roles. LLM-based frameworks such as ChatDev [16], MetaGPT [17], and AutoGen [18] have demonstrated that distributing responsibilities across multiple agents yields better results than single-agent approaches on complex software engineering tasks.

In the security domain, multi-agent architectures enable defense-in-depth by assigning distinct security responsibilities to specialized agents. Kassimi et al. [19] proposed a multi-agent framework for big data security where agents handle authentication control and intrusion detection across distributed clusters. Chen et al. [20] developed AgentVerse, showing that specialized agents collaborating on tasks requiring diverse expertise outperform monolithic systems. Sarker et al. [30] surveyed AI-driven cybersecurity approaches, noting that multi-agent architectures are well suited to the distributed nature of modern security challenges.

However, existing multi-agent security systems share two key limitations. First, they primarily target network-level threats—intrusion detection, access control, and traffic monitoring—rather than the software development lifecycle where vulnerabilities originate. Second, they lack integration with development workflows: agents operate on deployed systems but are disconnected from the code analysis, policy generation, and CI/CD enforcement stages where preventive security is most effective. Our framework addresses both limitations by applying the multi-agent paradigm to DevSecOps, assigning agents to distinct lifecycle stages—code analysis, repair, policy generation, enforcement, and runtime monitoring—and coordinating them through a shared state graph to provide end-to-end coverage from code commit to production.

Table I summarizes the functional coverage of representative approaches. The comparison is qualitative because these systems target different tasks and are not evaluated on a common benchmark. It nevertheless clarifies the architectural gap addressed by this work: existing tools typically cover one or two stages of the security lifecycle, while the proposed framework connects detection, candidate repair generation, policy generation, CI enforcement, and runtime response through an auditable shared-state workflow.

III. Framework Design and Implementation

This section presents the design and implementation of our multi-agent DevSecOps framework. A central design principle is the separation of security concerns into two complementary phases that mirror the software lifecycle. Static code analysis operates at development time, catching vulnerabilities and generating security policies before code reaches production; dynamic runtime monitoring operates continuously in the production environment, detecting threats that only manifest under real workloads. This two-phase design reflects a defense-in-depth strategy: static analysis provides preventive security by shifting left, while runtime monitoring provides detective and reactive security for issues that escape pre-deployment checks—such as zero-day exploits, configuration drift, or attacks targeting application logic that static tools cannot model. By connecting both phases through shared state and a common agent architecture, the framework closes the feedback loop between development-time findings and production-time observations.

A. Overall Architecture

Figure 1 illustrates the overall architecture. The framework is implemented as a FastAPI service that receives webhook events from GitHub Actions. When a developer pushes code or opens a pull request on a monitored branch, the CI/CD workflow sends the commit metadata and Semgrep scan results to the framework's `/start` endpoint, which initializes the static analysis pipeline. A separate `/enforce` endpoint handles post-merge policy enforcement, and `/monitor` endpoints manage periodic runtime monitoring. This event-driven design ensures that security analysis is triggered automatically by development activity rather than relying on manual initiation, embedding security into the natural development workflow without requiring developers to change their practices.

Table II summarizes the implementation technologies for each component.

TABLE I
FUNCTIONAL COMPARISON WITH REPRESENTATIVE SECURITY AUTOMATION APPROACHES

Approach	Detection	Candidate Repair	Policy Generation	CI Gate	Runtime Monitoring	Coordination Model
SAST tools (Semgrep, CodeQL) [8], [9]	Rule-based	No	No	Partial	No	Tool output
Policy-as-code tools (OPA, Sentinel) [2], [10]	Config checks	No	Manual	Yes	Partial	Policy engine
LLM vulnerability detectors [12], [23], [24]	LLM-based	No	No	No	No	Single task
LLM repair systems [13], [14]	Assumed input	Suggestions or patches	No	No	No	Single task
AI-driven SaC MAS [6]	AI-assisted	Manual	Yes	Yes	Yes	Multi-agent messages
General MAS for software engineering [16]–[18]	Task dependent	Task dependent	No	No	No	Multi-agent dialogue
Our framework	**Semgrep + LLM**	**Yes**	**Yes**	**Yes**	**Yes**	**LangGraph shared state**

