TECHNISCHE UNIVERSITÄT
CHEMNITZ

Privacy-Preserving Source Code Vulnerability Detection and Repair using Retrieval-Augmented LLMs for Visual Studio Code

Master Thesis

Submitted in Fulfilment of the
Requirements for the Academic Degree
M.Sc. Web Engineering

Dept. of Computer Science
Chair of Computer Engineering

Submitted by: Md Hafizur Rahman
Student ID: 806286
Date: 15.12.2025

Internal Supervisor: Abubaker Gaber M.Sc.
Internal Examiner: Dr.-Ing. Sebastian Heil

Abstract

Secure coding support within the integrated development environment is increasingly demanded, as developers prefer immediate and actionable feedback during implementation rather than deferred security checks in later pipeline stages. Traditional static application security testing tools remain effective for rule-based vulnerabilities but exhibit limitations in detecting weaknesses that require semantic reasoning, cross-file context, or domain knowledge. Large Language Models (LLMs) offer new potential for vulnerability detection and automated repair suggestions; however, their adoption is constrained by inconsistent detection quality, high false-positive rates, outdated security knowledge, and significant privacy concerns when proprietary source code is processed by cloud-based services.

This thesis investigates a privacy-preserving approach to source code vulnerability detection and repair by integrating locally deployed LLMs with Retrieval-Augmented Generation (RAG) in a Visual Studio Code extension. The proposed system operates entirely on-device, ensuring zero code exfiltration while dynamically incorporating up-to-date vulnerability knowledge from curated Common Weakness Enumeration (CWE), Common Vulnerabilities and Exposures (CVE), and OWASP sources. Two operational modes are designed: a low-latency inline mode using LLM-only inference for real-time feedback, and an audit mode that combines LLM reasoning with retrieval-based context enrichment for deeper analysis.

The system is evaluated on a project-authored curated JavaScript/TypeScript dataset aligned to CWE/OWASP concepts. Quantitative evaluation compares LLM-only and LLM+RAG configurations across precision, recall, F1-score, false-positive rate, JSON parse success, and latency. Results show a strong model-dependent trade-off: the smallest model provides very low latency but low recall, mid-sized models increase recall with high alert noise, and the best F1 in this run is obtained by `qwen3:8b` with RAG (42.70%) at multi-second latency. The run also shows that parse robustness can dominate practical quality for some model families. Overall, the results demonstrate that privacy-preserving, locally deployed LLM workflows can provide useful vulnerability analysis assistance, while model selection, parse robustness, and output constraints remain critical for practical reliability.

Keywords: source code security, vulnerability detection, secure code repair, large language models, retrieval-augmented generation, privacy-preserving systems, Visual Studio Code

Acknowledgment

I would like to express my sincere gratitude to all those who supported me throughout the completion of this Master's thesis.

First and foremost, I am deeply grateful to my internal supervisor, *Abubaker Gaber, M.Sc.*, for his continuous guidance, constructive feedback, and professional support during all phases of this work. His availability for discussion, clear recommendations, and critical insights were instrumental in shaping both the technical direction and academic quality of the thesis.

I would also like to thank the academic staff involved in the Master's program for providing a solid theoretical foundation and an intellectually stimulating learning environment that supported my interest in this research topic.

Furthermore, I am thankful to my colleagues and peers for their general encouragement and for fostering a motivating atmosphere throughout the study program.

Finally, I would like to express my heartfelt appreciation to my family for their constant support, understanding, and encouragement throughout my academic journey. Their support played a vital role in enabling me to complete this work.

This thesis would not have been possible without the support and contributions of the individuals mentioned above, to whom I express my sincere thanks.

Task Description

Traditional rule-based security analysis tools are effective for detecting predefined vulnerability patterns but cannot reason about cross-file or context-dependent weaknesses. Cloud-based language model assistants provide stronger semantic analysis yet compromise data confidentiality and reproducibility because proprietary source code must leave the local environment. This thesis aims to design and evaluate a fully local, privacy-preserving secure coding assistant for Visual Studio Code that integrates large language models (LLMs) with Retrieval-Augmented Generation (RAG). All processing occurs on the developer's machine under a strict zero-egress policy to ensure that source code and intermediate data remain confidential.

The proposed assistant combines two complementary modes: real-time inline diagnostics that deliver immediate vulnerability alerts, and asynchronous audit and question–answering modes that use offline retrieval of vulnerability information such as CWE, CVE, and OWASP entries. A modular architecture separates the LLM inference layer, the local retrieval pipeline, and the Visual Studio Code extension interface. The system enforces privacy through local embeddings, signed corpora, and network isolation, while reproducibility is achieved through deterministic decoding, version-pinned dependencies, and containerized execution. The threat model assumes local adversaries capable of attempting code exfiltration, retrieval poisoning, or prompt injection, mitigated through corpus signing, network isolation, and provenance verification.

Evaluation will be conducted on benchmark and real-world JavaScript/TypeScript code. The comparison will be conducted across benchmark datasets (Juliet and OWASP Benchmark, approximately 40–50 test cases mapped to CWE identifiers) and one actively maintained real-world JavaScript/TypeScript project with documented vulnerabilities and verified patches. Evaluation metrics include precision, recall, F1-score, false-positive rate, JSON parse success, latency, privacy validation, and reproducibility, all measured under deterministic local execution.

Contents

List of Figures

List of Tables

List of Abbreviations

AST	Abstract Syntax Tree
CVE	Common Vulnerabilities and Exposures
CWE	Common Weakness Enumeration
F1	F1 Score
FPR	False Positive Rate
HNSW	Hierarchical Navigable Small World
IDE	Integrated Development Environment
LLM	Large Language Model
LRU	Least Recently Used
OWASP	Open Worldwide Application Security Project
RAG	Retrieval-Augmented Generation
SAST	Static Application Security Testing
VS Code	Visual Studio Code

1. Introduction

Modern software systems form the foundation of critical infrastructure, enterprise platforms, and consumer-facing applications. As software continues to increase in size, complexity, and development velocity, vulnerabilities introduced during implementation remain one of the dominant attack vectors. Industry and standards-oriented guidance repeatedly highlights recurring weakness classes—such as injection, broken access control, and insecure design—as persistent sources of exploitable flaws [59, 39, 38]. These weaknesses persist despite widespread adoption of secure development practices and automated analysis tools [21, 33, 43].

