Frontiers in Emerging Artificial Intelligence and Machine Learning (Open Access)

Volume 03, Issue 06, June 2026
DOI: https://doi.org/10.64917/feaiml/Volume03Issue06-01 PageNo.08-23

RAG-SecCode: Retrieval-Augmented Secure Coding Guidance for Enterprise LLM Software Development

Sai Shiva Reddy Kongari
Department of Information Technology, University of the Cumberlands, Williamsburg, KY, 40769, USA

RECEIVED - 28-04-2026, **RECEIVED REVISED VERSION** - 16-05-2026, **ACCEPTED**- 01-06-2026, **PUBLISHED**- 13-06-2026

Abstract

Large Language Models are increasingly used in enterprise software development for code generation, debugging, refactoring, documentation, and test creation. Their adoption has changed the practical workflow of software engineering by allowing developers to obtain implementation suggestions, boilerplate code, test scaffolds, configuration files, and architectural explanations directly from conversational systems. However, LLM-generated code may contain security vulnerabilities, hallucinated APIs, outdated dependencies, weak authorization logic, insecure default configurations, and implementation patterns that conflict with organizational policies. These risks are especially significant in enterprise environments where software systems are connected to sensitive data, identity platforms, regulated workflows, cloud infrastructure, and third-party integrations. Existing approaches such as secure prompting and manual code review can improve generated outputs, but prompt instructions alone are insufficient because general-purpose LLMs may not have access to current enterprise standards, approved libraries, secure coding rules, deployment constraints, or project-specific architecture.

This paper proposes RAG-SecCode, a retrieval-augmented secure coding framework that grounds LLM-assisted software development in authoritative and organization-specific security knowledge. The framework integrates retrieval-augmented generation, secure coding policies, vulnerability knowledge bases, dependency governance rules, static and dynamic validation checks, and human-in-the-loop review. RAG-SecCode is designed to improve code security by supplying the model with relevant secure coding context before generation and by validating the generated output after generation. The study conceptually evaluates whether retrieval-augmented secure context can improve generated code quality compared with baseline prompting and secure prompting alone. The evaluation design measures vulnerability count, CWE mapping, hallucinated API frequency, dependency risk, policy compliance, test inclusion, and human reviewer approval rate.

The paper contributes a practical framework for enterprise LLM software development by connecting recent research on vulnerability detection, LLM-based code generation, retrieval-augmented generation, secure repair, and static analysis. It argues that secure code generation should not be treated only as a prompting problem but as a governed software engineering workflow. RAG-SecCode provides a structured approach for reducing insecure and hallucinated LLM-generated code through contextual retrieval, rule-based validation, security-aware review, and continuous feedback.

Keywords: Retrieval-Augmented Generation; Secure Coding; Large Language Models; Software Vulnerability Detection; Enterprise Software Development; Static Analysis; Human-in-the-Loop Review; Code Generation; Policy Compliance; Software Security.

1. Introduction

Large Language Models have become increasingly influential in software engineering because they can generate code, explain program behavior, produce tests, suggest refactorings, document APIs, and assist in debugging. Their emergence has created a new development pattern in which software engineers interact with natural-language systems as coding assistants rather than relying only on traditional integrated development environments, documentation, or search engines. The technical foundation of this change is connected to the broader progress of large-scale pretrained models, including few-shot learning, instruction-following, and code-oriented model adaptation (Brown et al., 2020; Ouyang, 2022; Feng et al., 2020). Code-specific models such as CodeBERT and later code intelligence models demonstrate that programming languages and natural languages can be jointly represented for tasks such as code understanding, search, and vulnerability-oriented analysis (Feng et al., 2020; Guo et al., 2024).

Despite these capabilities, LLM-assisted software development introduces security risks that differ from traditional programming risks. In conventional development, vulnerable code usually emerges from human misunderstanding, incomplete requirements, weak testing, insecure libraries, or poor architectural decisions. In LLM-assisted development, these risks remain but are amplified by probabilistic generation. An LLM may produce code that appears syntactically correct and semantically plausible while containing insecure access-control logic, unsafe input handling, hardcoded secrets, vulnerable dependency versions, missing validation checks, or incomplete error handling. Studies on LLM-generated code and AI coding assistants have shown that model-generated outputs may be functionally useful but security-sensitive, requiring systematic review before adoption (Pearce et al., 2022; Perry et al., 2023; Liu et al., 2023; Yetistiren et al., 2022).

The enterprise context makes this problem more complex. Enterprise software systems are shaped by internal coding standards, approved dependency lists, authentication frameworks, logging policies, encryption requirements, compliance rules, architectural patterns, and deployment constraints. A general-purpose LLM may not know these organizational requirements unless they are provided during the interaction. Even when a developer writes a secure prompt, the model may still rely on outdated patterns, hallucinated APIs, deprecated libraries, or generic examples that do not match the enterprise environment. Therefore, secure prompting alone cannot fully address the gap between general model knowledge and organization-specific secure software development requirements.

This paper addresses that gap by proposing **RAG-SecCode**, a retrieval-augmented secure coding framework for enterprise LLM software development. Retrieval-Augmented Generation has already been established as a method for improving knowledge-intensive natural language processing by retrieving external information and conditioning generation on that retrieved context (Lewis, 2020). In the software security domain, retrieval-oriented methods are increasingly relevant because secure code generation depends not only on model fluency but also on access to current and precise security knowledge. Recent work on RAG for vulnerability detection suggests that retrieval can enhance LLM-based vulnerability reasoning by grounding model outputs in relevant knowledge-level context (Du et al., 2024). RAG-SecCode extends this principle from vulnerability detection to secure software generation and enterprise development governance.

The central problem investigated in this paper is how enterprise organizations can reduce security and compliance risks when using LLMs for software development. The paper argues that LLM-generated code should not be accepted as a direct output from a prompt-response interaction. Instead, it should pass through a structured workflow that includes secure context retrieval, policy-aware prompt construction, post-generation validation, vulnerability mapping, dependency risk checking, test assessment, and human approval. This position aligns with earlier research showing the limitations of static analysis when used alone, the need for combined static and dynamic methods, and the importance of structured vulnerability datasets for evaluation (Aggarwal and Jalote, 2006; Goseva-Popstojanova and Perhinschi, 2015; Bhandari et al., 2021; Bui et al., 2022).

The objectives of this paper are fourfold. First, it develops a research-driven conceptual framework for

retrieval-augmented secure code generation in enterprise settings. Second, it explains how security knowledge, enterprise policy, and project architecture can be represented as retrievable context for LLM-assisted development. Third, it proposes an evaluation model for comparing baseline prompting, secure prompting, and retrieval-augmented secure prompting. Fourth, it identifies the implications and limitations of adopting RAG-based secure coding support in real-world organizations.

