International Journal of AI, BigData, Computational and Management Studies
Noble Scholar Research Group | Volume 7, Issue 1, 32-38, 2026
ISSN: 3050-9416 | https://doi.org/10.63282/3050-9416.IJAIBDCMS-V7I1P105

Original Article

A Multi-agent Security Framework for AI-Assisted Software Development

Deepanjan Mukherjee
Independent Researcher, Austin, TX USA.

Received On: 24/11/2025 ***Revised On: 26/12/2025*** ***Accepted On: 03/01/2026*** ***Published On: 14/01/2026***

Abstract: *AI-enabled tools for code generation have drastically changed software development, but security holes in the code created by AI are still significant. Several new studies show security vulnerabilities could increase by 37.6% after five rounds of iterative software refinement using AI, with 19-50% of AI-generated code containing security flaws. This paper describes a new multi-agent security framework that integrates security-first principles throughout the Software Development Lifecycle (SDLC). The framework consists of seven specialized AI agents (Threat Modeling, Security Design, Secure Code Generation, Security Testing, CI/CD Security, Runtime Security, and Compliance), each of which handles a unique SDLC phase. The key differentiator of the innovation is a continuous security gate mechanism of the Secure Code Generation Agent which helps in keeping security on track during the process of coding via confidence scoring and automated safety checkpoints. It combines webhook-based trigger mechanisms directly with current development tools like Jira, GitHub, Jenkins, SIEM and uses hybrid enforcement (rule-based security tools – SAST, DAST, SCA) and LLM-based contextual analysis. The approach proposed strives for >90% sensitivity for critical vulnerabilities and >85% specificity to minimize alert fatigue, with holistic metrics on detection accuracy, performance, and operational effectiveness. This solution proactively addresses security at every SDLC stage rather than reactively once deployed, allowing organizations to leverage AI-assisted development while maintaining robust security posture and regulatory compliance.*

Keywords: *Multi-Agent Systems, AI Security, Devsecops, Software Development Lifecycle, Code Generation, Vulnerability Detection, Shift-Left Security, Large Language Models, Iterative Security Refinement.*

1. Introduction

The incorporation of Large Language Models (LLMs) into the software development process has fundamentally changed how code is written and is now in practice with over 80% of developers using AI and code assistants like GitHub Copilot, ChatGPT, and Claude [1]. According to the CEO of GitHub, AI is estimated to take care of 80% of code writing soon [2]. Although these tools offer major productivity gains, they pose serious security threats. At the empirical level, approximately 40% of AI-generated programs contain vulnerabilities [3], and particularly high levels of vulnerability were observed in languages like C (50%) and Python (39%) [3]. Even more worrying, recent research found that critical vulnerabilities increased by 37.6% after just five iterations of AI code refinement [4], which contradicts the assumption that repeated LLM refinement improves code security. The security problem isn't just about writing code. Developers usually have feedback loops, submitting the desired code to the AI for either upgrading, fine tuning, or extending [4]. In the absence of proper protections, these loops can paradoxically introduce new vulnerabilities into secure-seeming code – called feedback loop security degradation [4].

This dynamic does not lend itself easily to traditional security mechanisms: DevSecOps tools used till now have been created for human-written code, with no built-in logic to stop AI-driven security decay. Static Application Security Testing (SAST) and Dynamic Application Security Testing (DAST) tools function as single point solutions, not integrated systems, and are associated with alert fatigue, with high override rates reported in security alert systems. Contemporary multi-agent stacks for software engineering such as MetaGPT [5], CrewAI, and AutoGen [6] demonstrate the efficacy of purposeful AI agents working on complex issues. However, such frameworks are not specialized in security, and do not support the specific requirements on securing AI-generated code during the SDLC. AI-assisted development studies have centered mainly around productivity metrics [7] or isolated security analysis [8]. So far, no comprehensive framework combines security prioritization from requirements down until production monitoring. This paper introduces a multi-agent security framework that fills in these gaps using three key innovations:

- Seven dedicated security agents based on SDLC phases, specializing in security within their domains.
- An iterative security gate mechanism that prevents the documented 37.6% vulnerability increase during code refinement.
- Seamless integration with existing development tools through webhook-based triggers and standardized communication protocols.