Static Application Security Testing (SAST) tools are commonly employed to detect vulnerabilities early in the Software Development Life Cycle (SDLC). Rule-based analyzers and taint-analysis-driven systems provide deterministic results and can scale to very large codebases [4, 10, 31]. However, both empirical studies and practitioner experience highlight limitations, including false positives, difficulty capturing framework-specific semantics, and limited fix guidance [23, 11]. As a result, developers frequently face large volumes of security findings that are costly to triage and may not reflect exploitable vulnerabilities.

In contemporary development workflows, security findings are often surfaced under significant time pressure. When a vulnerability is flagged, developers must interpret diagnostic output, locate the root cause, and apply an appropriate fix while preserving functional correctness. This process is cognitively demanding and error-prone, particularly for developers without specialized security expertise. Empirical studies show that warnings can be ignored, misinterpreted, or deprioritized when they are noisy or poorly explained [23, 11].

Recent advances in Large Language Models (LLMs) have demonstrated strong capabilities in source code understanding, generation, and transformation [6, 9, 47]. LLMs can produce plausible code completions and refactorings, enabling new forms of developer assistance directly in the IDE. However, empirical evaluations show that LLM-generated code can be *functionally correct yet insecure*, and that models may hallucinate or omit security-critical constraints without explicit grounding [61, 22].

Traditional SAST tools and LLM-based approaches exhibit complementary strengths and weaknesses. Static analyzers offer deterministic behavior and clear specifications but can struggle with noisy rule sets and missing semantic context. In contrast, LLMs provide flexible reasoning and explanation generation but are probabilistic

and can hallucinate [10, 22]. This tension suggests that neither approach is sufficient in isolation for reliable vulnerability detection and repair.

Retrieval-Augmented Generation (RAG) has emerged as a promising paradigm to ground LLM outputs in external knowledge without retraining [29, 25, 19, 34]. By augmenting prompts with retrieved vulnerability descriptions and secure coding guidance, RAG can reduce hallucinations and improve explanation consistency. Nevertheless, many existing assistants rely on cloud-hosted models or external services, raising privacy and confidentiality concerns when proprietary code is transmitted off-device. Privacy risks are amplified by known model leakage and memorization phenomena [8, 15].

These concerns motivate the development of privacy-preserving, locally deployed LLM systems integrated directly into the Integrated Development Environment (IDE). Local deployment enables developers to benefit from advanced reasoning capabilities while preserving code confidentiality, reducing data leakage risks, and improving reproducibility. Despite increasing interest, the design and systematic evaluation of such systems—particularly those combining retrieval, static analysis, and LLM-based reasoning within the IDE—remain underexplored.

This thesis addresses this gap by investigating a privacy-preserving vulnerability detection and repair system based on retrieval-augmented local LLMs integrated into Visual Studio Code. The proposed approach combines (i) a locally hosted LLM, (ii) a curated vulnerability knowledge base, and (iii) IDE-level context extraction to support real-time detection and repair suggestions for JavaScript and TypeScript codebases. The system is evaluated using reproducible local ablation experiments and quantitative metrics, including precision, recall, F1-score, false-positive rate, JSON parse success, and latency.

1.1. Objectives and Research Questions

The overarching objective of this thesis is to design and evaluate an IDE-integrated secure coding assistant that improves developer feedback loops while preserving code confidentiality. Concretely, the thesis investigates the following research questions:

- **RQ1 (Feasibility):** To what extent can a locally deployed LLM integrated into VS Code provide useful vulnerability detection and repair suggestions without transmitting source code to external services?
- **RQ2 (Grounding):** How does retrieval-augmented prompting influence detection quality, explanation consistency, and hallucination behavior compared to an LLM-only configuration?

- **RQ3 (Practicality):** What performance mechanisms (scoping, caching, debouncing) are necessary to make LLM-based security feedback usable within interactive IDE workflows?

1.2. Scope and Assumptions

This thesis focuses on IDE-integrated security assistance for **JavaScript and TypeScript** codebases inside **Visual Studio Code**. The central privacy objective is **no source-code exfiltration**: analyzed code and IDE-derived context are sent only to a local LLM runtime. The system may optionally refresh its local security knowledge base from *public* vulnerability metadata sources (e.g., CVE/NVD descriptions); this does not transmit user code and can be disabled for fully offline operation.

Threat Model and Trust Assumptions

The primary threat addressed by this thesis is unintended disclosure of proprietary code through cloud-hosted analysis services. Local inference reduces this exposure surface, but does not remove all security risks.

Table 1.1.: Threat model summary for Code Guardian.

Threat class	Assumption / attack path	Design response in this thesis
Source-code exfiltration	External API calls could expose proprietary code or derived context.	Local inference only; code and prompts sent to localhost backend; no code upload in evaluated workflows.
Prompt injection	Attacker-controlled comments/strings try to override analysis instructions [60].	Strict JSON-output contract in diagnostic mode; defensive parsing; human review before any repair is applied.
Retrieval poisoning	Low-quality or malicious knowledge entries degrade grounding quality.	Use curated local knowledge sources and explicit retrieval scope; future work adds provenance scoring and stricter source controls.
Local host compromise	Malware or untrusted local users can access code, prompts, or caches.	Out of scope for this thesis; assumes trusted developer machine and OS-level hardening.

In scope are privacy-preserving inference boundaries and robustness of developer-facing outputs. Out of scope are full endpoint hardening, hardware-level attacks, and enterprise identity/network controls.

1.3. Contributions

This work makes the following contributions:

- **Code Guardian prototype:** a VS Code extension for local vulnerability analysis and developer-controlled repair suggestions for JavaScript/TypeScript.
- **Privacy-preserving architecture:** a design that keeps code and analysis artifacts on-device and uses optional local retrieval to ground model reasoning.
- **Evaluation harness:** a reproducible, repository-contained benchmark suite and script for comparing models and configurations using precision/recall/F1, structured-output robustness, and latency.
- **Implementation insights:** practical patterns for integrating local LLM inference into IDE feedback loops (debounced triggers, function-level scoping, and caching).