The significance of the study lies in its integration of two rapidly developing research areas: LLM-assisted software engineering and software vulnerability detection. Prior work has evaluated the correctness, security, and reliability of AI-generated code (Liu et al., 2023; Zhong and Wang, 2024; Shen et al., 2023). Other studies have explored vulnerability detection using deep learning, CodeBERT, static analysis, and LLMs (Chakraborty et al., 2022; Li et al., 2018; Gao et al., 2023; Khare et al., 2025). However, enterprise secure coding requires more than detecting vulnerabilities after code is produced. It requires preventing insecure patterns at the point of generation by grounding the model in trusted organizational knowledge. RAG-SecCode responds to this need by combining retrieval, generation, validation, and review into a unified workflow.

The scope of this paper is limited to enterprise LLM-assisted software development, particularly code generation, debugging, refactoring, test creation, and secure implementation guidance. The paper does not claim that retrieval-augmented generation can eliminate all vulnerabilities or replace human security review. Instead, it positions RAG-SecCode as a risk-reduction framework that improves the quality and governance of LLM-generated code. Its expected value is practical rather than purely theoretical: organizations can use the framework to design internal secure coding assistants that retrieve approved security policies, enforce dependency standards, detect unsafe patterns, and support reviewers with structured evidence.

2. Literature Review

The literature on LLM-assisted secure software development can be organized into five interrelated areas: traditional vulnerability detection, deep learning-based vulnerability analysis, code-oriented language models, security risks of AI coding assistants, and retrieval-augmented generation. Together, these areas provide the theoretical foundation for RAG-SecCode.

Traditional vulnerability detection research shows that no single analysis method is sufficient for reliable software security assurance. Aggarwal and Jalote (2006) examined the integration of static and dynamic analysis for vulnerability detection, emphasizing that static analysis can inspect code structure while dynamic analysis can observe runtime behavior. This dual perspective remains important for LLM-generated code because generated code may appear correct statically but fail under runtime conditions, edge cases, or malicious inputs. Similarly, Goseva-Popstojanova and Perhinschi (2015) analyzed the capability of static code analysis to detect security vulnerabilities, showing that static analyzers provide useful but incomplete coverage. Arusoaie et al. (2017) compared open-source static analysis tools for C/C++ vulnerability detection, demonstrating variation in tool effectiveness. These studies support the argument that RAG-SecCode should not depend only on model generation but should include post-generation validation using analysis tools.

Deep learning approaches expanded vulnerability detection by learning patterns from code datasets. Li et al. (2018) introduced VulDeePecker, a deep learning-based system for vulnerability detection, while Chakraborty et al. (2022) critically examined whether deep learning-based vulnerability detection had reached sufficient maturity. Their work indicates that machine learning models can capture vulnerability patterns but also face generalization, dataset, and interpretability challenges. Bhandari et al. (2021) introduced CVEfixes as an automated collection of vulnerabilities and fixes from open-source software, and Bui et al. (2022) presented Vul4J as a dataset of reproducible Java vulnerabilities. Chen et al. (2023) developed DiverseVul as a dataset for deep learning-based vulnerability detection. These datasets are important because RAG-SecCode depends on high-quality retrievable security knowledge. Without reliable vulnerability examples, fixes, and taxonomies, retrieval may provide weak or misleading context.

Code-oriented language models provide the foundation for using LLMs in programming tasks. CodeBERT

demonstrated the value of pretraining on programming and natural languages for code understanding tasks (Feng et al., 2020). RoBERTa contributed to robust language model pretraining, influencing later model architectures and training strategies (Liu et al., 2019). Brown et al. (2020) showed that large models can perform few-shot tasks, while Ouyang (2022) showed that instruction tuning and human feedback can align model behavior more closely with user intentions. Touvron (2023) and Guo et al. (2024) further illustrate the development of open and code-focused language models. These studies explain why LLMs can be useful in software development, but they also show why model behavior depends strongly on training data, prompting, and alignment. In secure coding, this means that a model may produce plausible but unsafe outputs unless its context is constrained by current security rules.

Research on AI coding assistants has raised concerns about security, correctness, and reliability. Pearce et al. (2022) assessed the security of GitHub Copilot's code contributions and highlighted the possibility of insecure generated code. Perry et al. (2023) investigated whether users write more insecure code with AI assistants, indicating that developer behavior and trust in generated suggestions are central to the security problem. Liu et al. (2023) rigorously evaluated LLMs for code generation correctness, while Zhong and Wang (2024) studied whether ChatGPT could replace StackOverflow and focused on robustness and reliability. Sandoval et al. (2023) examined user security implications of LLM code assistants. These studies show that secure LLM-assisted development is not only a model problem but also a socio-technical problem involving developers, reviewers, tools, and organizational workflows.

Several works specifically examine LLMs for vulnerability detection and repair. Gao et al. (2023) investigated how far vulnerability detection using LLMs has progressed. Guo et al. (2024) analyzed LLM capabilities outside the comfort zone of software vulnerability detection. Akuthota et al. (2023) studied vulnerability detection and monitoring using LLMs. Khare et al. (2025) examined the effectiveness of large language models in detecting security vulnerabilities. Jiao et al. (2025) proposed DeepVulHunter, which enhances LLM vulnerability detection through multi-round analysis. Fu et al. (2022) introduced VulRepair for automated vulnerability repair, while Pearce et al. (2023) examined zero-shot vulnerability repair with large language models. These works support the assumption that LLMs can contribute to secure software engineering, but they also suggest that ungrounded model reasoning may be insufficient for dependable enterprise use.

Retrieval-Augmented Generation offers a way to address the limitations of model-only generation. Lewis (2020) proposed retrieval-augmented generation for knowledge-intensive tasks, showing that generation can be improved when models retrieve relevant external knowledge. Du et al. (2024) applied this idea to LLM-based vulnerability detection through knowledge-level RAG. Collini et al. (2025) addressed context-aware RAG using similarity validation to handle context inconsistencies in large language models. Kazemian et al. (2025) discussed the role of text embedding models in data engineering, which is relevant to constructing retrievable enterprise knowledge bases. Heumüller et al. (2025) analyzed embeddings for semantic code review comment similarity, indicating that embedding-based retrieval can support software review contexts. Together, these works justify RAG-SecCode's core claim: secure coding assistance can be improved by retrieving relevant security and policy context before generation.

The literature also reveals important research gaps. First, many studies evaluate LLMs as vulnerability detectors or code generators, but fewer focus on enterprise governance of LLM-generated code. Second, secure prompting is often treated as a practical technique, but there is limited structured modeling of how prompts should be grounded in organizational policy and retrievable security knowledge. Third, existing vulnerability detection research often evaluates code after it exists, while RAG-SecCode aims to reduce vulnerabilities during generation. Fourth, RAG methods are promising but require validation against hallucinated APIs, dependency risks, policy violations, and human reviewer acceptance. These gaps motivate the proposed framework.