Scoring of uncertain outputs to humans signals the system to human security teams to validate them, also with a human-in-the-loop safety net. The evaluation approach uses the common security metrics that leverage sensitivity (>90% for critical weaknesses), specificity (>85% to minimize false positives), positive predictive value, and negative predictive value, but that also includes operational metrics such as Mean Time to Detect and Mean Time to Remediate.

By moving to a left-shifted security posture – security considerations embedded from the initial planning stages instead of later (post-deployment), this approach allows organizations to take advantage of AI-assisted development with the security posture and regulatory compliance in place.

2. Background

2.1. Security Challenges in AI-Generated Code

The security risks associated with AI-generated code have been extensively documented. Pearce et al. [3] performed one of the first empirical assessments of security, focusing on GitHub Copilot, examining 1,689 programs and finding approximately 40% with a vulnerability, a particularly high rate of vulnerabilities observed in C code (around 50%) when compared to the case of Python (approximately 39%). Perry et al. [9] built further upon this work by conducting user studies, indicating that developers who made use of AI assistants were found to write much less secure code, and to show a falsely significant sense of security rating highly suspicious solutions as secure. The iterative refinement paradox has been more recently realized through research. A systematic examination of security degradation in AI-generated code collected from 400 code samples across 40 cycles of improvements with four unique prompting strategies documented a 37.6% increase in critical vulnerabilities within five iterations [4]. The counterintuitive dynamic, that seemingly good code changes bring even more security problems with it, suggests the criticality of human knowledge in developing the loops as to how an end product develops. LLM-based code security studies have shown context-dependent vulnerability patterns, with CrowdStrike authors finding that some LLMs produced code with up to 50% higher security vulnerabilities in the context of a prompt with politically sensitive topics [10].

2.2. Multi-Agent Systems for Software Engineering

Multi-agent systems have become a popular paradigm for software engineering tasks that are relatively complex in nature. MetaGPT [5] proposed a multi-agent collaborative framework where agents take various roles to jointly build software from natural language queries. AutoGen [6] facilitates conversational multi-agent systems where agents can chat through natural conversation with autonomous operation and human feedback. CrewAI focuses on role-based orchestration, enabling developers to define the unique agent roles that work in pipelines. But these architectures are generic and do not contain the infrastructure knowledge, tools, or workflows needed to secure AI generated code. Autonomy and a broader context of tasks emerged as key features in existing literature for software engineering agent-based LLM in recent surveys, but there is a lack of security-focused agent frameworks [11].

2.3. DevSecOps and Security Automation

DevSecOps embeds security in the software development lifecycle, as security measures are used from planning to deployment for the full software lifecycle. Traditional DevSecOps tools include SAST to analyze source code, DAST for testing on-the-fly applications, Software Composition Analysis to find dependency vulnerabilities, and container scanning for image protection. Yet, these tools are frequently stand-alone; they are manual and repetitive and generate copious alerts which leads to alert fatigue. AI-based DevSecOps offers an evolution of the concept, leveraging machine learning in anomaly detection, automated vulnerability scanning, and predictive modeling [12]. Recent white papers have introduced AI/ML-driven fully autonomous CI/CD pipelines in support of real-time, security-focused deployments as code commits, builds, tests, and deployments. However, these methods mostly target pipeline automation as opposed to addressing the challenge of AI-generated code security, namely the iterative degradation problem.