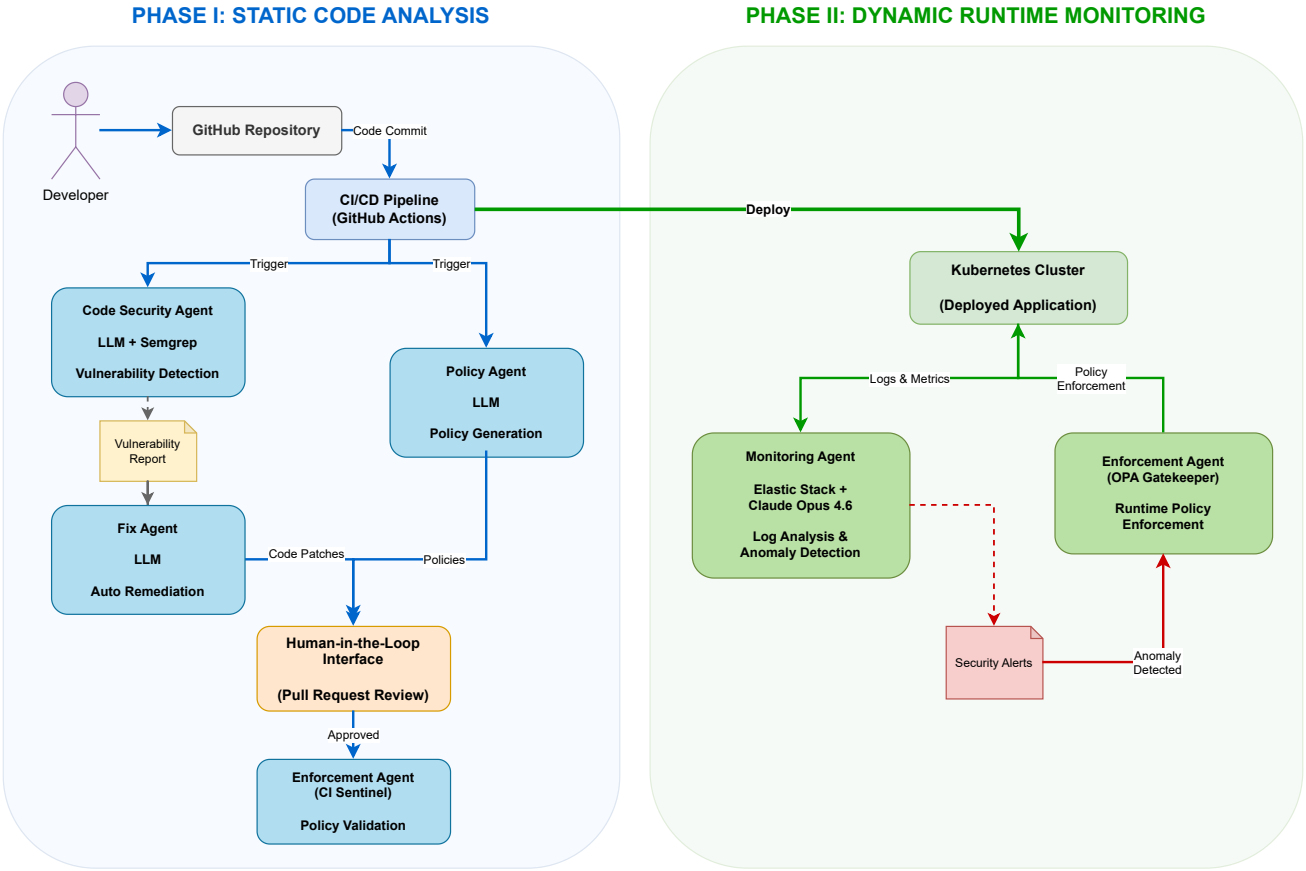

Fig. 1. Overall architecture of the multi-agent DevSecOps framework.

A key architectural decision is the use of multiple LangGraph [21] state graphs rather than a single monolithic pipeline. Security tasks in the static and runtime phases have fundamentally different triggering conditions, execution frequencies, and state requirements: static analysis runs once per commit with rich code context, CI enforcement runs once per merge with policy-focused state, and runtime monitoring runs periodically with telemetry data. Separating these into three independent graphs allows each to define a minimal, purpose-specific typed state dictionary, avoiding the complexity of a single state object that would need to accommodate all scenarios. It also enables independent scaling and failure isolation—a crash in the monitoring loop does not affect the static analysis pipeline.

TABLE II
IMPLEMENTATION TECHNOLOGIES

Component	Role	Technologies
Code Security Agent	Vulnerability detection via LLM + static analysis	LLM (ChatOpenAI), Semgrep, LangChain Agent
Fix Agent	Automated code patch generation	LLM, LangChain Agent, MemorySaver, GitPython
Policy Agent	Security policy generation	LLM, LangChain Agent, GitPython
Enforcement Agent (CI)	Pre-deployment policy enforcement	LLM, LangChain Agent (read-only tools)
Enforcement Agent (Runtime)	Runtime policy enforcement	LLM, kubectl, OPA Gatekeeper
Monitoring Agent	Runtime anomaly detection	Elasticsearch, LLM, LangChain Agent
Human-in-the-Loop	Review coordination via pull requests	PyGitHub, Git push
Orchestration	Agent coordination and state management	LangGraph StateGraph, FastAPI, APScheduler

The primary graph manages the static analysis pipeline through a `MasSecState` typed dictionary whose fields capture trigger metadata, repository paths, structured vulnerability findings, file edit records, policy artifacts, and human review status. Upon receiving a webhook event, the graph executes a CI/CD initialization node that clones two separate copies of the target repository—one for security fixes and one for policy generation—and creates dedicated working branches for each. This dual-clone strategy is deliberate: the Fix Agent and Policy Agent modify different files for different purposes, and isolating their changes on separate branches prevents merge conflicts and allows independent review. The Code Security Agent and Policy Agent then execute in parallel (since policy generation depends on infrastructure context, not vulnerability findings), followed by the Fix Agent (which depends on the Code Security Agent's output), and finally the Human-in-the-Loop node that creates pull requests. A second graph handles CI enforcement as an independent pipeline triggered when a pull request is merged to the main branch. A third graph manages runtime monitoring, with a conditional edge that routes to the Runtime Enforcement Agent only when the Monitoring Agent reports medium or higher risk—low-risk results terminate the graph immediately, avoiding unnecessary cluster operations.

Table III details the main information exchanged through the shared state. Agents do not directly overwrite each other's outputs; instead, each stage appends typed artifacts such as findings, commits, policies, verdicts, anomalies, and alerts. When outputs conflict, the framework applies conservative precedence rules: unresolved or inconsistent vulnerability findings remain visible in the review summary, policy or enforcement verdicts can block downstream deployment, and any ambiguous enforcement decision is mapped to `needs_review`. This design avoids silent overwrites while keeping the workflow lightweight enough for CI/CD use.

Each agent is implemented as a LangChain agent with tool-calling capabilities, following the ReAct (Reasoning + Acting) pattern [31]: at each step, the agent reasons about the current state, selects a tool to invoke, observes the result, and decides whether to continue or produce a final output. The LLM is accessed through a unified `ChatOpenAI` interface configured via environment variables, allowing deployment with different model providers. Temperature settings are tuned per agent: lower values (0.0) for enforcement and policy tasks requiring deterministic output, and slightly higher values (0.2–0.3) for code analysis and summarization tasks.