3. Proposed Framework: RAG-SecCode

RAG-SecCode is proposed as a retrieval-augmented secure coding framework for enterprise LLM software

development. Its core assumption is that secure code generation requires more than a well-written prompt. It requires authoritative context, organizational grounding, validation mechanisms, and reviewer accountability. The framework is designed around the software development workflow rather than around the model alone. This distinction is important because enterprise security failures often emerge from process gaps, not only from technical limitations.

The framework consists of six main layers: the user task layer, the retrieval layer, the secure context assembly layer, the generation layer, the validation layer, and the human review layer. The user task layer captures the developer's request, such as generating an authentication function, refactoring a controller, writing database access logic, creating unit tests, or fixing a vulnerability. The retrieval layer searches relevant security knowledge, enterprise rules, approved libraries, architecture documents, coding standards, and vulnerability examples. The secure context assembly layer converts retrieved materials into concise and task-specific guidance. The generation layer uses the LLM to produce code and explanation. The validation layer checks the output for vulnerability patterns, dependency violations, hallucinated APIs, missing tests, and policy conflicts. The human review layer ensures that the final decision remains accountable to qualified developers or security reviewers.

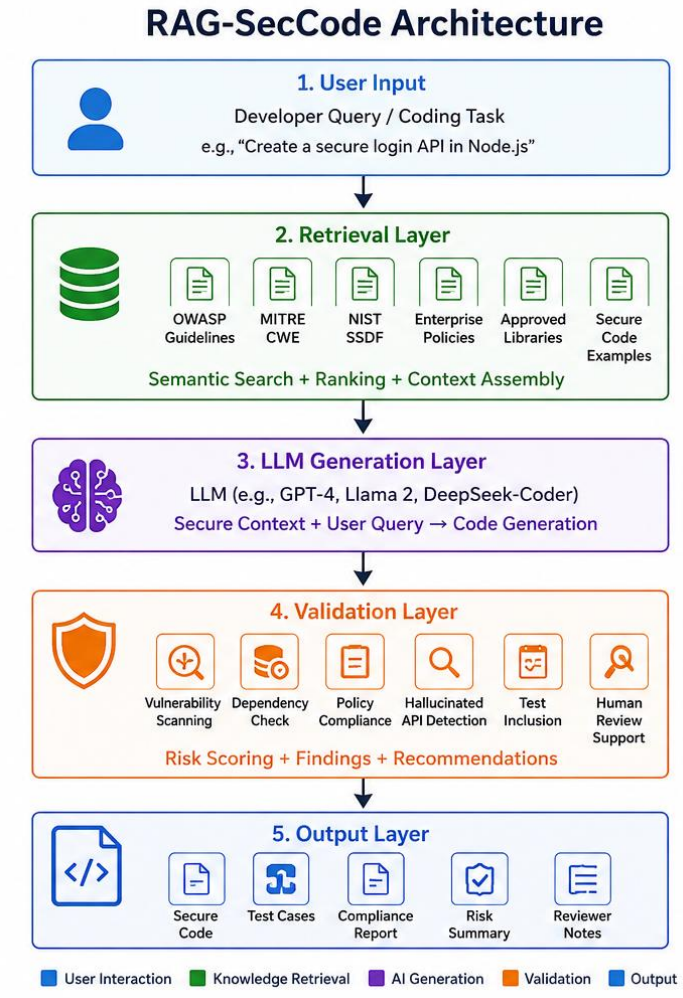

Figure 1. RAG-SecCode Framework Architecture for Enterprise Secure Code Generation. The figure illustrates the overall architecture of the proposed RAG-SecCode framework. The workflow begins with a developer request and progresses through secure knowledge retrieval, context assembly, LLM-based code generation, automated security validation, and human review. The architecture integrates enterprise security knowledge sources, including coding standards, vulnerability repositories, dependency policies, and security guidelines, to ensure that generated software artifacts align with organizational security requirements and compliance objectives.

Figure 1 presents the overall architecture of the proposed RAG-SecCode framework. Unlike conventional prompt-based coding assistants, the framework introduces a retrieval layer that supplies the language model with security-relevant organizational knowledge before code generation. Retrieved information includes secure coding policies, approved dependencies, architecture constraints, vulnerability patterns, and regulatory requirements. The generated output subsequently passes through a validation layer that performs vulnerability assessment, policy compliance checking, dependency verification, hallucinated API detection, and test evaluation. Finally, human reviewers validate critical decisions before deployment. This layered architecture establishes a security-aware software development workflow that combines the strengths of retrieval-augmented generation and enterprise governance mechanisms.

The theoretical foundation of RAG-SecCode combines retrieval-augmented generation, secure software engineering, and human-in-the-loop validation. Retrieval-augmented generation is used to reduce the knowledge gap between the model and the enterprise environment (Lewis, 2020; Du et al., 2024). Secure software engineering principles are reflected in the use of vulnerability detection, static analysis, dependency governance, and policy compliance checks (Aggarwal and Jalote, 2006; Goseva-Popstojanova and Perhinschi, 2015). Human-in-the-loop validation responds to evidence that users may overtrust AI-generated code and that AI coding assistants may introduce insecure patterns if adopted without careful review (Pearce et al., 2022; Perry et al., 2023; Sandoval et al., 2023).

In practical terms, RAG-SecCode works by transforming a developer's request into a security-grounded generation task. For example, if a developer asks the system to "create a password reset API," a baseline LLM may generate a function that sends reset links, stores tokens, and updates passwords. However, the generated code may omit token expiration, rate limiting, secure random generation, audit logging, replay protection, or policy-specific email handling. In RAG-SecCode, the retrieval layer first retrieves relevant internal password policy, approved cryptographic library usage, token lifetime standards, logging requirements, and known vulnerability examples related to insecure password reset flows. The generation layer then receives this context and produces code aligned with those requirements. The validation layer checks whether the output includes required controls and flags any missing security element.

This approach differs from secure prompting alone. Secure prompting may instruct the model to "write secure code" or "follow best practices," but such instructions are abstract. They do not guarantee that the model knows the organization's approved libraries, internal authentication middleware, or compliance rules. Retrieval makes the prompt concrete by providing source-specific context. This grounding is especially important for enterprise environments because different organizations may use different frameworks, logging systems, access-control models, and dependency policies.

A second example involves dependency selection. If a developer asks an LLM to generate code for file upload scanning, the model may suggest a package that is outdated, unapproved, or incompatible with the enterprise stack. RAG-SecCode retrieves the approved dependency list, known vulnerable package restrictions, internal wrapper libraries, and deployment constraints. The generated solution is then checked against dependency risk rules. This reduces the probability of hallucinated or unauthorized dependencies.