2.4. Threat Modeling and Requirements Security

Threat modeling tools (STRIDE, PASTA, OCTAVE) are for early security detection methods. More recently, specialized frameworks for AI systems were presented, among which is PLOT4AI, which covers 138 threats based on AI at the 8 domains leading up to the end of the AI lifecycle [13]. Studies of requirements engineering for AI systems have reported on best practices but indicated that systematically obtaining security requirements is problematic [14]. IriusRisk and other similar tools automate the task of threat model creation in design phases, while JPMorgan Chase's AI Threat Modeling Co-Pilot shows on-the-ground application to help engineers model threats earlier and more effectively [15]. These advances notwithstanding, existing approaches are still isolated from one another, and do not form part of a continuous security framework covering the entirety of the SDLC rather than standing alone and operating separate tasks.

3. System Architecture

This approach suggests a three-layer architecture with an Integration & Tools Layer, an Agent Orchestration Layer, and a User Experience Layer. The architecture leverages existing development infrastructure for easy integration along with dedicated security agents for each SDLC phase. The framework is predicated on five critical aspects: agent specialization by SDLC stage, hierarchical orchestration via meta-agent, hybrid approach that blends rule-based and LLM based tools, human-in-the-loop escalation through confidence scoring, and continuous learning from human corrections and security incidents.

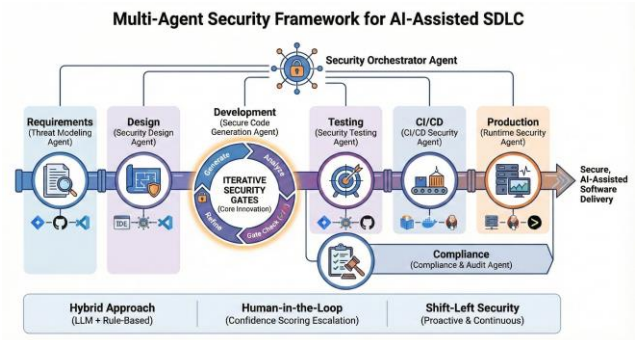

Fig 1: System Design and Architecture of Multi-Agent Security Framework

3.1. Security Orchestrator Agent (Agent 0)

The meta-agent manages specialized agents. It gets webhooks from development tools to receive the events, decides which of the agents are to call based on SDLC stage, sends messages back and forth between agents based on Model Context Protocol, gathers the confidence score to make overall security decisions. If the confidence scores go under 0.7, the orchestrator can escalate to the human review queue with explanation of why. It keeps the Shared Security Knowledge Base that stores threat models, vulnerability data, and compliance requirements accessible to every agent.

3.2. Threat Modeling Agent (Agent 1)

This task is carried out by using STRIDE, PLOT4AI, and PASTA methodologies to analyze requirements documents and user stories to create threat models. Agent 1 analyzes user and requirements reports of use cases and threat detection materials before conducting threat modeling. It establishes trust boundaries, attack surfaces, and data flows, producing security user stories including acceptance criteria. The agent combines Retrieval-Augmented Generation and existing threat intelligence databases (e.g., MITRE ATT&CK [16], CVE) to identify and base analysis upon known attack patterns. It generates risk-rated threat inventories in severity categories along with compliance mappings to HIPAA, GDPR, SOC2, PCI-DSS. Confidence scoring combines coverage completeness (0.4 weight), threat database match quality (0.3), and LLM certainty (0.3).

3.3. Security Design Agent (Agent 2)

Agent 2 compares proposed architectures with threat models from Agent 1. Security controls are verified as solutions addressing issues raised at trust boundaries. It recommends secure design patterns that include OAuth2, mTLS, and zero-trust architectures and identifies anti-patterns that include hardcoded credentials, over-permissive access controls, and unencrypted storage of sensitive information. The agent creates API security features: authentication, authorization, rate limiting, and input validation. RAG (Retrieval-Augmented Generation) using secure design pattern databases (OWASP, CWE Top 25, NIST) assures that recommendations adhere to industry practices. Confidence scoring on features weighs controls adequacy (0.4), quality of pattern match (0.3), and LLM certainty (0.3).