B. Phase I: Static Code Analysis

The static analysis phase embodies the "shift-left" principle of DevSecOps: by analyzing code at the point of commit rather than after deployment, vulnerabilities can be caught when they are cheapest to fix and before they propagate through the release pipeline. This phase decomposes the security analysis task into four specialized agents, each addressing a distinct concern that would be difficult for a single agent to handle well. The Code Security Agent combines rule-based and semantic detection to maximize coverage while controlling false positives. The Fix Agent translates detection results into concrete code patches, bridging the gap between finding a vulnerability and resolving it—a handoff that is typically manual and time-consuming. The Policy Agent shifts from reactive vulnerability fixing to proactive security governance by generating infrastructure-level policies that prevent entire classes of misconfiguration. The Enforcement Agent (CI mode) serves as a final safety gate, ensuring that no code reaches production without passing policy checks. Together, these agents form a pipeline that progresses from detection through remediation to prevention, with a Human-in-the-Loop node ensuring that all automated changes receive human approval before merging.

1) Code Security Agent

The Code Security Agent performs hybrid vulnerability detection by combining Semgrep's rule-based scanning [8] with LLM-based contextual analysis. The agent operates in three stages. First, it parses the Semgrep JSON results received from the CI/CD pipeline, extracting structured `Finding` objects that capture rule ID, message, file path, line range, severity, and metadata. Second, it launches a LangChain agent to review the changed code and discover vulnerabilities that Semgrep may have missed. Third, it generates an LLM-powered summary of all findings. Findings from Semgrep and the LLM review are merged with deduplication based on a composite key of rule ID, file path, line range, message, and severity.

TABLE III
INTER-AGENT INFORMATION FLOW THROUGH SHARED STATE

Agent	State Read	State Written	Used By
Code Security Agent	Commit metadata, changed files, Semgrep JSON	Deduplicated `Finding` objects with severity and evidence	Fix Agent and review summary
Fix Agent	Findings, source files, fix branch	Local commits, `fix_status`, edited file records	Human-in-the-loop review PR
Policy Agent	Changed files, IaC context, policy branch	OPA Rego, Kubernetes NetworkPolicy, policy commits	CI Enforcement Agent
Enforcement Agent (CI)	Merged repository, policy files, IaC files	`pass`, `block`, or `needs_review` verdict	Deployment gate and human review
Monitoring Agent	Elasticsearch logs and metrics	Log anomalies, resource anomalies, risk level	Runtime Enforcement Agent
Enforcement Agent (Runtime)	Risk level, anomalies, cluster state	NetworkPolicy/constraint actions and security alerts	Cluster state and operators

Code Security Agent

System Prompt

You are a code security review agent. Your goal is to review changed code and find high-value vulnerabilities that Semgrep may have missed. Focus on injection, authentication/authorization, sensitive data exposure, deserialization, command execution, and path traversal in new/modified code. Use `find_repo_files` to locate context, then `read_repo_file` to examine it. Only call `add_finding` when evidence is clear. Do not duplicate existing findings.

Tools

- `list_changed_files` — returns the list of files modified in the current commit or pull request.
- `read_changed_patch` — retrieves the diff patch for a specific changed file.
- `find_repo_files` — searches the repository for files matching a keyword, enabling the agent to locate related code such as authentication middleware or configuration files.
- `read_repo_file` — reads full file contents from the cloned repository for contextual analysis.
- `add_finding` — records a new vulnerability finding with structured fields (rule ID, message, path, line range, severity).

2) Fix Agent

The Fix Agent automatically generates code patches to remediate vulnerabilities identified by the Code Security Agent. It receives the list of `Finding` objects and processes each vulnerability individually in a separate LLM conversation to avoid context window saturation. The agent uses a `MemorySaver` checkpointer to maintain conversation history across findings within the same analysis run, allowing it to build cumulative understanding of the codebase.

After each finding is processed, the agent tracks whether a new commit was created by comparing commit counts and SHA values in the runtime state. Each finding's `fix_status` is updated to one of: `fixed` (commit created), `not_fixed` (agent ran but produced no commit), `skipped` (missing file path), or `failed` (execution error). All fixes are committed to a dedicated working branch; the agent is explicitly prohibited from pushing or creating pull requests, ensuring that all changes go through the Human-in-the-Loop review process.

Fix Agent

System Prompt

You are a code security fix assistant. All file operations and commits must be done through tools. Some vulnerabilities may span multiple files; read and modify related files as needed. If unfamiliar with the repository structure, use `list_directory`/`find_files` to locate files first. For each vulnerability, read necessary files, write complete fixed file contents, then create exactly one local commit (may include multiple files). Do not push or create pull requests.

Per-Finding Prompt

Process one vulnerability in a file.
Vulnerability information: {finding details}
Requirements: (1) Target file is {path}, but read/modify related files if the vulnerability involves cross-file context. (2) Use `list_directory`/`find_files` to locate related files if unsure. (3) Call `read_source_file` to read necessary files. (4) Call `write_source_file` with complete fixed content. (5) Create exactly one local commit via `commit_source_files`. (6) Minimize changes; fix only this vulnerability. (7) Do not push or create pull requests.

Tools

- `list_directory` — lists files and subdirectories under a repository-relative path.
- `find_files` — searches for files by keyword in their path.
- `read_source_file` — reads full file contents for understanding surrounding context.
- `write_source_file` — writes complete fixed file contents, with automatic stripping of markdown code fences.
- `commit_source_files` — stages specified files and creates a local Git commit.

3) Policy Agent

The Policy Agent automatically generates security policy files (Terraform IAM policies, Kubernetes NetworkPolicies) based on the application's infrastructure context. The agent first analyzes the changed files and repository structure to determine the policy focus: if the commit modifies `.tf`/`.tfvars`/`.hcl` files, it generates Terraform security policies; if it modifies Kubernetes manifests or the repository contains a `k8s/` directory, it generates Kubernetes NetworkPolicies. If neither infrastructure pattern is detected, the policy phase is skipped. The agent operates on a separate clone of the repository with its own working branch, preventing conflicts with the Fix Agent's changes.

Policy Agent

System Prompt

You are the Policy Generator Agent. Your role is to generate and update versionable security policy files (Security as Code). Based on code changes and context, output executable policies rather than issue lists. All file operations and commits must be done through tools. Only local commits are allowed; do not push or create pull requests.