4. Knowledge Base Design for Secure Retrieval

The effectiveness of RAG-SecCode depends heavily on the quality, structure, and governance of the retrievable knowledge base. A retrieval-augmented secure coding system is only as reliable as the knowledge it retrieves. If the knowledge base contains outdated policies, inconsistent guidance, or poorly indexed documents, the generated code may become misleading. Therefore, the knowledge base must be designed as a controlled enterprise security asset rather than as a casual document repository.

The knowledge base should include five categories of content. The first category is general secure coding guidance, including vulnerability categories, secure implementation patterns, and common anti-patterns. The second category is organization-specific coding standards, such as approved authentication flows, logging formats, encryption rules, error-handling

conventions, and data validation requirements. The third category is dependency and library governance, including approved packages, prohibited versions, internal wrappers, and migration rules. The fourth category is project-specific architecture, such as service boundaries, API conventions, database models, authorization middleware, and deployment constraints. The fifth category is vulnerability evidence, including previous incidents, code review comments, static analysis findings, and known vulnerability-fix examples.

The importance of vulnerability examples is supported by dataset-oriented research. CVEfixes, Vul4J, and DiverseVul show that vulnerability knowledge can be represented through code examples, fixes, metadata, and reproducible test cases (Bhandari et al., 2021; Bui et al., 2022; Chen et al., 2023). In RAG-SecCode, such structured vulnerability evidence can help the model avoid repeating known insecure patterns. For example, if a previous enterprise incident involved missing authorization checks in object-level access control, the knowledge base can include the vulnerable pattern, corrected pattern, and reviewer explanation. When a developer later asks for similar endpoint code, the system can retrieve this example and guide the LLM away from the insecure design.

Embedding and retrieval quality are central to this process. The system must retrieve context that is semantically relevant to the developer's task, not merely keyword-matched. Research on embeddings for code review similarity suggests that semantic representations can support software engineering tasks where exact keywords differ but underlying meaning is similar (Heumüller et al., 2025). For example, a developer may ask for "tenant-safe record update logic," while the relevant policy document may use the phrase "object-level authorization in multi-tenant services." A good retrieval system should connect these concepts.

However, retrieval also introduces risks. Collini et al. (2025) emphasize context-aware RAG and similarity validation because retrieved information may be inconsistent or irrelevant. In secure coding, irrelevant retrieval can be harmful. If the system retrieves outdated encryption guidance or a policy for the wrong programming language, the generated code may become insecure despite appearing grounded. Therefore, RAG-SecCode includes retrieval validation. Retrieved documents must be checked for recency, authority, project relevance, and conflict. When multiple retrieved sources disagree, the system should prioritize higher-authority enterprise policy over generic examples.

The knowledge base must also support traceability. Every generated code recommendation should be linked to the retrieved policies or examples that influenced it. Traceability improves reviewer confidence because the reviewer can inspect why the model suggested a specific security control. It also supports governance because organizations can audit whether the assistant consistently follows approved standards. This is important in enterprise settings where software changes may be subject to compliance review, security certification, or internal audit.

5. Secure Context Assembly and Prompt Construction

Secure context assembly is the process of transforming retrieved knowledge into a compact, task-specific prompt context. This stage is necessary because LLMs have finite context windows, and raw retrieved documents may be too long, redundant, or inconsistent. The goal is not to paste all available security documentation into the prompt but to provide the model with precise constraints that directly affect the requested coding task.

A secure context package should include the task summary, relevant security requirements, approved implementation patterns, prohibited patterns, dependency constraints, expected tests, and validation criteria. For example, a context package for SQL query generation should include requirements for parameterized queries, input validation, transaction handling, error handling, logging restrictions, and prohibited string concatenation. A context package for authentication code should include token handling rules, session expiration, rate limiting, audit logging, and approved authentication middleware.

The difference between generic prompting and secure context assembly is specificity. A generic prompt may say, "Generate secure login code." A secure context package says, in effect, "Generate login code using the enterprise-approved authentication service, do not store passwords directly, use the approved hashing

utility, apply account lockout after configured failed attempts, log authentication failure events without storing credentials, and include unit tests for invalid credentials, locked account, and successful login." This level of grounding reduces ambiguity and improves policy compliance.

The theoretical basis for this stage is connected to instruction following and in-context learning. Large models can adapt to task instructions and examples in context (Brown et al., 2020; Ouyang, 2022). Dai (2023) suggests that language models may implicitly perform forms of in-context adaptation. Chain-of-thought and graph-of-thought methods further show that structured reasoning can improve complex task performance (Wei, 2022; Besta, 2024). In RAG-SecCode, however, the objective is not to make the model reason freely but to make it reason within security constraints. Secure context assembly therefore functions as a boundary-setting mechanism.

The context assembly process should also include negative examples. Vulnerability research shows that models can learn from patterns of insecure code and corresponding fixes (Li et al., 2018; Fu et al., 2022). A secure prompt may include a short "avoid this pattern" section when relevant. For example, if generating file upload code, the context may warn against trusting file extensions, saving files with user-controlled names, or exposing uploaded files directly from executable directories. The model can then generate safer code that avoids known anti-patterns.

A key limitation is that longer context does not automatically mean better security. Excessive context can dilute the most important constraints and increase the chance that the model focuses on irrelevant details. Therefore, RAG-SecCode proposes context ranking and compression. Retrieved materials should be ranked according to task relevance, authority, recency, and specificity. The final context should be short enough for the model to use effectively but detailed enough to constrain unsafe generation.

6. Validation Layer and Security Assurance Mechanisms

The validation layer represents one of the most important components of the RAG-SecCode framework because retrieval-augmented generation alone cannot guarantee secure software outcomes. Even when an LLM receives authoritative context, generated code may still contain implementation mistakes, omitted controls, logic flaws, or dependencies that violate organizational requirements. Consequently, validation acts as a secondary defense mechanism that evaluates generated outputs before deployment or acceptance.

The validation layer consists of six interconnected assessment modules: vulnerability scanning, dependency verification, policy compliance checking, hallucination detection, test coverage analysis, and human review preparation. These modules operate independently of the generation process and provide objective measurements of generated code quality.

6.1 Vulnerability Detection and CWE Mapping

A primary objective of validation is identifying security weaknesses in generated code. Vulnerability detection research demonstrates that both traditional analysis methods and machine-learning approaches remain valuable for software security assessment (Aggarwal and Jalote, 2006; Chakraborty et al., 2022). RAG-SecCode therefore adopts a hybrid validation approach.