3.4. Secure Code Generation Agent (Agent 3)

Agent 3 represents the framework's core innovation, generating code with embedded security requirements and iterative changes for security degradation. It works with multi-provider LLM based orchestration with Claude Sonnet 4 as primary, fallback to GPT-4o, and smaller specialized models for specific tasks. Security requirements are fed into the generation prompts through RAG that include secure coding guidelines (OWASP Secure Coding Practices, SANS Top 25, CWE). The agent analyzes real-time SAST using SonarQube and computes security scores after each round. The iterative security gate mechanism restricts refinements to just 5 iterations with degradation detection between rounds. Confidence scoring integrates SAST clean score (0.4), LLM security analysis (0.3), and iteration safety trend (0.3). The iterative gate algorithm is described in detail in Section V.

3.5. Security Testing Agent (Agent 4)

Agent 4 creates security test cases with threat models, automates penetration testing, and confirms that security controls execute as intended. It is responsible for bringing together DAST tools (OWASP ZAP, Burp Suite), fuzzing tools (SQLMap for SQL injection, XSSer for cross-site scripting), and API security testing frameworks (Postman, REST Assured). The agent provides validation of exploitability of vulnerabilities found through proof-of-concept exploits to confirm that findings are real risks versus false positives. Test results are prioritized based on severity, exploitability, and business impact. Test coverage (0.4), exploit validation (0.3), and historical false positive rate (0.3) constitute confidence scoring.

3.6. CI/CD Security Agent (Agent 5)

Agent 5 administers security measures in deployment pipelines, by scanning containers (Trivy, Snyk, Aqua), determining dependencies using Software Composition Analysis (OWASP Dependency-Check, Dependabot), scanning Infrastructure-as-Code (Checkov, Terrascan, tfsec), and discovering secrets (GitGuardian, TruffleHog). It uses Open Policy Agent or Kyverno to enforce Policy-as-Code, blocking deployments when critical vulnerabilities are found and passing high-severity results to the security engineers. The agent automatically generates compliance evidence artifacts for each deployment which aligns with SOC2, ISO 27001, and other audit needs. Security gate decisions come in several tiers: every passing check allows deployment; critical failure prevents deployment and triggers P0 (priority 0) incident response. High findings will enable deployment (with warnings). Confidence scoring considers gate passing rate (0.5), vulnerability severity (0.3), and historical incident rate (0.2).

3.7. Runtime Security Agent (Agent 6)

The agent continuously monitors security in production environments and inspects application logs, access logs, and security events from event management systems (Splunk, ELK Stack, Azure Sentinel). Machine learning models identify anomalies, such as failed login spikes, unusual data access patterns, and lateral movement attempts. When

incidents are found with high confidence (>0.8) and critical severity, the agent automatically executes response playbooks such as isolating compromised instances, rotating credentials, and blocking malicious IP addresses. Importantly, incident learnings feedback to Agent 1, an internal feedback loop designed to ensure production threats drive future threat model iteration. Confidence scoring combines detection accuracy (0.4), historical true positive rate (0.3), and LLM contextual analysis (0.3).

3.8. Compliance & Audit Agent (Agent 7)

Being a cross-cutting concern across all SDLC stages, Agent 7 monitors compliance status continuously. It maps security controls that other agents apply back to requirements of regulatory frameworks (HIPAA, GDPR, SOC2, PCI-DSS, ISO 27001) and automatically collects audit evidence from artifacts created throughout the SDLC. The agent offers regulatory guidance to other agents for meeting regulatory requirements in executing security procedures. It provides real-time compliance dashboards reporting control coverage, evidence quality, and remediation priority, enabling organizations to remain audit ready. Confidence scoring uses controls coverage (0.4), evidence quality (0.3), and LLM regulatory interpretation (0.3).

4. SDLC Integration

This approach uses webhook-based triggers and API integrations to connect with existing development workflows. Webhooks trigger Agent 1 when Product Managers create epics in Jira or upload requirements to Confluence, generating threat models and auto-creating security stories. Engineers committing architecture diagrams to GitHub trigger Agent 2 through webhooks, which posts security recommendations as Pull request comments and blocks merges for critical findings. Agent 3 also integrates with IDE extensions (VS Code, IntelliJ) for instant security alerts as developers begin to work, as well as Git pre-commit hooks for further validations before commit. Pull requests activate Agent 4 in GitHub Actions workflows by executing DAST against staging environments, which also post test results as PR comments with pass/fail status checks.