1.4. Thesis Structure

Chapter 2 derives requirements for privacy-preserving vulnerability detection and repair assistance. Chapter 3 presents the conceptual system design and its mapping to the requirements. Chapter 4 details the implementation of Code Guardian as a VS Code extension with local inference and optional retrieval grounding. Chapter 5 evaluates detection quality, structured output robustness, and responsiveness. Chapter 6 concludes the thesis and Chapter 7 outlines opportunities for further improvement.

2. Analysis

This chapter analyzes the problem space of automated vulnerability detection and repair within modern software development workflows. The objective is to identify the fundamental technical and practical requirements for integrating Large Language Models (LLMs) into secure coding assistance, to critically examine existing approaches, and to derive design implications for a privacy-preserving, retrieval-augmented framework integrated into an Integrated Development Environment (IDE).

2.1. Requirements

This section outlines the core requirements that a system must fulfill to support effective, privacy-preserving vulnerability detection and repair within modern software development workflows. The focus of this thesis is on assisting developers during implementation by identifying security-relevant weaknesses in source code and providing actionable repair suggestions directly within the Integrated Development Environment (IDE).

The primary goal of the system proposed in this thesis is to enable developers to detect and remediate vulnerabilities in JavaScript and TypeScript code using a locally deployed Large Language Model (LLM) augmented with retrieval-based security knowledge. Rather than relying on cloud-hosted services or post hoc pipeline analysis, the system operates entirely on the developer's machine and integrates seamlessly into Visual Studio Code. Source code, analysis results, and vulnerability knowledge remain local at all times, ensuring privacy, reproducibility, and suitability for regulated or security-sensitive environments.

In contrast to traditional Static Application Security Testing (SAST) tools, which rely on predefined rules and often provide limited remediation guidance, the proposed system leverages LLM-based semantic reasoning combined with Retrieval-Augmented Generation (RAG). This enables the system to reason about code context, explain detected issues, and suggest concrete repairs grounded in curated vulnerability knowledge. To ensure practical usefulness, the system must satisfy both functional and non-functional requirements related to accuracy, transparency, performance, and usability.

Based on the challenges identified in Chapter 1 and the analysis of existing approaches, six core requirements (R1–R6) are defined. These requirements establish a systematic basis for evaluating the proposed architecture and guide the design decisions discussed in subsequent chapters.

R1 – Detection Accuracy and Consistency: The system must reliably identify security-relevant vulnerabilities in source code with stable behavior across repeated analyses. Detection results should be consistent for identical inputs, independent of invocation timing or interaction mode. The system should minimize false positives while maintaining adequate recall, ensuring that developers can trust reported findings and are not overwhelmed by spurious warnings.

R2 – Context-Aware Vulnerability Reasoning: The system must analyze vulnerabilities in their surrounding code context rather than relying solely on syntactic patterns. This includes reasoning about data flow, API usage, and control structures at the function and file level. The system should correctly distinguish between vulnerable and benign code patterns that appear syntactically similar and must avoid flagging issues that are mitigated by contextual safeguards.

R3 – Explainability and Transparency: To foster developer trust and facilitate efficient remediation, the system must provide transparent explanations for detected vulnerabilities. Each finding should be accompanied by a clear description of the underlying weakness, references to relevant vulnerability classes (e.g., CWE), and an indication of which code fragments contributed to the detection. Explanations should be concise, technically precise, and suitable for developers without specialized security expertise.

R4 – Actionable Repair Suggestions: The system must generate concrete, security-aware repair suggestions that address the root cause of detected vulnerabilities while preserving functional correctness. Suggested fixes should be directly applicable to the affected code region and must avoid introducing new security issues or breaking existing behavior. Developers must retain full control over whether and how suggested changes are applied.

R5 – Privacy-Preserving Operation: All analysis, retrieval, and generation steps must be performed locally without transmitting source code or derived artifacts to external services. The system must operate with locally deployed models and locally stored vulnerability knowledge, ensuring that proprietary code remains confidential and that results are reproducible across environments.

R6 – Usability and Responsiveness: The system must integrate smoothly into the IDE and provide feedback within acceptable latency bounds. Inline detection should support near–real-time interaction to avoid disrupting the development flow, while more comprehensive audit operations may tolerate higher latency. Usability is evaluated with respect to end-to-end response time, clarity of presented information, and compatibility with typical developer hardware configurations.

These six requirements define the evaluation criteria for the proposed system. In the following chapters, each requirement is addressed through specific architectural choices and implementation strategies. The evaluation chapter assesses to what extent the system satisfies these requirements in practice, using quantitative metrics and reproducible benchmarks.

2.1.1. R1: Detection Accuracy and Consistency

Consistency refers to the system's ability to produce stable, repeatable, and uniformly structured vulnerability detection results when analyzing identical or semantically equivalent source code. In the context of security analysis, consistency means that the same vulnerability is detected, classified, explained, and localized in a comparable manner across repeated runs, invocation modes, and interaction contexts.

Unlike general-purpose code analysis, vulnerability detection demands a high degree of determinism. Security findings are often used to guide remediation decisions, trigger audits, or satisfy compliance requirements. Inconsistent detection outcomes—such as reporting a vulnerability in one analysis but not in another, or fluctuating between different vulnerability classes—undermine developer trust and reduce the practical usability of the system. More generally, LLM-based systems are known to exhibit non-deterministic behavior and hallucination under unconstrained generation, motivating strict prompting and grounding strategies for stable outputs [47, 22, 61].

An ideal solution should ensure that once a vulnerability is identified, it is reported in a consistent manner. This includes stable classification (e.g., mapping to the same CWE category), consistent localization of the affected code region, and uniform explanation structure. For example, an input validation flaw should not alternately be reported as a generic "security issue," an injection vulnerability, or a logic error across different executions if the underlying code has not changed.

Consistency operates across multiple dimensions of vulnerability reporting. At the level of detection outcome, the presence or absence of a vulnerability should be stable across repeated analyses. At the level of classification, detected issues should map consistently to the same vulnerability categories and severity levels. At the level of explanation, descriptions should follow standardized phrasing and structure, avoiding unnecessary variation in terminology or level of detail. At the level of localization, the same code regions should be highlighted as relevant to the vulnerability.