Generation Prompt

Generate or update Security-as-Code policy files based on the current code changes.
Changed files: {file_list}. Focus: {terraform, kubernetes}.
Requirements: (1) Analyze relevant configurations using tools, output versionable policy files. (2) Place policies in `security/policies/` directory. (3) For Terraform: generate least-privilege IAM policies and secure default configurations. (4) For Kubernetes: generate/update NetworkPolicy (default-deny + minimal allow rules). (5) Create one local commit after modifications.

Tools

Same repository tools as Fix Agent: `list_directory`, `find_files`, `read_source_file`, `write_source_file`, `commit_source_files`.

4) Human-in-the-Loop Coordination

After the Code Security, Fix, and Policy Agents complete their work, the Human-in-the-Loop node consolidates results and creates GitHub pull requests for human review. It pushes the Fix Agent's working branch and creates a PR titled "[AUTO] Security Fix Review Required" with a structured body listing all findings grouped by status (fixed, pending, failed), including fix commit SHAs and affected file paths. If the Policy Agent generated new or updated policy files, a separate PR titled "[AUTO] Security Policy Update Review Required" is created from the policy working branch. Both PRs target the repository's default branch, ensuring that all automated changes require explicit human approval before merging.

5) Enforcement Agent (CI Sentinel Mode)

The Enforcement Agent operates as an independent pipeline triggered when a pull request is merged to the main branch. It clones the repository at the merge commit, scans for infrastructure and policy files (`.tf`, `.tfvars`, `.hcl`, `.yaml`, `.yml`) in security-related directories, and submits them to an LLM agent for review. The agent is restricted to read-only tools (`list_directory`, `find_files`, `read_source_file`) to prevent unintended modifications. The LLM's final response is parsed for the `DECISION` keyword, which maps to an `EnforcementDecision` with three possible gates: `pass` (deployment proceeds), `block` (deployment halted), or `needs_review` (escalated to human operator). If the agent fails to produce a clear decision, the system defaults to `needs_review`.

Enforcement Agent (CI Sentinel Mode)

System Prompt

You are Enforcement Agent (CI Sentinel). Role: Review security policy files (Terraform / Kubernetes) and determine whether configurations meet security baseline requirements. Review criteria: Terraform — IAM least privilege, security groups not overly permissive (e.g., 0.0.0.0/0), S3 encryption and access logging enabled. Kubernetes — NetworkPolicy exists (default deny), pods run as non-root, hostNetwork/hostPID restricted.
You must read policy files using tools, then provide a final verdict. Final reply must end with: `DECISION: PASS` or `DECISION: BLOCK` or `DECISION: NEEDS_REVIEW`.

Tools

Read-only subset: `list_directory`, `find_files`, `read_source_file`.

C. Phase II: Dynamic Runtime Monitoring

Static analysis, however thorough, cannot detect threats that emerge only at runtime: zero-day exploits, credential compromise, anomalous traffic patterns, or configuration drift introduced after deployment. The dynamic monitoring phase addresses this gap by continuously observing production systems and responding to detected threats in near real-time. This phase is designed around two complementary roles. The Monitoring Agent acts as a sensor, ingesting application logs and system metrics from Elasticsearch and using LLM reasoning to distinguish genuine security anomalies from normal operational noise—a task that rule-based alerting systems handle poorly due to high false-alarm rates. The Runtime Enforcement Agent acts as an actuator, translating the Monitoring Agent's risk assessments into concrete Kubernetes operations (applying network restrictions, scaling down compromised services, or creating alerts for human review). The separation of sensing from acting is intentional: it allows the monitoring logic to evolve independently of the enforcement logic, and it ensures that enforcement actions are always grounded in an explicit risk assessment rather than triggered by raw log patterns. An `APScheduler`-based scheduler periodically invokes the monitoring graph at a configurable interval (default: 5 minutes), and a conditional edge ensures that the Enforcement Agent is only activated when the risk level reaches medium or

above—avoiding unnecessary cluster operations during normal conditions.

1) Monitoring Agent

The Monitoring Agent analyzes application logs and system resource metrics from Elasticsearch [32] to detect runtime security anomalies. The monitoring infrastructure uses the Elastic Stack: Filebeat ships logs from containers and hosts to Elasticsearch, which indexes them for querying.

Monitoring Agent

System Prompt

You are the Runtime Monitoring Agent. Analyze application logs and system metrics to detect security anomalies.
Workflow: (1) Query recent logs via `query_es_logs`, focusing on ERROR/WARNING and keywords: injection, unauthorized, attack, exploit. (2) Query CPU, memory, network metrics via `query_es_metrics`. (3) Record anomalies via `add_log_anomaly` or `add_resource_anomaly`.
Judgment: injection patterns → `injection_attempt`; mass auth failures → `auth_failure`; anomalous access → `suspicious_access`; CPU/memory >90% → `resource_abuse`. End with `RISK_LEVEL: LOW/MEDIUM/HIGH/CRITICAL`.

Tools

- `query_es_logs` — queries Elasticsearch for application logs with optional keyword filtering.
- `query_es_metrics` — retrieves system resource metrics (CPU, memory, network, disk).
- `add_log_anomaly` — records a log-based anomaly with severity and category.
- `add_resource_anomaly` — records a resource anomaly when a metric exceeds threshold.

2) Enforcement Agent (Runtime Mode)

Enforcement Agent (Runtime Mode)

System Prompt

You are the Runtime Enforcement Agent. Role: Based on monitoring report anomalies, execute appropriate Kubernetes security hardening measures.
Available measures: (1) `kubectl_apply`: apply OPA Gatekeeper Constraint or NetworkPolicy. (2) `kubectl_scale`: emergency scale-down of suspicious services (critical level only). (3) `create_security_alert`: record alert for human review. (4) `kubectl_get`: query current cluster state.
Execution principles: MEDIUM risk — create alert, recommend but do not auto-execute. HIGH risk — apply NetworkPolicy or OPA Constraint + create alert. CRITICAL risk — consider scale-down + apply restrictions + create urgent alert. Always query current state before acting. Prefer minimal-impact measures.