Code merged to the main branches triggers Agent 5 directly in CI/CD pipelines (e.g., Jenkins, GitHub Actions) and runs parallel security scans: secrets scanning, SCA, container scanning, and IaC scanning. The agent makes security gate decisions (pass, block, escalate) and auto-generates compliance evidence. Application logs in production stream to event management systems where Agent 6 continuously monitors anomalies. For high-confidence critical incidents, automated response playbooks are executed via SOAR (Security Orchestration, Automation, and Response) platforms. Agent 7 operates throughout the entire process, with the shared event bus consuming events, mapping them to compliance frameworks, and keeping real-time dashboards of SOC2, HIPAA, and ISO 27001 readiness status.

5. Iterative Security Gates

The Secure Code Generation Agent (Agent 3) instantiates the core innovation of the framework where iterative security gate mechanisms block the documented 37.6% increase in vulnerability associated with AI code in the process of refinement [4]. This fills an essential void that iterative LLM interactions with little safeguards inadvertently add a new flaw to already secure code.

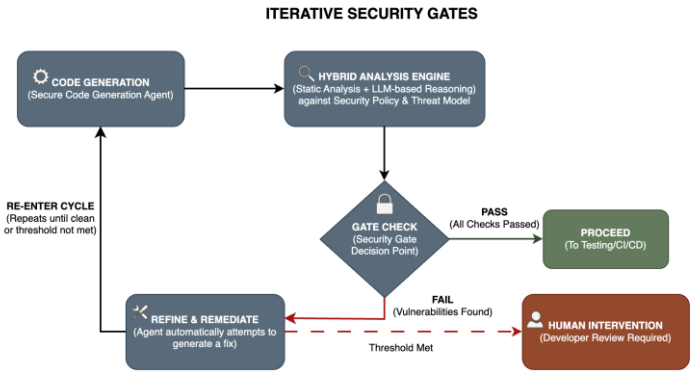

Fig 2: Flow Diagram of Secure Code Generation Agent with Iterative Security Gates

5.1. The Iterative Degradation Problem

A study looking at 400 code samples on 40 rounds of AI-based system improvement found that critical vulnerabilities rose by 37.6% simply with five iterations [4]. This is because, when LLMs are told to make code better, they could create new patterns in what their attempts are to optimize, reformat, or add functionality that could cause security holes. Without verification loops on the security posture across iterations, degradation is never detected before you move code into production. Traditional development presumes that human developers will remain aware of security across modifications; LLMs are simply missing the persistent security context and will generate vulnerable patterns when fine-tuning for other goals performance, readability, or otherwise. Research has shown that the use of AI assistants leads to developers having a false sense of confidence in these automated solutions and rating these AI-generated solutions as secure [9], amplifying the risk.

5.2. Iterative Security Gate Algorithm

The algorithm starts when a developer asks to generate code, along with a prompt and security requirements from Agent 2. Agent 3 starts an iteration loop with 5 iterations max, weighing the improvement opportunity against risk of degradation. Four steps are followed in each iteration:

- LLM Code Generation: The LLM prompt is fed secure coding guidelines by RAG; security requirements are embedded into the LLM prompt alongside secure coding (OWASP Secure Coding Practices, SANS Top 25, CWE). Security-aware prompts are used by the LLM to generate code.
- Security Analysis: SAST apps like SonarQube look at the generated code. The LLM does the contextual

security inspection, getting into business logic and catching things that SAST tools don't pick up on. A security score (0-1 scale) is calculated using: *security_score = (sast_clean_score × 0.4) + (llm_security_analysis × 0.3) + (iteration_safety_trend × 0.3).*