This requirement is particularly critical because modern development workflows increasingly rely on automated security feedback integrated into the IDE. If a vulnerability warning appears intermittently or changes classification without code modifications, developers may disregard the warning entirely. Similar concerns have been documented in studies of static analysis tools, where inconsistent or noisy warnings reduce adoption and remediation rates [23].

From a system design perspective, achieving consistency requires controlling sources of nondeterminism in LLM inference and grounding vulnerability reasoning in structured security knowledge. Retrieval-Augmented Generation contributes to this goal by anchoring model outputs to curated vulnerability descriptions and examples, thereby reducing reliance on purely generative reasoning and mitigating hallucination.

For evaluation, detection consistency is assessed using repeated analyses of identical code samples under fixed configurations. We measure agreement between runs using macro-averaged precision, recall, and F1-score for vulnerability presence and classification. In addition, label agreement metrics are used to quantify stability in vulnerability categorization across runs. A high degree of agreement indicates that the system produces stable and reliable security findings.

Consistency Level	**Interpretation (example thresholds)**
High	**High consistency.** Vulnerability presence and classification are stable across runs (macro F1 ≥ 0.80), with minimal variation in localization and explanation structure.
Medium	**Moderate consistency.** Minor variations in classification or explanation occur (macro F1 in $[0.65, 0.80)$), but core vulnerability detection remains stable.
Low	**Low consistency.** Frequent changes in vulnerability presence or classification (macro F1 in $[0.50, 0.65)$), indicating unstable detection behavior.
None	**No consistency.** Detection results vary substantially across runs (macro F1 < 0.50), undermining trust in the system.

Table 2.1.: Evaluation scale for R1: Detection Consistency (example thresholds).

In summary, R1 ensures that vulnerability detection results are stable, reproducible, and uniformly structured across repeated analyses. By enforcing consistency across detection outcomes, classification, explanation, and localization, the system provides developers with reliable security feedback that can be trusted and acted upon within real-world development workflows.

2.1.2. R2: Context-Aware Vulnerability Reasoning

Context-aware vulnerability reasoning refers to the system's ability to correctly identify, classify, and localize security vulnerabilities by analyzing source code within

its surrounding semantic and structural context. This requirement goes beyond surface-level pattern matching and instead relies on understanding data flow, control flow, API semantics, and usage constraints to determine whether a code fragment constitutes a genuine security risk.

In contrast to traditional static analyzers that operate primarily on syntactic rules or predefined patterns, effective vulnerability reasoning must consider how code behaves in context. Prior research has shown that many vulnerabilities only manifest under specific execution paths, input assumptions, or API usage scenarios, and cannot be reliably detected without contextual analysis [31, 10]. Similarly, LLM-based approaches that lack explicit grounding may misclassify benign code as vulnerable or overlook subtle security flaws when context is incomplete or fragmented [61, 22].

An ideal solution must detect vulnerabilities even when relevant information is distributed across multiple statements, functions, or files. The system should associate related code fragments into a coherent reasoning context while ignoring unrelated logic. For example, input validation performed in a helper function should be correctly recognized when assessing the safety of downstream API usage, and defensive checks should prevent false positives when they effectively mitigate a potential vulnerability.

This requirement is critical because real-world codebases are rarely self-contained or linear. Security-relevant information is often scattered across variable initializations, conditional branches, utility functions, and framework abstractions. Without robust contextual reasoning, a system may either miss vulnerabilities that emerge from interdependent logic or incorrectly flag code that is secure by design. Such errors reduce developer confidence and limit the system's usefulness in practice.

Effective context-aware reasoning operates along three complementary dimensions. First, the system must correctly integrate dispersed information. Security-relevant signals may appear in different parts of a file or across multiple files, and the system must combine these fragments into a unified vulnerability assessment. Second, the system must recognize mitigation logic. If appropriate safeguards—such as input sanitization, authentication checks, or bounds validation—are present, the system should account for them and avoid reporting false positives. Third, the system must avoid unsupported inference. When insufficient context is available to determine whether a vulnerability exists, the system should explicitly acknowledge uncertainty rather than hallucinating a definitive conclusion.

Retrieval-Augmented Generation supports this requirement by grounding vulnerability reasoning in structured security knowledge, such as Common Weakness Enumeration (CWE) descriptions, secure coding guidelines, and historical vulnerability examples. Retrieved context helps the model align observed code patterns with known vulnerability semantics, reducing reliance on implicit assumptions and improving reasoning reliability.

The advantages of strong context-aware vulnerability reasoning are multifold. It improves detection accuracy by reducing false positives and false negatives caused by superficial pattern matching. It enhances robustness to coding style variation by focusing on semantic behavior rather than syntactic form. It also improves developer trust, as reported vulnerabilities more closely align with actual security risks in the codebase.

For evaluation, context-aware reasoning is assessed using classification metrics that measure the correctness of vulnerability detection and categorization in context-rich scenarios. Precision, recall, and macro-averaged F1-score are computed over benchmarks containing both vulnerable and non-vulnerable code samples with similar surface patterns. In addition, localization accuracy is measured by evaluating whether the system correctly identifies the relevant code regions contributing to the vulnerability. These metrics collectively capture the system's ability to reason about vulnerabilities in context rather than in isolation.

Reasoning Level	**Interpretation (example criteria)**
High	**High context awareness.** The system correctly integrates dispersed context, recognizes mitigation logic, and avoids unsupported inference. Vulnerability classification and localization are accurate (macro F1 ≥ 0.80).
Medium	**Moderate context awareness.** The system integrates some contextual information but occasionally misses mitigations or misinterprets dependencies (macro F1 in $[0.65, 0.80)$).
Low	**Low context awareness.** The system relies primarily on surface patterns, leading to frequent false positives or missed vulnerabilities (macro F1 in $[0.50, 0.65)$).
None	**No context awareness.** The system fails to incorporate contextual information and produces unreliable vulnerability assessments (macro F1 < 0.50).

Table 2.2.: Evaluation scale for R2: Context-Aware Vulnerability Reasoning.

In summary, R2 ensures that vulnerability detection is grounded in semantic and structural understanding of source code rather than superficial pattern matching. By integrating dispersed context, accounting for mitigation logic, and avoiding unsupported assumptions, the system provides accurate and trustworthy vulnerability assessments that reflect real-world security risks in modern codebases.