Tools

- `kubectl_get` — queries Kubernetes resource status (pods, deployments, network policies, constraints).
- `kubectl_apply` — applies Kubernetes YAML manifests (OPA Gatekeeper constraints, NetworkPolicies).
- `kubectl_scale` — scales deployment replicas for emergency containment.
- `create_security_alert` — records a security alert for human review.

The Runtime Enforcement Agent executes graduated security responses on the Kubernetes cluster: medium-risk findings trigger alerts without automatic action, high-risk findings apply NetworkPolicies or OPA constraints, and critical-risk findings may additionally scale down suspicious deployments. The agent always queries current cluster state before acting and prefers restrictive policies over destructive actions.

TABLE IV
TEST APPLICATION STRUCTURE

Batch	Scope	Files	Vulns
1	Core API endpoints	main.py	13
2	Authentication & database	auth.py, database.py	10
3	Admin panel	admin.py	10
4	File handling	upload.py	7
5	Infrastructure as Code	*.tf, *.yaml	10
Total		**10 files**	**50**

IV. EVALUATION

This section evaluates the multi-agent DevSecOps framework through a series of experiments on a purpose-built test application containing 50 manually planted vulnerabilities across Python application code and infrastructure-as-code configurations. All experiments were conducted on a single machine with the framework deployed locally, using the dual-LLM configuration described in Section III.

A. Experimental Setup

The test application is a FastAPI web service with deliberately planted vulnerabilities spanning 25 CWE categories, including injection (CWE-78, CWE-89), authentication flaws (CWE-287, CWE-384), cryptographic weaknesses (CWE-327, CWE-330), infrastructure misconfigurations (CWE-250, CWE-284), and sensitive data exposure (CWE-798, CWE-532). The application is divided into five batches (Table IV) to enable incremental analysis. Batch 5 includes Terraform and Kubernetes manifests with IAM over-privilege, public S3 buckets, unencrypted RDS instances, and privileged containers. To evaluate the contribution of each detection component, we run three modes as an ablation study: Mode A (Semgrep only), Mode B (LLM only), and Mode C (Semgrep + LLM combined). Semgrep uses the default community ruleset (∼1059 rules); the LLM reviews git diffs through a ReAct agent that reports findings via a structured `add_finding` tool.

TABLE V
DETECTION ABLATION RESULTS (50 GROUND-TRUTH VULNERABILITIES)

Mode	TP	FP	FN	Recall	F1
A (Semgrep only)	17	0	33	34.0%	50.7%
B (LLM only)	45	0	5	90.0%	94.7%
C (Semgrep + LLM)	46	0	4	92.0%	95.8%

The planted vulnerabilities were selected to reflect common web-application and cloud-native DevSecOps risks rather than a single synthetic pattern. They cover injection, broken authentication, sensitive data exposure, insecure deserialization, file handling, hardcoded credentials, weak cryptography, and infrastructure misconfiguration categories that overlap with OWASP Top 10 risks [33] and common CWE classes. The purpose-built application provides a controlled oracle for measuring recall and false positives, while the inclusion of Python application code, Terraform, and Kubernetes manifests reflects the mixed application/IaC repositories encountered in practical CI/CD pipelines. This setup does not replace validation on large open-source projects; rather, it provides a reproducible first-stage evaluation with known ground truth, and broader external validation is discussed as future work.

B. Vulnerability Detection Coverage

The purpose of this experiment is to quantify the detection contribution of each component—rule-based static analysis versus LLM semantic reasoning—and to determine whether their combination yields complementary coverage.

Table V reports the ablation results. Under the manually labeled ground-truth oracle, all three modes achieved 100% precision (zero false positives). A reported finding was counted as a true positive only when its file location, vulnerability type, and security impact matched a planted vulnerability; otherwise it would be counted as a false positive. Semgrep alone detected only 17 of 50 vulnerabilities (34.0% recall), with coverage concentrated in well-known patterns such as `subprocess` shell injection, `eval`/`exec` usage, and Kubernetes privilege escalation. The LLM alone raised recall to 90.0% by detecting semantic vulnerabilities invisible to pattern matching—authentication bypass, insecure session handling, SSRF, open redirects, and hardcoded credentials across variable assignments. The combined mode adds one additional detection (RDS missing audit logging, CWE-778) that Semgrep's infrastructure rules caught but the LLM overlooked when reviewing diffs, bringing recall to 92.0%. The five vulnerabilities missed by all modes are ReDoS (CWE-1333), XXE via Python's `ElementTree` (CWE-611), missing file-type validation (CWE-434), overly permissive egress rules (CWE-284), and missing CloudWatch log exports (CWE-778)—categories requiring specialized domain knowledge that neither generic rules nor general-purpose LLM reasoning currently cover.

C. Automated Fix Quality

The purpose of this experiment is to measure whether the Fix Agent can produce reviewable candidate patches at scale—specifically, whether it can create focused remediation commits for identified vulnerabilities without execution failures.

Table VI shows that the Fix Agent successfully committed candidate patches for 60 of 61 findings (98.4% commit rate) with zero execution failures. This metric measures whether the agent can generate and stage reviewable remediation changes, not full functional correctness in production. The single skip occurred when a subsequent finding targeted code already modified by a prior fix in the same batch. The agent applied appropriate remediation strategies across 17 vulnerability types: parameterized queries for SQL injection, `shlex.split` for command injection, `html.escape` for XSS, environment variables for hardcoded credentials, `secrets` module for weak randomness, `yaml.safe_load` for unsafe deserialization, and least-privilege configurations for infrastructure misconfigurations. Average fix time was 35.5 seconds per vulnerability, with infrastructure fixes (22.3s) completing faster than application code fixes (39.4s) due to shorter file lengths and more formulaic remediation patterns. Patch quality was checked by manual inspection of the modified files, confirmation that each candidate patch addressed the intended vulnerability pattern, and re-scanning representative fixed files where applicable. The primary limitation is the absence of a full automated regression test suite, because the test application was designed as a vulnerability corpus rather than a production service with unit tests.