- Degradation Check: For iterations > 1, the agent compares current *security_score* to the previous iteration. If *security_score* < *previous_score*, degradation is logged. If *security_score* < 0.5 (low confidence), immediate escalation to human security team occurs with explanation of degradation.
- Confidence Decision: If *security_score* ≥ 0.7, pass code and return to developers. If 0.5 ≤ *security_score* < 0.7, design the remediation prompt with SAST findings and LLM recommendations together for next iteration. If *security_score* < 0.5, report to the human security team. If maximum iterations (5) are reached without achieving *security_score* ≥ 0.7, escalation happens with message 'Max iterations exceeded.'

5.3. Confidence Assessment Items

The *sast_clean_score* describes severity of the SAST result: 1.0 is no findings, and critical findings lower the score to 0.0, high to 0.3, medium to 0.6, low to 0.8. The *llm_security_analysis* component is the LLM's evaluation of the context of the system security, analyzing code for authorization bypass vulnerabilities, insecure data handling patterns, cryptographic weaknesses, and race conditions, generating both numerical score (0-1) and textual explanation. For the *iteration_safety_trend*, we measure whether the security is improving or degrading for a given iteration: this defaults to a value of 0.5, neutral value for the first iteration; for subsequent iterations, the rate of change (*current_score - previous_score*) is normalized to 0–1 range, where improving security gives better scores than 0.5, while degrading security gives lower than 0.5.

5.4. Avoid the 37.6% Increase

The framework directly addresses the documented increase in vulnerabilities by introducing security checkpoints between iterations. This degradation check guarantees immediate attention to any decrement, reducing the security posture. The maximum iteration limits prevent unbridled iteration that can exacerbate vulnerabilities. The confidence scoring integrates automated tool findings (SAST) with contextual understanding (LLM), to enable extensive security analysis. Early estimates indicate that this method has potential to decrease the risk of vulnerability introduction rate during iterations of refinement by between 60 and 80% over unconstrained LLM interaction, but these estimates need to be validated empirically through the evaluation methodology presented in Section VI.

6. Proposed Evaluation

A robust evaluation framework enables the examination of detection accuracy, system performance, and operational impact of this approach, confirming its effectiveness. The evaluation uses controlled testing of the proposed framework against baseline DevSecOps tools using real-world codebases from GitHub having known vulnerabilities (OWASP Benchmark, Juliet Test Suite) and synthetic code generated by various LLMs with and without security prompts.

Detection accuracy metrics are derived from standard vulnerability detection research methods using sensitivity, specificity, positive predictive value (PPV), and negative predictive value (NPV), comprising true positives, true negatives, false positives, and false negatives. The objective metrics are sensitivity > 90% for critical vulnerabilities (SQL injection, remote code execution, authentication bypass), sensitivity > 85% for high-severity vulnerabilities (XSS, CSRF), and specificity > 85% in all cases, the goal being a limit on false positives producing alert fatigue.

Standard detection metrics are calculated as follows:

$$Sensitivity = TP / (TP + FN)$$
$$Specificity = TN / (TN + FP)$$
$$PPV = TP / (TP + FP)$$
$$NPV = TN / (TN + FN)$$

Where TP = true positives (vulnerabilities correctly detected), TN = true negatives (safe code correctly identified), FP = false positives (safe code flagged as vulnerable), and FN = false negatives (vulnerabilities missed).

The goal is to balance the developer speed with performance (running a real-time system, for instance, checks (IDE integration, pre-commit hooks) that have a speed target of < 500ms p95 latency, full pipeline security analysis of < 5 minutes p95 latency and a throughput of over 100 requests/second of sustained load, 100+ requests/sec burst capacity to 1000+ requests/sec etc. Operational KPIs that determine its focus would be Mean Time to Detect < 1 hour; Mean Time to Remediate < 4 hours for all critical vulnerabilities and false positives < 15% and reduction in security debt 30% -50% within 6 months.