2.1.3. R3: Explainability and Transparency

Explainability and transparency refer to the system's ability to make its vulnerability detection and repair reasoning understandable, inspectable, and verifiable by developers. In the context of security analysis, transparency means that the system does not merely report that a vulnerability exists, but clearly communicates *why* it was detected, *which code elements contributed to the decision*, and *what security principles are being violated*. This requirement is essential for fostering developer trust, enabling informed remediation decisions, and supporting auditability in security-sensitive environments.

Unlike traditional static analyzers, which often expose explicit rules or taint paths, LLM-based systems risk operating as opaque black boxes. Prior research has shown that when developers cannot understand the rationale behind automated security findings, they are more likely to ignore warnings or apply fixes incorrectly [23, 11]. This issue is amplified for LLM-based approaches, where probabilistic reasoning and generative explanations may obscure the causal relationship between code patterns and reported vulnerabilities.

An ideal solution should present vulnerability findings alongside structured explanations that link detected issues to concrete code regions and recognized vulnerability classes. For example, when reporting an injection vulnerability, the system should indicate the untrusted input source, the absence or insufficiency of validation or sanitization, and the sensitive sink where exploitation may occur. Explanations should be concise, technically precise, and aligned with established security taxonomies such as the Common Weakness Enumeration (CWE).

Explainability operates across several dimensions. At the level of localization, the system should highlight the specific lines or code fragments that contributed to the detection, enabling developers to quickly identify the relevant context. At the level of reasoning, the system should describe the logical chain that led from observed code patterns to the vulnerability conclusion, avoiding vague or purely descriptive statements. At the level of justification, the system should reference recognized vulnerability categories or security guidelines to ground its explanations in established knowledge rather than ad hoc model intuition.

This requirement is particularly important in real-world development workflows, where developers must often balance security concerns against functional requirements and delivery timelines. Transparent explanations allow developers to assess whether a reported issue is relevant in their specific context and to determine whether a suggested fix aligns with project constraints. In regulated domains, transparency further supports accountability by enabling security findings to be reviewed, documented, and justified during audits.

Retrieval-Augmented Generation plays a central role in supporting explainability. By grounding explanations in retrieved vulnerability descriptions, secure coding

guidelines, and historical examples, the system can produce explanations that are both informative and consistent. This reduces the risk of hallucinated or misleading justifications and improves alignment between detection outcomes and established security knowledge.

For evaluation, explainability and transparency are assessed through a combination of qualitative and quantitative criteria. We evaluate whether explanations correctly reference the underlying vulnerability type, accurately identify the contributing code regions, and maintain internal coherence between detection, explanation, and suggested repair. In addition, explanation completeness is assessed by verifying that all essential components of the vulnerability reasoning—such as source, sink, and missing mitigation—are explicitly addressed. While explainability is inherently qualitative, structured scoring rubrics enable reproducible assessment across benchmarks.

Transparency Level	**Interpretation (example criteria)**
High	**High transparency.** Explanations clearly identify the vulnerability type, affected code regions, and reasoning steps, and reference established security knowledge. Developers can easily verify and act on the findings.
Medium	**Moderate transparency.** Explanations identify the vulnerability and affected code but provide limited reasoning detail or incomplete justification. Additional developer interpretation is required.
Low	**Low transparency.** Explanations are vague, generic, or loosely connected to the reported vulnerability, making verification difficult.
None	**No transparency.** The system reports vulnerabilities without meaningful explanation or justification, effectively operating as a black box.

Table 2.3.: Evaluation scale for R3: Explainability and Transparency.

In summary, R3 ensures that vulnerability detection and repair suggestions are accompanied by clear, structured, and verifiable explanations. By making the system's reasoning transparent and grounding it in established security knowledge, the system supports developer trust, facilitates correct remediation, and enables accountable use in real-world software development settings.

2.1.4. R4: Actionable Repair Suggestions

Actionable repair suggestions refer to the system's ability to generate concrete, security-aware code modifications that effectively remediate detected vulnerabilities while preserving the original program's functional correctness. In contrast to generic advice or high-level recommendations, actionable repairs must be directly applicable to the affected code region and sufficiently specific to support immediate developer adoption.

In the context of vulnerability detection, identifying a security flaw is only the first step. Developers ultimately require guidance on how to fix the issue correctly and efficiently. Empirical evidence suggests that warning overload and unclear remediation guidance reduce adoption and follow-through, especially when developers must interpret findings and design fixes under time pressure [23, 11]. Automated program repair research also highlights that patch quality and evaluation are non-trivial: fixes must address the root cause without introducing regressions or unintended behavior changes [71, 26, 40]. Inconsistent or incorrect fixes may leave vulnerabilities partially unresolved or introduce new flaws, undermining the value of automated detection.

An ideal solution should provide repair suggestions that are tightly coupled to the detected vulnerability and localized to the relevant code region. For example, if an injection vulnerability is identified, the system should suggest concrete input validation or parameterization mechanisms that are appropriate for the specific API and execution context, rather than issuing abstract recommendations such as "sanitize input." Suggested repairs should reflect established secure coding practices and align with recognized vulnerability classes, such as those defined by the Common Weakness Enumeration (CWE) and practitioner guidance such as OWASP cheat sheets [38, 55, 57, 49].

More broadly, actionable repair guidance should cover a range of common weakness classes beyond injection, including CSRF defenses, robust input validation, safe deserialization, secure file upload handling, password storage, and security logging practices [48, 51, 50, 58, 56, 52].

Actionable repair suggestions operate across several dimensions. At the level of specificity, suggested fixes must include precise code changes rather than vague guidance. At the level of correctness, repairs must eliminate the underlying vulnerability without breaking existing functionality or introducing new security issues. At the level of contextual appropriateness, fixes should respect the surrounding code structure, library usage, and project conventions, avoiding disruptive or unrealistic refactorings. Finally, at the level of control, developers must retain full authority over whether and how suggested repairs are applied.

This requirement is particularly important in IDE-integrated workflows, where developers expect rapid, low-friction assistance. Repair suggestions that are overly verbose, difficult to interpret, or incompatible with the existing codebase are likely

to be ignored. Conversely, concise and correct fixes that can be reviewed and applied incrementally encourage adoption and improve remediation rates.