TABLE VI
FIX AGENT RESULTS ACROSS ALL BATCHES

Batch	Attempted	Committed	Skip	Avg Time (s)
1 (Core API)	14	14	0	45.4
2 (Auth & DB)	17	17	0	39.9
3 (Admin)	8	8	0	38.6
4 (File handling)	9	9	0	33.6
5 (Infrastructure)	13	12	1	22.3
Total	**61**	**60**	**1**	**35.5**

D. Policy Generation

The purpose of this experiment is to verify that the Policy Agent correctly identifies infrastructure-as-code scenarios and generates syntactically valid, security-relevant policy files without human intervention.

Table VII summarizes the Policy Agent's behavior. Across five batches, the agent correctly determined that only Batch 5 contained infrastructure files requiring policy generation, skipping the four Python-only batches without false activation. When triggered, it produced three files per run: an OPA Rego policy covering six Terraform security rules (IAM least privilege, S3 public access blocking, S3 encryption, RDS private access, RDS encryption, security group minimum exposure) and a Kubernetes NetworkPolicy implementing default-deny ingress with minimal port allowlisting. Two independent runs

TABLE VII
POLICY AGENT RESULTS

Metric	Value
IaC scenario identification accuracy	100% (5/5 batches)
Policy generation success rate	100% (2/2 runs)
Average generation time	56.0s
Generated file types	OPA Rego + K8s NetworkPolicy
Terraform security rules covered	6
Kubernetes rules covered	2
Cross-run consistency	High (same structure)

produced structurally consistent policies with only minor naming differences, demonstrating reproducibility. The average generation time of 56 seconds compares favorably to the estimated 1–2 hours required for manual policy authoring by a security engineer.

E. Multi-Agent Design Justification

To assess whether the multi-agent architecture provides benefits beyond the detection combination alone, we compare the responsibilities enabled by reduced configurations (Table VIII). This ablation is architectural rather than a direct accuracy benchmark, because removing agents changes the task scope. A detection-only pipeline can identify vulnerabilities but cannot close the remediation loop. Adding the Fix Agent enables candidate patch generation, but residual infrastructure risks may remain undetected until an enforcement gate evaluates the resulting repository. Adding the Policy and Enforcement Agents introduces prevention and deployment gating, while the Monitoring and Runtime Enforcement Agents extend the workflow to post-deployment threats. The full configuration is therefore justified not only by detection recall, but also by lifecycle coverage and conservative escalation behavior when agent outputs are incomplete or inconsistent.

F. Enforcement Decision Accuracy

The purpose of this experiment is to evaluate whether the Enforcement Agent (CI sentinel mode) can reliably distinguish compliant from non-compliant infrastructure configurations when acting as a deployment gate.

Table IX reports four runs with progressively improving configurations. Run 1 correctly blocked the original vulnerable infrastructure (IAM wildcard permissions, public S3, privileged containers). Run 2 is particularly informative: after the Fix Agent's automated remediation, the Enforcement Agent still issued BLOCK because it identified residual issues the Fix Agent missed—missing S3 server-side encryption, overly permissive egress rules, and absence of `runAsNonRoot`. This demonstrates defense-in-depth: the CI gate catches what upstream agents overlook. Run 3 passed after manual remediation of all residual issues. Run 4, using identical configuration to Run 3, received a NEEDS_REVIEW verdict due to LLM non-determinism—the model raised concerns about egress rules to `0.0.0.0/0` on ports 443 and 53 that it had previously accepted. The system design mitigates this variability: NEEDS_REVIEW routes to human review rather than auto-passing, ensuring that conservative judgments do not compromise security. All runs completed in under 34 seconds with only 2 LLM calls each, consuming 2,647–4,135 tokens—an order of magnitude less than the detection phase.

We further repeated the compliant configuration used in Runs 3–4 to examine enforcement stability. Across five repeated invocations with the same prompt template and low-temperature enforcement setting, the agent returned PASS in three runs and NEEDS_REVIEW in two runs; no run incorrectly returned BLOCK. The observed variability was caused by different interpretations of controlled outbound egress to DNS and HTTPS endpoints. This result supports two practical design choices: enforcement prompts should keep security baselines explicit, and uncertain LLM judgments should be routed to human review rather than treated as automatic approval.

G. Runtime Monitoring and Response

The purpose of this experiment is to evaluate the Monitoring Agent's ability to detect security events from production logs and metrics, and the Enforcement Agent's (runtime mode) ability to execute proportional automated responses.

Table X reports three scenarios tested against a live FastAPI application with Filebeat log collection and Metricbeat system metrics flowing into Elasticsearch. In the normal traffic scenario, the Monitoring Agent correctly identified zero log-level anomalies; the MEDIUM risk rating and single alert action resulted from a disk utilization anomaly (a Docker container filesystem artifact, not a security event). In the mixed attack scenario (SQL injection + brute-force login + unauthorized access attempts generated simultaneously), the agent detected all three attack categories and escalated to CRITICAL—the highest severity observed. The injection-only scenario received HIGH, demonstrating that risk assessment correlates with attack diversity and severity. The Enforcement Agent successfully executed a complete response loop in the injection scenario: deploying a `default-deny-ingress` NetworkPolicy to restrict traffic and creating a security alert. In the mixed attack scenario, the agent expressed correct remediation intent but could not execute because the test Kubernetes cluster contained no deployable workloads. All monitoring cycles completed within 42 seconds, with the Monitoring Agent consuming 22,899–82,418 tokens depending on log volume (more logs require more context for LLM analysis).

H. End-to-End Pipeline Timing

The purpose of this experiment is to characterize the time cost of each pipeline stage, identifying where optimization effort should be directed and confirming that the framework operates within acceptable CI/CD time budgets.