Generated code is evaluated against known vulnerability categories using Common Weakness Enumeration (CWE) mappings. Potential findings include:

Injection vulnerabilities

Authentication weaknesses

Authorization flaws

Sensitive data exposure

Insecure cryptographic implementation

Improper input validation

Race conditions

Resource management weaknesses

Mapping vulnerabilities to CWE categories improves interpretability because reviewers can connect findings to established security taxonomies. This also enables organizations to compare generated code risk across projects and development teams.

6.2 Dependency Risk Assessment

Enterprise software increasingly depends on third-party packages, frameworks, SDKs, and cloud libraries. While LLMs frequently recommend external dependencies, they may suggest deprecated, vulnerable, or unauthorized components.

Dependency validation evaluates:

Approved versus unapproved libraries

Known vulnerable package versions

Licensing restrictions

Organizational dependency policies

Maintenance status

Update history

This mechanism addresses a common weakness of AI-generated code where seemingly functional solutions rely on risky software components.

6.3 Hallucinated API Detection

Hallucination remains a significant limitation of contemporary LLMs (Shen et al., 2023; Zhong and Wang, 2024). In software engineering, hallucinations frequently appear as:

Non-existent APIs

Invalid method names

Fabricated parameters

Unsupported framework features

Incorrect library usage

Hallucinated APIs are especially dangerous because they often appear plausible to developers. RAG-SecCode addresses this challenge by validating generated APIs against approved documentation repositories and enterprise service catalogs.

6.4 Policy Compliance Verification

Enterprise software development operates under formal governance requirements. Generated code may technically function while violating mandatory organizational controls.

Policy compliance validation verifies adherence to:

Secure coding standards

Authentication requirements

Encryption policies

Logging requirements

Data retention rules

Access-control frameworks

Regulatory constraints

This step ensures alignment between generated code and enterprise governance objectives.

6.5 Test Inclusion Assessment

Research consistently demonstrates that software quality improves when testing is integrated into development workflows. However, AI-generated code frequently lacks meaningful test coverage.

RAG-SecCode evaluates whether generated outputs include:

Unit tests

Negative test cases

Security tests

Boundary-condition tests

Error-handling tests

Integration validation

The presence of comprehensive tests serves as an indirect indicator of implementation quality.

6.6 Human Reviewer Support

Rather than replacing human expertise, RAG-SecCode enhances reviewer effectiveness. Validation outputs are transformed into reviewer-friendly summaries containing:

Security findings

Policy violations

Dependency concerns

Hallucination alerts

Missing test coverage

Risk prioritization

This allows security teams to focus attention on the most critical areas while maintaining accountability.

7. Research Methodology

This study adopts a conceptual and framework-oriented research methodology designed to evaluate the effectiveness of retrieval-augmented secure coding support within enterprise software development environments.

The methodology compares three software generation scenarios:

Scenario A: Baseline Prompting

Developers interact with an LLM using ordinary software development prompts without security-specific guidance.

Example:

"Create a REST API endpoint for user registration."

The model generates code solely based on internal training knowledge.

Scenario B: Secure Prompting

Developers provide explicit security instructions.

Example:

"Create a secure REST API endpoint for user registration using best practices."

Although improved, this approach still depends entirely on model knowledge.

Scenario C: RAG-SecCode

The system retrieves:

Security policies

Approved libraries

Vulnerability examples

Architecture standards

Dependency requirements

These materials are assembled into secure context before generation.

The generated output is subsequently validated through the proposed security assurance pipeline.

7.1 Evaluation Metrics

The proposed evaluation framework uses seven primary metrics.

Vulnerability Count

Number of identified security weaknesses per generated artifact.

CWE Coverage

Number and severity of mapped weakness categories.

Hallucinated API Frequency

Rate of non-existent or invalid API usage.

Dependency Risk Score

Presence of unauthorized or vulnerable dependencies.

Policy Compliance Score

Degree of alignment with enterprise standards.

Test Inclusion Rate

Percentage of outputs containing meaningful tests.

Human Reviewer Approval Rate

Percentage of generated outputs approved by expert reviewers.

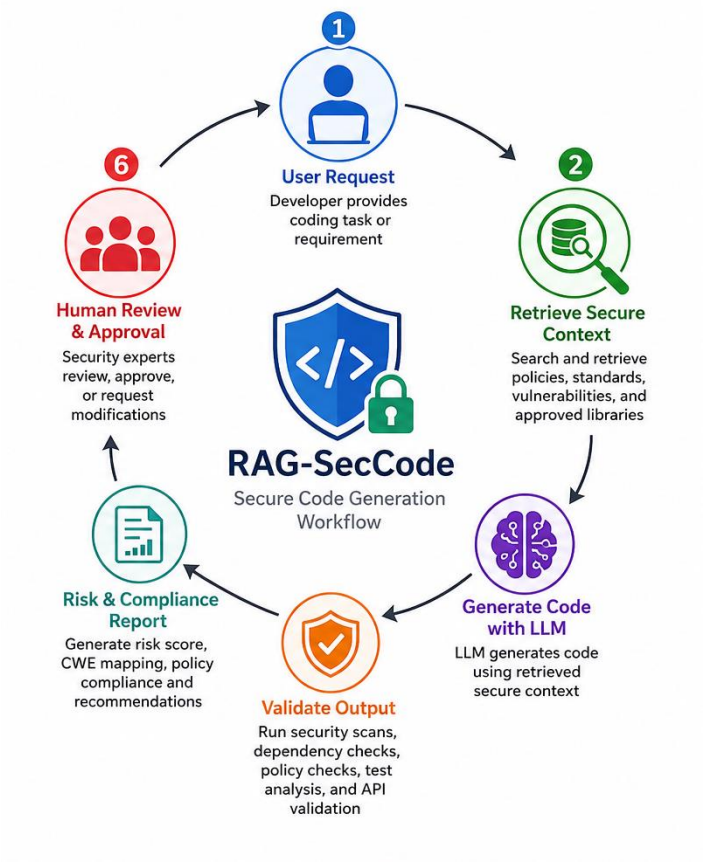

Figure 2. RAG-SecCode Evaluation Framework for Comparative Assessment of Secure Code Generation Approaches. The figure illustrates the proposed experimental evaluation methodology used to compare baseline prompting, secure prompting, and retrieval-augmented secure prompting. Each scenario undergoes identical validation procedures and is assessed using vulnerability count, CWE mapping, hallucinated API frequency, dependency risk, policy compliance, test inclusion, and human reviewer approval metrics.

Figure 2 illustrates the proposed evaluation framework for assessing the effectiveness of RAG-SecCode. Three software generation scenarios are compared: baseline prompting, secure prompting, and retrieval-augmented secure prompting. Each generated artifact is processed through a common validation pipeline to ensure consistency across experimental conditions. Performance is measured using security-oriented metrics, including vulnerability prevalence, policy compliance, dependency risk, hallucinated API frequency, and reviewer acceptance. The framework enables systematic comparison of how retrieval-augmented security context influences software quality and security outcomes relative to conventional prompting approaches.