Retrieval-Augmented Generation supports actionable repair suggestions by grounding generated fixes in curated vulnerability knowledge and historical remediation examples. Retrieved context enables the system to align suggested repairs with established security practices and reduce the risk of hallucinated or insecure fixes. By decoupling security knowledge from the model parameters, RAG further allows repair logic to evolve as new vulnerability patterns and recommended mitigations emerge.

For evaluation, repair quality is assessed using a combination of functional and security-oriented metrics. Functional correctness is evaluated by verifying that repaired code preserves expected behavior, for example through regression tests or benchmark-provided test cases. Security effectiveness is evaluated by re-analyzing the repaired code to confirm that the original vulnerability is no longer detected and that no new vulnerabilities are introduced. In addition, repair precision is assessed by measuring the proportion of suggested fixes that are both applicable and correct without manual modification.

Repair Quality Level	**Interpretation (example criteria)**
High	**High-quality repairs.** Suggested fixes are directly applicable, remove the vulnerability, preserve functional correctness, and align with secure coding practices.
Medium	**Moderate-quality repairs.** Suggested fixes address the vulnerability but require minor manual adjustment or introduce small, non-critical side effects.
Low	**Low-quality repairs.** Suggested fixes are vague, incomplete, or partially incorrect, requiring substantial developer intervention.
None	**No actionable repair.** The system fails to provide a usable fix or produces insecure or functionally incorrect code.

Table 2.4.: Evaluation scale for R4: Actionable Repair Suggestions.

In summary, R4 ensures that vulnerability detection is complemented by practical, security-aware repair suggestions that developers can readily apply within their workflow. By emphasizing specificity, correctness, contextual appropriateness, and developer control, this requirement bridges the gap between vulnerability identifica-

tion and effective remediation, thereby enhancing the overall utility of the proposed system.

2.1.5. R5: Privacy-Preserving Operation

Privacy-preserving operation refers to the system's ability to perform vulnerability detection, reasoning, and repair generation without exposing source code or derived artifacts to external services. This requirement ensures that all stages of analysis—including model inference, retrieval of security knowledge, and generation of explanations or fixes—are executed locally within the developer's environment.

Source code frequently contains proprietary logic, intellectual property, or sensitive business information. In many industrial, governmental, and regulated settings, transmitting such data to cloud-hosted services is unacceptable due to confidentiality, compliance, or contractual constraints. Consequently, any practical vulnerability detection system intended for real-world adoption must provide strong guarantees that code remains under the developer's control at all times.

An ideal solution should operate entirely on local hardware, using locally deployed LLMs and locally stored vulnerability knowledge bases. No source code, intermediate representations, embeddings, or analysis results should be transmitted beyond the local machine. This includes not only raw code but also prompts, retrieved documents, and generated outputs, all of which may inadvertently leak sensitive information if handled improperly.

Privacy-preserving operation encompasses several dimensions. At the level of deployment, the system must rely exclusively on local inference engines and avoid dependencies on external APIs or remote model hosting. At the level of data handling, all inputs and outputs must remain confined to local memory or storage, with no background telemetry or logging that could result in unintended data egress. At the level of reproducibility, local execution ensures that analysis results can be replicated across environments without reliance on changing external services or opaque model updates.

This requirement is particularly important for vulnerability detection, where analysis often requires access to complete source files or project-level context. Partial redaction or anonymization strategies are insufficient, as they may remove security-relevant information and degrade detection accuracy. By contrast, local execution allows full-context analysis while maintaining strict confidentiality.

Retrieval-Augmented Generation supports privacy-preserving operation by decoupling security knowledge from the model parameters and enabling the use of locally maintained knowledge bases. Vulnerability descriptions, secure coding guidelines, and historical examples can be curated and updated locally without requiring cloud-based retrieval or retraining. This design enables timely incorporation of new vulnerability knowledge while preserving data sovereignty.

For evaluation, privacy preservation is assessed through architectural inspection and runtime verification. We verify that no *external* network communication occurs during analysis by monitoring outbound connections and ensuring that all model inference and retrieval operations are confined to local processes (localhost communication is expected when using a local inference server). In addition, we assess whether the system functions correctly in offline environments, confirming that detection and repair capabilities remain available when network access is unavailable. These checks provide objective evidence that privacy guarantees are upheld in practice.

Privacy Level	**Interpretation (example criteria)**
High	**Strong privacy guarantees.** All analysis stages run locally, no outbound communication to external services is observed during analysis (localhost communication is expected), and the system functions offline using cached/baseline knowledge.
Medium	**Partial privacy.** Core analysis is local, but auxiliary components (e.g., optional updates or logging) require network access. No source code is transmitted.
Low	**Weak privacy.** Some analysis steps or prompts rely on external services, introducing potential data exposure risks.
None	**No privacy guarantees.** Source code or derived artifacts are transmitted to remote services during analysis.

Table 2.5.: Evaluation scale for R5: Privacy-Preserving Operation.

In summary, R5 ensures that vulnerability detection and repair can be performed without compromising source code confidentiality. By enforcing fully local execution, eliminating external dependencies, and enabling offline operation, the system addresses a key barrier to adoption of LLM-based security assistance in real-world, security-sensitive development environments.

2.1.6. R6: Usability and Responsiveness

Usability in the context of the proposed framework refers primarily to **latency**, defined as the end-to-end time delay between a developer action and the presentation of security feedback within the Integrated Development Environment (IDE). This includes the time required for context extraction, model inference, retrieval-augmented reasoning, and rendering of vulnerability findings or repair suggestions. While the system supports multiple interaction modes, the core task is the transformation

of *source code input* into *actionable security feedback*. Accordingly, the latency requirement applies uniformly across inline detection, on-demand analysis, and repair suggestion workflows.

Responsiveness directly determines whether security assistance can be integrated naturally into the development process. Human–computer interaction research consistently shows that feedback delivered within a few seconds preserves a sense of flow and supports effective turn-taking, whereas longer delays disrupt concentration and reduce tool adoption [7, 44]. In the context of IDE-based development, developers expect near-immediate feedback comparable to other static diagnostics such as type errors or linting warnings. Excessive latency risks relegating security analysis to a background task that is ignored or deferred.