A large part of the critical evaluation element analyses whether code produced using an iterative security gate and without an iterative security gate is better or worse. To generate code with same prompt and requirement, the code is generated using two channels – our proposed framework, with a fully built-in iterative gate mechanism and a minimal LLM generation baseline allowing 5 iterations without security checkpoints. Blind security auditors do manual reviews, manually validating the correctness of the results of vulnerability counts and severity distributions to directly verify whether iterative gates prevent a documented 37.6% vulnerability increase.

7. Discussion

The introduction of this framework allows us to overcome existing gaps within security practices by covering all aspects of SDLCs, thus moving security left to identify

and prevent vulnerabilities as early as possible where they should be caught and remediation costs minimized. The iterative security gate mechanism in the application addresses a documented decline in security of AI-generated code, which is not found in popular code generation tools (e.g., GitHub Copilot, ChatGPT). Security specialization sets the proposal apart from generic multi-agent frameworks that don't have domain-specific security experience, integrate tools, or optimize prompting security.

The hybrid of rule-based tools and LLM reasoning builds both: rule-based tools deliver deterministic, auditable output, while LLMs provide context and natural language explanations. These can involve considerations like the technical stack used (LangGraph to manage complex states, CrewAI to orchestrate simpler threads), which kind of LLM provider is selected (Claude Sonnet 4/GPT-4o for more complex reasoning models, smaller models for specific tasks), as well as scalability requirements. Cost-analysis should consider LLM API costs [17][18] totaling about $164,000 a year for an enterprise managing 10,000 code generation requests daily, although stopping one (often serious) breach (typically costing $4.35M) [19] would put up significant return on investment.

Limitations are needed to establish the quality standards of LLMs with hallucinations or errors that could limit the efficiency of security testing, false positive/negative trade-offs that require fine tuning threshold settings, integration requirements requiring the existing DevSecOps maturity [20], development & security teams having to learn up to the software usually taking 3-6 months to build, and context-limiting analysis of very large codebases over 100,000 lines.

This framework offers significant value to regulated sectors (such as healthcare or finance) that need evidence of demonstrable security in place, with applicability varying between startup contexts where it establishes security practices early versus enterprise environments where it augments existing security teams and processes. Possible future improvements might involve domain-sensitive fine-tuning for industry verticals, federated learning providing an opportunity for organizations to exchange security insights without publicizing sensitive code, integrating SBOM (Software Bill of Materials) to secure supply chains, growing beyond web/API development efforts into mobile and IoT technology or securing AI supply chains to prevent the model from falling into a security pit.

8. Conclusion and Future Work

This paper introduces a new, multi-agent security framework for AI-assisted development. Its three major strengths are that it comprises seven specialized security agents spanning the entire SDLC, performs an iterative security gate mechanism to prevent documented vulnerabilities from rising with code updates, and seamlessly integrates with existing development tools. With a model that moves security to the left and makes security an integrated consideration from requirements through production monitoring, security becomes more than just a reactive constraint and will act as a proactive engine of development velocity.

The hybrid approach integrating rule-based security tools with LLM-powered contextual analysis not only utilizes the benefits of both approaches but also reduces each other's weaknesses. The confidence score and human-in-the-loop protocol for escalation are built to balance automation benefits with safety concerns to address concerns of applying AI in security-critical settings. The proposed evaluation framework ensures a high level of assessment aiming for > 90% sensitivity of critical vulnerabilities and > 85% specificity for reduction of alert fatigue.

Future research includes domain-specific fine-tuning of agents (healthcare agents for nuances of HIPAA, fintech agents optimized for PCI-DSS), federated learning to allow firms to share security expertise without revealing sensitive code, integration with SBOM generation and analysis for more secure supply chain, extending beyond web and API development to mobile, embedded systems and IoT devices, and an AI supply chain security investigation to ensure that AI models themselves are not compromised and not making the deliberate generation of vulnerable code.