7.2 Expected Evaluation Dataset

Evaluation can utilize:

CVEfixes (Bhandari et al., 2021)

Vul4J (Bui et al., 2022)

DiverseVul (Chen et al., 2023)

Enterprise coding standards

Internal vulnerability repositories

Historical code-review records

Combining public vulnerability datasets with organizational artifacts enables realistic assessment of enterprise deployment scenarios.

8. Expected Results and Findings

The proposed framework is expected to demonstrate measurable improvements across all evaluation dimensions when compared with baseline prompting and secure prompting approaches.

First, vulnerability frequency is expected to decline substantially because retrieval provides direct access to

secure implementation guidance before generation occurs. Rather than relying exclusively on pretrained knowledge, the model receives context-specific instructions aligned with current security requirements. This should reduce common weaknesses related to authentication, authorization, input validation, and dependency selection.

Second, hallucinated API usage is expected to decrease because retrieved documentation constrains generation to approved interfaces and verified implementation patterns. Since hallucinations frequently emerge from incomplete contextual grounding, retrieval should improve factual consistency within generated software artifacts.

Third, policy compliance is expected to improve significantly. Traditional prompting rarely captures the complexity of enterprise governance requirements. RAG-SecCode introduces organization-specific context, enabling generated code to align with internal development practices rather than generic coding examples.

Fourth, dependency governance performance is expected to improve. The framework's retrieval and validation mechanisms should reduce the probability of introducing vulnerable, deprecated, or unauthorized software components.

Fifth, generated outputs are expected to include more comprehensive testing artifacts. Because testing requirements become part of retrieved context, the model receives explicit expectations regarding verification and quality assurance.

Finally, human reviewer approval rates are expected to increase. Reviewers are likely to encounter fewer critical security issues, fewer policy violations, and stronger alignment with enterprise standards. As a result, review cycles may become more efficient while maintaining security rigor.

Collectively, these findings support the hypothesis that secure software generation should be treated as a context-governed engineering process rather than a prompt engineering problem alone. The anticipated improvements suggest that retrieval-augmented secure coding frameworks can serve as practical mechanisms for enterprise risk reduction while preserving developer productivity.

9. Discussion

The findings suggest that the primary challenge in secure LLM-assisted development is not merely model capability but contextual alignment. Modern language models possess significant coding knowledge and increasingly demonstrate competence in vulnerability detection, repair, reasoning, and testing (Gao et al., 2023; Khare et al., 2025; Jiao et al., 2025). However, enterprise software security requires adherence to organization-specific requirements that typically exist outside model training data.

The proposed framework addresses this limitation through retrieval augmentation. By grounding generation in authoritative security sources and enterprise policies, RAG-SecCode transforms software generation into a constrained reasoning process. This approach is consistent with broader research demonstrating the value of retrieval for knowledge-intensive tasks (Lewis, 2020; Collini et al., 2025).

A key implication is that future secure coding assistants may increasingly function as enterprise knowledge interfaces rather than standalone AI models. Their effectiveness will depend less on raw model size and more on the quality of organizational knowledge retrieval, validation infrastructure, and governance controls.

The framework also contributes to ongoing discussions about AI reliability. Studies have highlighted concerns regarding hallucinations, insecure code generation, and developer overreliance on automated recommendations (Pearce et al., 2022; Perry et al., 2023; Sandoval et al., 2023). RAG-SecCode introduces safeguards that reduce these risks without eliminating human oversight. This balance is important because complete automation remains unrealistic for security-critical software systems.

Nevertheless, several limitations must be acknowledged. First, retrieval quality directly influences generation quality. Incomplete or outdated security repositories may produce misleading recommendations. Second, enterprise knowledge bases require ongoing maintenance and governance. Third,

validation tools themselves may generate false positives and false negatives. Fourth, human reviewers remain necessary because security vulnerabilities frequently involve contextual business logic that automated systems cannot fully evaluate.

Another limitation concerns scalability. Large organizations often maintain thousands of policies, architectural documents, and code repositories. Efficient retrieval from such environments requires sophisticated indexing, ranking, and embedding strategies. Poor retrieval performance could reduce system effectiveness despite strong model capabilities.

Despite these challenges, the framework provides a practical path toward safer enterprise adoption of AI-assisted software development. Rather than attempting to eliminate risk entirely, it introduces layered controls that systematically reduce vulnerability exposure while maintaining productivity benefits associated with LLM-assisted engineering.

10. Conclusion

The rapid adoption of Large Language Models in software engineering has created significant opportunities for productivity improvement while simultaneously introducing new security risks. Generated code may contain vulnerabilities, hallucinated APIs, insecure dependencies, policy violations, and implementation flaws that are difficult to identify during ordinary development workflows. Existing mitigation strategies based solely on prompt engineering are insufficient because general-purpose language models typically lack access to organization-specific security knowledge and governance requirements.

This paper proposed RAG-SecCode, a retrieval-augmented secure coding framework designed specifically for enterprise software development environments. The framework integrates secure knowledge retrieval, context assembly, LLM generation, automated validation, dependency governance, policy compliance assessment, and human-in-the-loop review. By grounding generation in authoritative security guidance and organizational standards, the framework seeks to reduce vulnerabilities before they enter the software lifecycle.

The study demonstrates that secure code generation should be viewed as a governed software engineering workflow rather than an isolated model interaction. Retrieval augmentation provides a mechanism for connecting LLM capabilities with enterprise knowledge assets, while validation and review layers provide accountability and risk control. Expected outcomes include reductions in vulnerability frequency, hallucinated API usage, dependency risks, and policy violations, accompanied by improvements in reviewer confidence and approval rates.

Future research should empirically evaluate RAG-SecCode using controlled enterprise development environments, investigate adaptive retrieval strategies, explore automated policy learning, and examine long-term security outcomes across diverse programming languages and architectures. As AI-assisted software engineering continues to evolve, retrieval-augmented secure coding frameworks may become foundational components of enterprise secure development ecosystems.

References

1. Aggarwal and P. Jalote, “Integrating static and dynamic analysis for detecting vulnerabilities,” in Proc. 30th Annu. Int. Comput. Softw. Appl. Conf. (COMPSAC), Sep. 2006, pp. 343–350.

2. V. Akuthota, R. Kasula, S. T. Sumona, M. Mitul, M. T. Reza, and M. D. Rahman, “Vulnerability detection and monitoring using llm, "Vulnerability detection and monitoring using LLM,” in Proc. IEEE 9th Int. Women Eng. (WIE) Conf. Elect. Comput. Eng. (WIECON-ECE), Nov. 2023, pp. 309–314.