Vulnerability annotations and repair suggestions should appear promptly after a triggering event, such as saving a file or explicitly invoking an analysis command. Timely feedback enables developers to assess security implications while the relevant code context is still active, reducing cognitive load and improving remediation efficiency.

Unlike other requirements—such as detection accuracy (R1), contextual reasoning (R2), explainability (R3), repair quality (R4), or privacy preservation (R5)—usability in this thesis is scoped strictly to latency. This focus reflects the practical reality that even accurate and well-explained security findings are unlikely to be acted upon if they arrive too late to fit within normal development workflows. This concern is particularly relevant for local LLM-based systems, where inference time can be substantial compared to traditional static analysis.

Latency, however, is not an absolute property of the system alone. It is strongly influenced by the **hardware and deployment environment** on which the system operates. Dedicated accelerators such as GPUs can significantly reduce inference time, whereas CPU-only or resource-constrained environments typically incur higher latency. Additionally, factors such as model size, retrieval depth, and concurrency affect responsiveness. For this reason, all latency measurements must be reported together with the corresponding hardware profile, including processor type, available memory, and accelerator configuration. This ensures that usability claims are interpreted relative to realistic deployment scenarios rather than as hardware-agnostic performance guarantees.

For evaluation, usability is measured using the 95th percentile end-to-end (p95 E2E) latency across representative interaction scenarios. The p95 metric captures worst-case responsiveness experienced by users while remaining robust to isolated outliers. Separate latency measurements are reported for interactive inline detection and for more comprehensive, explicitly triggered analyses.

In summary, usability in this framework is defined as responsiveness measured through end-to-end latency for IDE-integrated vulnerability detection and repair. Because latency is inherently dependent on hardware and deployment conditions,

Visual Score	Interpretation
	**High usability.** p95 end-to-end latency $\leq 2\,\mathrm{s}$ for inline detection on the declared hardware profile. Interaction remains fluid and non-disruptive.
	**Medium usability.** p95 end-to-end latency in $(2, 5)\,\mathrm{s}$. Delay is noticeable but acceptable for security feedback that is not continuously triggered.
	**No usability.** p95 end-to-end latency $> 5\,\mathrm{s}$ for inline scenarios or frequent timeouts. Feedback is too slow for practical integration into the development workflow.

Table 2.6.: Evaluation scale for R6: Usability (Latency).

all usability evaluations must be contextualized by reporting the corresponding execution environment. This ensures that results are comparable, interpretable, and grounded in realistic usage scenarios.

2.2. Related Work

This section reviews prior research relevant to LLM-based vulnerability detection and repair, with a focus on static analysis approaches, large language models for software security, retrieval-augmented generation, and privacy-preserving deployment. The discussion highlights both the strengths and limitations of existing work and positions the present thesis within the current research landscape.

2.2.1. Traditional Static Application Security Testing

Static Application Security Testing (SAST) tools have long been the primary means of detecting vulnerabilities during development. Rule-based analyzers and taint-analysis systems, such as Semgrep and CodeQL, identify predefined vulnerability patterns by statically inspecting source code [66, 18]. These tools provide deterministic and reproducible results, which makes them suitable for automated pipelines and compliance-driven environments.

From a research perspective, modern static analyzers build on foundational ideas such as abstract interpretation [12] and scalable bug-finding techniques [14]. In practice, large-scale deployments demonstrate that static analysis can find real defects in industrial codebases at massive scale [4]. Security-oriented static analysis has also been studied explicitly, including work that frames static analysis as an effective

security engineering control when combined with secure development processes [10, 21, 33].

Despite these strengths, SAST tools can struggle with contextual reasoning and generalization. They frequently produce false positives when security-relevant patterns appear in benign contexts (e.g., input validated elsewhere, framework-enforced invariants) and may miss vulnerabilities that depend on semantic relationships or framework-specific behavior. Moreover, many SAST tools offer limited remediation guidance, requiring developers to interpret findings and manually design fixes. Empirical studies show that warning overload and poor actionability reduce adoption and remediation rates [23, 11].

These limitations motivate research into approaches that complement deterministic static analysis with more flexible reasoning and explanation generation. In this thesis, Code Guardian aims to preserve the determinism and localization benefits of IDE diagnostics while using LLM-based reasoning (grounded by security knowledge) to improve explanation quality and provide developer-controlled repair suggestions.

2.2.2. LLM-Based Vulnerability Detection and Repair

Recent advances in Large Language Models (LLMs) have prompted extensive investigation into their use for code understanding, vulnerability detection, and repair suggestion. Contemporary models build on transformer architectures [68] and are often trained with large-scale pretraining objectives in the style of BERT/T5 [13, 62]. General-purpose LLMs and code-specialized models can generate syntactically plausible code and perform transformations such as refactoring and patch drafting [6, 9, 47, 16, 69, 45]. These capabilities make LLMs attractive for IDE-integrated security assistance, where developers benefit from explanations and candidate fixes at the point of code.

However, empirical evaluations show that model-generated code can be *functionally correct yet insecure*, and that probabilistic generation can introduce hallucinated assumptions or omit critical security checks [61, 22, 3]. This is particularly problematic for security, where small omissions (e.g., missing validation, unsafe sinks) can create exploitable weaknesses.

Before LLMs, machine-learning-based vulnerability detection explored learning semantic representations of code for classification. Early systems used deep learning over code fragments and patterns [30], while representation-learning work explored embeddings for code structure and semantics [2]. More recent approaches leveraged graph representations to capture richer program semantics [72, 1]. These methods improved over purely lexical baselines, but often required task-specific training data and did not naturally provide developer-facing explanations or repair suggestions.

Finally, security-related failure modes of LLMs extend beyond simple false positives/negatives. The broader LLM risk literature emphasizes both ethical/operational

risks [70] and adversarial behaviors such as jailbreaks and prompt-based attacks [73]. For IDE-integrated security tools, these risks motivate strict output contracts, careful prompt design, and conservative integration of model suggestions into developer workflows.