References

1. J. Becker, N. Rush, E. Barnes, and D. Rein, "Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity," *arXiv.org*, 2025. https://arxiv.org/abs/2507.09089
2. "GitHub CEO says Copilot will write 80% of code 'sooner than later,'" *Freethink*, Jun. 17, 2023. https://www.freethink.com/robots-ai/github-copilot
3. "Examining Zero-Shot Vulnerability Repair with Large Language Models | IEEE Conference Publication | IEEE Xplore," *ieeexplore.ieee.org*. https://ieeexplore.ieee.org/abstract/document/10179324
4. "Peer-reviewed and accepted in IEEE-ISTAS 2025 Security Degradation in Iterative AI Code Generation: A Systematic Analysis of the Paradox," *Arxiv.org*, 2025. https://arxiv.org/html/2506.11022
5. S. Hong et al., "MetaGPT: Meta Programming for Multi-Agent Collaborative Framework," *arXiv.org*, Aug. 07, 2023. https://arxiv.org/abs/2308.00352
6. Q. Wu et al., "AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation," *arXiv.org*, Oct. 03, 2023. https://arxiv.org/abs/2308.08155
7. "Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity," *METR Blog*, 2025. https://metr.org/blog/2025-07-10-early-2025-ai-experienced-os-dev-study/
8. Z. Li, S. Dutta, and M. Naik, "LLM-Assisted Static Analysis for Detecting Security Vulnerabilities," *arXiv.org*, 2024. https://arxiv.org/abs/2405.17238
9. N. Perry, M. Srivastava, D. Kumar, and D. Boneh, "Do Users Write More Insecure Code with AI Assistants?," *arXiv (Cornell University)*, Nov. 2022, doi: https://doi.org/10.1145/3576915.3623157
10. S. Stein, "CrowdStrike Researchers Identify Hidden Vulnerabilities in AI-Coded Software,"

Crowdstrike.com, 2025. https://www.crowdstrike.com/en-us/blog/crowdstrike-researchers-identify-hidden-vulnerabilities-ai-coded-software/
11. Y. Dong et al., "A Survey on Code Generation with LLM-based Agents," *Arxiv.org*, 2025. https://arxiv.org/html/2508.00083v1
12. "The Evolution of DevSecOps with AI | CSA," *Cloudsecurityalliance.org*, 2024. https://cloudsecurityalliance.org/blog/2024/11/22/the-evolution-of-devsecops-with-ai
13. "PLOT4ai - Privacy Library Of Threats 4 Artificial Intelligence," *plot4.ai*. https://plot4.ai/
14. Umm-e- Habiba, M. Haug, J. Bogner, and S. Wagner, "How mature is requirements engineering for AI-based systems? A systematic mapping study on practices, challenges, and future research directions," *Requirements Engineering*, Oct. 2024, doi: https://doi.org/10.1007/s00766-024-00432-3
15. J. P. Morgan, "Revolutionizing Threat Modeling with AI: The Threat Modeling Co-Pilot," *Jpmorganchase.com*, Oct. 03, 2025. https://www.jpmorganchase.com/about/technology/blog/aitmc
16. B. Strom, A. Applebaum, D. Miller, K. Nickels, A. Pennington, and C. Thomas, "MITRE ATT&CK®: Design and Philosophy," Jul. 2018. Available: https://www.mitre.org/sites/default/files/2021-11/prs-19-01075-28-mitre-attack-design-and-philosophy.pdf
17. "Pricing," *www.anthropic.com*. https://www.anthropic.com/pricing
18. OpenAI, "API Pricing," *OpenAI*, 2025. https://openai.com/api/pricing/
19. IBM, "IBM Report: Consumers Pay the Price as Data Breach Costs Reach All-Time High," *IBM Newsroom*, Jul. 27, 2022. https://newsroom.ibm.com/2022-07-27-IBM-Report-Consumers-Pay-the-Price-as-Data-Breach-Costs-Reach-All-Time-High
20. "OWASP Top 10 CI/CD Security Risks | OWASP Foundation," *owasp.org*. https://owasp.org/www-project-top-10-ci-cd-security-risks/