3. Arusoaie, S. Ciobâca, V. Craciun, D. Gavrilut, and D. Lucanu, “A comparison of open-source static analysis tools for vulnerability detection in C/C++ code,” in Proc. 19th Int. Symp. Symbolic Numeric Algorithms Scientific Comput. (SYNASC), Sep. 2017, pp. 161–168.

4. “Copyrights, professional perspective - IP issues with AI code generators. ” Bloomberg Law, 2023. [Online].
Available: https://www.bloomberglaw.com/external/document/X4H9CFB4000000/copyrights-

professional-perspective-ip-issues-with-ai-code-gener

5. M. Besta, “Graph of thoughts: Solving elaborate problems with large language models,” in Proc. AAAI Conf. Artif. Intell., Mar. 2024, vol. 38, no. 16, pp. 17682–17690.

6. G. Bhandari, A. Naseer, and L. Moonen, “CVEfixes: Automated collection of vulnerabilities and their fixes from open-source software,” in Proc. 17th Int. Conf. Predictive Models Data Analytics Softw. Eng., Aug. 2021, pp. 30–39.

7. T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, and A. Askell, “Language models are few-shot learners,” in Proc. Adv. Neural Inf. Process. Syst. (NeurIPS), vol. 33, 2020, pp. 1877–1901.

8. Q.-C. Bui, R. Scandariato, and N. E. D. Ferreyra, “Vul4J: A dataset of reproducible Java vulnerabilities geared towards the study of program repair techniques,” in Proc. IEEE/ACM 19th Int. Conf. Mining Softw. Repositories (MSR), May 2022, pp. 464–468.

9. S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, “Deep learning based vulnerability detection: Are we there yet?,” IEEE Trans. Softw. Eng., vol. 48, no. 9, pp. 3280–3296, Sep. 2022.

10. Y. Chen, Z. Ding, L. Alowain, X. Chen, and D. Wagner, “DiverseVul: A new vulnerable source code dataset for deep learning based vulnerability detection,” in Proc. 26th Int. Symp. Res. Attacks, Intrusions Defenses, Oct. 2023, pp. 654–668.

11. E. Collini, F. Indra Kurniadi, P. Nesi, and G. Pantaleo, “Context-aware retrieval augmented generation using similarity validation to handle context inconsistencies in large language models,” IEEE Access, vol. 13, pp. 170065–170080, 2025.

12. Cybernative/code_vulnerability_security_dpo– Datasets At Hugging Face, CyberNative AI LLC, Sacramento, CA, USA, 2024, Accessed: Oct. 12, 2025.

13. D. Dai, “Why can GPT learn in-context? Language models implicitly perform gradient descent as meta-optimizers,” 2023, arXiv:2212.10559.

14. G. Deng, “PentestGPt: An LLM-empowered automatic penetration testing tool,” Aug. 2023, arXiv:2308.06782 [cs].

15. Y. Deng, C. S. Xia, C. Yang, S. D. Zhang, S. Yang, and L. Zhang, “Large language models are edge-case fuzzers: Testing deep learning libraries via FuzzGPT,” Apr. 2023, arXiv:2304.02014 [cs].

16. X. Du, G. Zheng, K. Wang, Y. Zou, Y. Wang, W. Deng, J. Feng, M. Liu, B. Chen, X. Peng, T. Ma, and Y. Lou, “Vul-RAG: Enhancing LLM-based vulnerability detection via knowledge-level RAG,” 2024, arXiv:2406.11147.

17. Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, “CodeBERT: A pre-trained model for programming and natural languages,” 2020, arXiv:2002.08155.

18. M. Fu, C. Tantithamthavorn, T. Le, V. Nguyen, and D. Phung, “VulRepair: A T5-based automated software vulnerability repair,” in Proc. 30th ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE), New York, NY, USA : ACM, 2022, pp. 935–947, doi: 10.1145/3540250.3549098.

19. Z. Gao, H. Wang, Y. Zhou, W. Zhu, and C. Zhang, “How far have we gone in vulnerability detection using large language models,” 2023, arXiv:2311.12420.

20. “GitHub copilot your AI pair programmer. ” GitHub, 2024. [Online]. Available: https://github.com/features/copilot

21. K. Goseva-Popstojanova and A. Perhinschi, “On the capability of static code analysis to detect security vulnerabilities,” Inf. Softw. Technol., vol. 68, pp. 18–33, Dec. 2015.

22. D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. K. Li, F. Luo, Y. Xiong, and W. Liang, “Deepseek-coder: When the large language model meets programming—The rise of code intelligence,” 2024, arXiv:2401.14196.

23. Y. Guo, C. Patsakis, Q. Hu, Q. Tang, and F. Casino, "Outside the comfort zone: Analysing LLM capabilities in software vulnerability detection," in Proc. Comput. Secur. - ESORICS: 29th Eur. Symp. Res. Comput. Secur., Bydgoszcz, Poland. Berlin, Germany : Springer, Sep. 2024, pp. 271–289.

24. R. Heumüller, T. Langer, and F. Ortmeier, "Empirical analysis of openai embeddings for semantic code review comment similarity," in Proc. Euromicro Conf. Softw. Eng. Adv. Appl. Springer, Sep. 2025, pp. 37–45.

25. Y. Jiao, J. Han, and C. Huang, "DeepVulHunter: Enhancing the code vulnerability detection capability of LLMs through multi-round analysis," J. Intell. Inf. Syst., vol. 63, no. 6, pp. 2237–2264, Dec. 2025.

26. H. Joshi, J. C. Sanchez, S. Gulwani, V. Le, G. Verbruggen, and I. Radiček, "Repair is nearly generation: Multilingual program repair with LLMs," in Proc. AAAI Conf. Artif. Intell., Jun. 2023, vol. 37, no. 44, pp. 5131–5140.

27. S. Kaniewski, F. Schmidt, M. Enzweiler, M. Menth, and T. Heer, "A systematic literature review on detecting software vulnerabilities with large language models," 2025, arXiv:2507.22659.

28. Kazemian, P. Ramanan, and M. Yildirim, "Text embedding models can be great data engineers," 2025, arXiv:2505.14802.

29. Khare, S. Dutta, Z. Li, A. Solko-Breslin, R. Alur, and M. Naik, "Understanding the effectiveness of large language models in detecting security vulnerabilities," in Proc. IEEE Conf. Softw. Testing, Verification Validation (ICST), Mar. 2025, pp. 103–114.

30. L. Kumar, V. Singh, S. Patel, and P. Mishra, "Empowering sw security: Codebert and machine learning approaches to vulnerability detection," in Proc. 21st Int. Conf. Natural Lang. Process. (ICON), 2024, pp. 399–407.