Table XI breaks down execution time by stage. The dominant cost is the Fix Agent, which processes vulnerabilities sequentially at 35.5 seconds each—for 10 findings, this alone accounts for nearly 6 minutes. The Code Security Agent's two-phase operation (review + summarization) takes approximately 104 seconds total. For a typical commit triggering 5–15

TABLE VIII
ARCHITECTURE-LEVEL ABLATION OF MULTI-AGENT RESPONSIBILITIES

Configuration	Detection	Candidate Repair	Policy Generation	CI Gate	Runtime Response
Semgrep only	Rule patterns only	No	No	No	No
LLM detection only	Semantic review	No	No	No	No
Detection + Fix Agent	Hybrid detection	Yes	No	No	No
Detection + Fix + Policy/CI Agents	Hybrid detection	Yes	Yes	Yes	No
Full multi-agent framework	Hybrid detection	Yes	Yes	Yes	Yes

TABLE IX
ENFORCEMENT AGENT CI MODE RESULTS

Run	Configuration	Expected	Actual	Time (s)
1	Original violations	BLOCK	BLOCK	22.8
2	After auto-fix	PASS	BLOCK	23.5
3	Fully compliant	PASS	PASS	24.5
4	Same as Run 3	PASS	REVIEW	33.8

TABLE X
RUNTIME MONITORING RESULTS

Scenario	Risk	Log Anom.	Actions	Time (s)
Normal traffic	MEDIUM	0	1 (alert)	33
Mixed attack	CRITICAL	3	0	36
Injection only	HIGH	2	2	42

TABLE XI
PIPELINE STAGE TIMING (AVERAGED ACROSS 5 BATCHES)

Stage	Average Time
Repository clone	20.6s
Semgrep scan (when enabled)	12.1s
Code Security Agent (LLM review)	50.9s
Code Security Agent (summarization)	52.9s
Policy Agent (when triggered)	49.0s
Fix Agent (per vulnerability)	35.5s
PR creation	6.8s
Full pipeline (10 findings)	∼9 min

findings, the full pipeline completes in 5–12 minutes, comparable to existing CI security scanning tools and delivered asynchronously via pull requests.

V. CONCLUSION

This paper presented a multi-agent DevSecOps framework that unifies vulnerability detection, automated repair, policy generation, enforcement, and runtime monitoring into a single event-driven pipeline. Five specialized agents collaborate through LangGraph shared state graphs across two operational phases. Evaluation on 50 planted vulnerabilities demonstrates that combining Semgrep rule matching with LLM semantic reasoning achieves 92.0% recall with zero false positives under the manually labeled test oracle—a 58 percentage point improvement over Semgrep alone (34.0%). The Fix Agent commits reviewable candidate patches for 98.4% of detected vulnerabilities in an average of 35.5 seconds each. The Policy Agent generates valid OPA Rego and Kubernetes NetworkPolicy files with 100% success rate, and the Enforcement Agent provides defense-in-depth by catching residual issues that upstream agents miss. Runtime monitoring detects multi-vector attacks with risk-proportional automated response within 42 seconds. The full static analysis pipeline completes in 5–12 minutes depending on finding count, operating within acceptable CI/CD time budgets.

Several limitations remain. The evaluation uses a purpose-built test application with planted vulnerabilities, which provides clear ground truth but does not fully capture the diversity of large open-source or enterprise systems. The current patch metric measures candidate patch generation and commit success rather than complete functional remediation under a regression test suite. LLM non-determinism causes identical configurations to receive different enforcement verdicts across runs, though the system's conservative default (routing uncertain cases to human review) prevents security gaps. Five vulnerability categories—ReDoS, XXE, missing file-type validation, egress rule analysis, and absent audit logging—evade both detection modes, indicating that specialized detectors or fine-tuned models are needed. Future work includes validating the framework on real-world repositories and enterprise CI/CD pipelines, integrating retrieval-augmented generation with security knowledge bases, adding parallel fix processing to reduce pipeline time, and conducting longitudinal studies on LLM verdict stability.

VI. ACKNOWLEDGEMENTS

This work was supported by in part by the Key Research and Development Program of Shandong Province of China (Grant No.2023CXPT056), the National Natural Science Foundation of China (Grant No. U24A20238, No.62502363, No. 92467201), the Natural Science Basic Research Program of Shaanxi Province (2025JC-YBQN-869), the Open Project of Shandong Provincial Key Laboratory of Independent and Reliable Computing Technology and Equipment (No.KT25500400-lab), the Key R&D Program of Shandong Province of China(No.2025CXPT089).

REFERENCES

[1] H. Myrbakken and R. Colomo-Palacios, "Devsecops: A multivocal literature review," in *Proceedings of the International Conference on Software Process Improvement and Capability Determination (SPICE 2017)*. Springer, 2017, pp. 17–29.

[2] HashiCorp, "What is infrastructure as code," https://www.hashicorp.com/resources/what-is-infrastructure-as-code, 2022.

[3] L. Prates and R. Pereira, "Devsecops practices and tools," *International Journal of Information Security*, vol. 24, no. 1, p. 11, 2025.

[4] M. Sánchez-Gordón and R. Colomo-Palacios, "Security as culture: A systematic literature review of devsecops," in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops (ICSE 2020 Workshops)*. IEEE/ACM, 2020, pp. 266–269.

[5] N. K. K. Reddy Yelkoti, "Security as code: An architectural framework for automated risk mitigation in devsecops pipelines," *Journal of Computer Science and Technology Studies*, vol. 7, no. 6, pp. 235–244, 2025. [Online]. Available: https://al-kindipublisher.com/index.php/jcsts/article/view/9959

[6] O. Vakhula and I. Opirskyy, "Ai-driven security as code for software development using multi-agent systems," in *Proceedings of the Fourth International Conference on Cyber Hygiene and Conflict Management in Global Information Networks (CH&CMiGIN'25)*, ser. CEUR Workshop Proceedings, vol. 4024, 2025, pp. 129–138. [Online]. Available: https://ceur-ws.org/Vol-4024/paper13.pdf

[7] National Institute of Standards and Technology (NIST), "Devsecops," https://csrc.nist.gov/projects/devsecops, accessed: 2026-05-25.

[8] Semgrep, Inc., "Semgrep," https://semgrep.dev, 2020, open-source static analysis tool, source code: https://github.com/semgrep/semgrep.

[9] GitHub Security Lab, "Codeql: Github's semantic code analysis engine," https://codeql.github.com/, 2019, blog announcement: https://github.blog/2019-11-14-security-lab/.

[10] Open Policy Agent, "Open policy agent (opa)," https://www.openpolicyagent.org/, 2023.

[11] H. Xu, S. Wang, N. Li, K. Wang, Y. Zhao, K. Chen, T. Yu, Y. Liu, and H. Wang, "Large language models for cyber security: A systematic literature review," 2024. [Online]. Available: https://arxiv.org/abs/2405.04760

[12] J. Bae, S. Kwon, and S. Myeong, "Enhancing software code vulnerability detection using gpt-4o and claude-3.5 sonnet: A study on prompt engineering techniques," *Electronics*, vol. 13, no. 13, p. 2657, 2024.

[13] J. Zhang, C. Wang, A. Li, W. Wang, T. Li, and Y. Liu, "Vuladvisor: Natural language suggestion generation for software vulnerability repair," in *Proceedings of the IEEE/ACM 39th International Conference on Automated Software Engineering (ASE 2024)*, 2024, pp. 1932–1944.

[14] Y. Charalambous, N. Tihanyi, R. Jain, Y. Sun, M. A. Ferrag, and L. C. Cordeiro, "A new era in software security: Towards self-healing software via large language models and formal verification," 2023. [Online]. Available: https://arxiv.org/abs/2305.14752

[15] J. Gong, N. Duan, Z. Tao, Z. Gong, Y. Yuan, and M. Huang, "How well do large language models serve as end-to-end secure code agents for python?" 2024. [Online]. Available: https://arxiv.org/abs/2408.10495

[16] C. Qian, X. Cong, W. Liu, C. Yang, W. Chen, Y. Su, Y. Dang, J. Li, J. Xu, D. Li, Z. Liu, and M. Sun, "Chatdev: Communicative agents for software development," 2023. [Online]. Available: https://arxiv.org/abs/2307.07924

[17] S. Hong, M. Zhuge, J. Chen, X. Zheng, Y. Cheng, C. Zhang, J. Wang, Z. Wang, S. K. S. Yau, Z. Lin, L. Zhou, C. Ran, L. Xiao, C. Wu, and J. Schmidhuber, "Metagpt: Meta programming for multi-agent collaborative framework," 2023. [Online]. Available: https://arxiv.org/abs/2308.00352

[18] Q. Wu, G. Bansal, J. Zhang, Y. Wu, S. Zhang, E. Zhu, B. Li, L. Jiang, X. Zhang, and C. Wang, "Autogen: Enabling next-gen llm applications via multi-agent conversation framework," 2023. [Online]. Available: https://arxiv.org/abs/2308.08155

[19] D. Kassimi, O. Kazar, H. Saouli, and O. Boussaid, "Design and implementation of a new approach using multi-agent system for security in big data," *International Journal of Software Engineering and Its Applications*, vol. 11, no. 9, pp. 1–14, 2017.

[20] W. Chen, Y. Su, J. Zuo, C. Yang, C. Yuan, C. Qian, C.-M. Chan, Y. Qin, Y. Lu, R. Xie *et al.*, "AgentVerse: Facilitating multi-agent collaboration and exploring emergent behaviors in agents," *arXiv preprint arXiv:2308.10848*, 2023.

[21] LangChain, Inc., "Langgraph: Build resilient language agents as graphs," https://github.com/langchain-ai/langgraph, 2024, accessed: 2025-03-15.

[22] W. Charoenwet, P. Thongtanunam, V.-T. Pham, and C. Treude, "An empirical study of static analysis tools for secure code review," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2024, pp. 691–703.

[23] X.-C. Wen, C. Gao, S. Gao, Y. Xiao, and M. R. Lyu, "Scale: Constructing structured natural language comment trees for software vulnerability detection," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 235–247.

[24] X. Du, M. Wen, J. Zhu, Z. Xie, B. Ji, H. Liu, X. Shi, and H. Jin, "Generalization-enhanced code vulnerability detection via multi-task instruction fine-tuning," in *Findings of the Association for Computational Linguistics: ACL 2024*. ACL, 2024, pp. 10 507–10 521.

[25] A. Zibaeirad and M. Vieira, "Diverse LLMs vs. vulnerabilities: Who detects and fixes them better?" *arXiv preprint arXiv:2512.12536*, 2025.

[26] K. Li, J. Zhang, S. Chen, H. Liu, Y. Liu, and Y. Chen, "Patchfinder: A two-phase approach to security patch tracing for disclosed vulnerabilities in open-source software," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 590–602.

[27] R. Kumar and R. Goyal, "Modeling continuous security: A conceptual model for automated devsecops using open-source software over cloud (adoc)," *Computers & Security*, vol. 97, p. 101967, 2020.

[28] O. B. Abiola and O. G. Olufemi, "An enhanced ci/cd pipeline: A devsecops approach," *International Journal of Computer Applications*, vol. 184, no. 48, pp. 8–13, 2023.

[29] V. Gupta and D. Sreenivasamurthy, "Prose2policy (P2P): A practical LLM pipeline for translating natural-language access policies into executable Rego," *arXiv preprint arXiv:2603.15799*, 2026.

[30] I. H. Sarker, M. H. Furhad, and R. Nowrozy, "Ai-driven cybersecurity: An overview, security intelligence modeling and research directions," *SN Computer Science*, vol. 2, no. 3, p. 173, 2021.

[31] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "React: Synergizing reasoning and acting in language models," in *The Eleventh International Conference on Learning Representations*, 2023. [Online]. Available: https://openreview.net/forum?id=WE_vluYUL-X

[32] Elastic. (2023) Elastic stack documentation. Logging, metrics, and monitoring documentation. [Online]. Available: https://www.elastic.co/docs

[33] OWASP Foundation, "Owasp top ten web application security risks," https://owasp.org/www-project-top-ten/, 2021.