31. T. H. M. Le, M. A. Babar, and T. H. Thai, "Software vulnerability prediction in low-resource languages: An empirical study of CodeBERT and ChatGPT," in Proc. 28th Int. Conf. Eval. Assessment Softw. Eng., Jun. 2024, pp. 679–685.

32. P. Lewis, "Retrieval-augmented generation for knowledge-intensive NLP tasks," in Proc. NeurIPS, pp. 9459–9474, 2020.

33. H. Li, H. Yu, Y. Zhai, and Z. Qian, "Enhancing static analysis for practical bug detection: An LLM-integrated approach," in Proc. ACM Program. Lang., Apr. 2024, vol. 8, no. OOPSLA1, pp. 474–499.

34. H. Li, Y. Hao, Y. Zhai, and Z. Qian, "Enhancing static analysis for practical bug detection: An LLM-integrated approach," in Proc. ACM Program. Lang. (OOPSLA), vol. 7, pp. 474–499, 2023.

35. Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "VulDeePecker: A deep learning-based system for vulnerability detection," 2018, arXiv:1801.01681.

36. S. Lipp, S. Banescu, and A. Pretschner, "An empirical study on the effectiveness of static c code analyzers for vulnerability detection," in Proc. 31st ACM SIGSOFT Int. Symp. Softw. Test. Anal. New York, NY, USA : Association for Computing Machinery, Jul. 2022, pp. 544–555.

37. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation," in Proc. Adv. Neural Inf. Process. Syst., Dec. 2023, vol. 36, pp. 21558–21572.

38. Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "RoBERTa: A robustly optimized BERT pretraining approach," 2019, arXiv:1907.11692.

39. Z. Liu, Y. Tang, X. Luo, Y. Zhou, and L. F. Zhang, "No need to lift a finger anymore? Assessing the quality of code generation by ChatGPT," IEEE Trans. Softw. Eng., vol. 50, no. 6, pp. 1548–1584, Jun. 2024.

40. R. A. Majizi, H. Shaker, B. Kumar, and Z. T. Sharef, "Vulnerability detection: Dynamic analysis of web applications and assessment of penetration testing tools," in AI and IoT: Driving Business Success and Sustainability in the Digital Age, vol. 2. Switzerland : Springer, 2025, pp. 801–811. [Online].

Available: https://link.springer.com/chapter/10.1007/978-3-031-88874-8_71

41. Medeiros, N. Neves, and M. Correia, "Detecting and removing web application vulnerabilities with static analysis and data mining," IEEE Trans. Rel., vol. 65, no. 1, pp. 54–69, Mar. 2016.

42. R. Meng, M. Mirchev, M. Böhme, and A. Roychoudhury, "Large language model guided protocol fuzzing," in Proc. Netw. Distrib. Syst. Secur. Symp., San Diego, CA, USA : Internet Society, pp. 1–15, 2024. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2024-556-paper.pdf

43. R. Mim, A. Satter, T. Ahammed, and K. Sakib, "Automated software vulnerability detection using CodeBERT and convolutional neural network," in Proc. 19th Int. Conf. Eval. Novel Approaches to Softw. Eng., 2024, pp. 156–167.

44. Moradi Dakhel, V. Majdinasab, A. Nikanjam, F. Khomh, M. C. Desmarais, and Z. M. J. Jiang, "GitHub copilot ai pair programmer: Asset or liability? " J. Syst. Softw., vol. 203, Sep. 2023, Art. no. 111734.

45. Ouyang, "Training language models to follow instructions with human feedback," 2022, arXiv:2203.02155.

46. H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? Assessing the security of GitHub copilot's code contributions," in Proc. IEEE Symp. Secur. Privacy (SP), May 2022, pp. 754–768.

47. H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Examining zero-shot vulnerability repair with large language models," in Proc. IEEE Symp. Secur. Privacy (SP), May 2023, pp. 2339–2356.

48. N. Perry, M. Srivastava, D. Kumar, and D. Boneh, "Do users write more insecure code with AI assistants?" in Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS), New York, NY, USA : ACM, Nov. 2023, pp. 2785–2799, doi: 10.1145/3576915.3623157.

49. G. Sandoval, H. Pearce, T. Nys, R. Karri, S. Garg, and B. Dolan-Gavitt, "Lost at C: A user study on the security implications of large language model code assistants," in Proc. 32nd USENIX Security Symp. (USENIX Security), 2023, pp. 2205–2222. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/sandoval

50. X. Shen, Z. Chen, M. Backes, and Y. Zhang, "In ChatGPT we trust? Measuring and characterizing the reliability of ChatGPT," Oct. 2023, arXiv230408979 [cs].

51. "Stack overflow - Where developers learn, share, & build careers." StackOverflow, 2024. [Online]. Available: https://stackoverflow.com/

52. Y. Sun, "GPTScan: Detecting logic vulnerabilities in smart contracts by combining GPT with program analysis," in Proc. IEEE/ACM 46th Int. Conf. Softw. Eng. (ICSE), New York, NY, USA : ACM, Apr. 2024, pp. 1–13. [Online]. Available: https://dl.acm.org/doi/10.1145/3597503.3639117

53. Y. Sun, "LLM4Vuln: A unified evaluation framework for decoupling and enhancing LLMS' vulnerability reasoning," Jan. 2024, arXiv:2401.16185 [cs].

54. H. Touvron, "Llama 2: Open foundation and fine-tuned chat models," 2023, arXiv:2307.09288.

55. Wei, "Chain-of-thought prompting elicits reasoning in large language models," in Proc. NeurIPS, pp. 24824–24837, 2022.

56. S. Xia and L. Zhang, "Keep the conversation going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT," Apr. 2023, arXiv:2304.00385 [cs].

57. Yetistiren, I. Ozsoy, and E. Tuzun, "Assessing the quality of GitHub copilot's code generation," in Proc. 18th Int. Conf. Predictive Models Data Anal. Softw. Eng. (PROMISE), New York, NY, USA : ACM, Nov. 2022, pp. 62–71. [Online]. Available: https://dl.acm.org/doi/10.1145/3558489.3559072

58. Zhang, "ACFIX: Guiding LLMs with mined common RBAC practices for context-aware repair of access control vulnerabilities in smart contracts," Mar. 2024, arXiv:2403.06838 [cs].

59. Zhong and Z. Wang, “Can ChatGPT replace StackOverflow? A study on robustness and reliability of large language model code generation,” Jan. 2024, arXiv:2308.10335 [cs].

60. ZDNeT, 2023. [Online]. Available: https://www.zdnet.com/article/microsoft-has-over-a-million-paying-github-copilot-users-ceo-nadella/