In response, contemporary approaches increasingly combine learned models with auxiliary signals such as retrieval, tools, or deterministic checks. Retrieval-augmented generation provides a mechanism to ground model reasoning in external security knowledge without retraining [29, 25, 19]. Tool-oriented prompting (e.g., using dedicated retrieval or analysis tools) further motivates modular designs where different components provide complementary evidence [65]. In Code Guardian, these ideas are reflected in the optional RAG module and the strict JSON-output contract that supports IDE diagnostics.

Repair suggestion quality is also connected to the broader literature on automated program repair. Classic systems such as GenProg and subsequent patch-learning approaches show both the promise of automation and the difficulty of evaluating patch correctness beyond passing tests [71, 28, 26, 40]. For IDE-integrated security assistance, these findings motivate presenting repairs as *optional* quick fixes, requiring explicit developer review and preserving responsibility for functional correctness.

2.2.3. Retrieval-Augmented Generation for Secure Coding

Retrieval-Augmented Generation (RAG) is a general paradigm for grounding language model outputs in external knowledge without retraining. In RAG, a retriever selects relevant documents for a given query and the generator conditions on those documents when producing an answer [29]. Retrieval can be implemented with lexical scoring functions such as BM25 [64] or with dense retrieval based on semantic embeddings. Dense retrieval approaches such as DPR and retrieval-augmented pretraining (REALM) motivate this design by showing that semantic retrieval can provide effective context for generation [25, 19]. Survey work further highlights RAG as a practical way to reduce hallucination and improve factuality [34, 22].

In secure coding settings, the retrieved context typically corresponds to vulnerability taxonomies (e.g., CWE), secure coding guidelines (e.g., OWASP cheat sheets), and historical vulnerability examples. This is a natural fit for vulnerability detection and repair because many issues are best explained by mapping code patterns to known weakness classes and well-established mitigations [38, 55, 59]. By grounding the model in such sources, the system can improve consistency (R1) and explanation quality (R3), while reducing unsupported guesses.

Implementation-wise, RAG depends on embedding models and efficient nearest-neighbor search. Sentence-level embedding methods such as Sentence-BERT are commonly used to map text into vector space [63]. Approximate nearest-neighbor indices such as HNSW provide high recall with practical latency [32], and libraries such

as FAISS popularized large-scale similarity search in practice [24]. In Code Guardian, these ideas are realized through local embeddings and a persistent HNSW vector store to keep retrieval on-device and compatible with IDE latency constraints.

2.2.4. Privacy-Preserving and IDE-Integrated Approaches

Privacy concerns pose a significant barrier to adopting LLM-based security tools in real-world settings. Cloud-hosted assistants require transmitting proprietary source code to external servers, which is unacceptable in many regulated or security-sensitive environments [15]. Beyond policy constraints, research has demonstrated concrete privacy risks in large language models, including memorization and extraction of training data [8] and inference attacks that raise questions about model confidentiality and data exposure [67]. For security tooling, these risks motivate keeping source code and analysis artifacts strictly local.

The IDE context introduces additional security considerations. Because the assistant processes attacker-controlled inputs (e.g., comments, strings, dependency code), prompt-injection and tool-manipulation attacks become relevant; OWASP explicitly catalogs such risks for LLM applications [60]. These concerns motivate designs that treat code as data, constrain outputs to machine-checkable schemas, and isolate retrieval sources to curated security knowledge.

IDE-integrated security tools can improve developer engagement and remediation rates by providing feedback during active development rather than post hoc analysis [23, 11]. However, many IDE assistants prioritize productivity features such as code completion and refactoring, with limited focus on security guarantees, reproducibility, or privacy boundaries. In practice, local deployment and deterministic interaction contracts become critical: users must understand what data is processed, where it is processed, and how results may vary across models and runs.

2.2.5. Positioning of This Thesis

In contrast to prior work, this thesis focuses explicitly on privacy-preserving vulnerability detection and repair using locally deployed LLMs integrated into Visual Studio Code. The proposed system combines retrieval-augmented reasoning with IDE-level context extraction to support both real-time and on-demand security analysis for JavaScript and TypeScript codebases. Unlike fine-tuned or cloud-dependent approaches, the system operates entirely offline, decouples security knowledge from model parameters, and is evaluated through reproducible local ablation experiments (LLM-only vs. LLM+RAG) with quantitative metrics.

By addressing detection consistency, contextual reasoning, explainability, actionable repair, privacy preservation, and usability within a unified framework, this work aims

to bridge the gap between academic advances in LLM-based security analysis and the practical requirements of real-world software development workflows.

3. Concept

This chapter presents the conceptual design of a privacy-preserving, retrieval-augmented vulnerability detection and repair system integrated into Visual Studio Code. The core objective is to assist developers in identifying and remediating security vulnerabilities in JavaScript and TypeScript code using locally deployed Large Language Models (LLMs), without transmitting source code to external services.

The design is motivated by three critical shortcomings observed in existing work. First, traditional Static Application Security Testing (SAST) tools rely on predefined rules and can struggle with contextual reasoning, producing false positives when security-relevant patterns appear in benign contexts [10, 4, 31]. Second, cloud-based LLM assistants require transmitting proprietary source code to external servers, which is unacceptable in regulated or security-sensitive environments [15, 60, 8]. Third, current approaches provide limited explainability and repair guidance, making it difficult for developers to understand and act upon detected vulnerabilities [23, 11].

The proposed system addresses these limitations through a modular architecture that enforces separation of concerns while preserving end-to-end coherence. The design directly supports the six requirements established in Section 2.1: detection consistency is enforced through retrieval-augmented grounding (R1), contextual reasoning is enabled through code analysis and data flow understanding (R2), intermediate artifacts and explanations provide transparency (R3), concrete repair suggestions are generated based on security knowledge (R4), all processing occurs locally without external data transmission (R5), and latency is optimized for IDE-integrated workflows (R6).

The remainder of this chapter is organized as follows. Section 3.1 derives the conceptual design from the analysis results, explaining how each requirement motivates specific architectural decisions. Section 3.2 describes the core components that comprise the system, detailing their responsibilities, inputs, outputs, and interactions. Section 3.3 presents detection and analysis workflows that illustrate how the system operates in different scenarios. Section 3.4 provides the high-level system architecture and design, including context diagrams, component views, and process flows. Detailed implementation and experimental validation are deferred to Chapters 4 and 5, respectively.

3.1. Concept Derivation from Analysis Results

The conceptual design of the proposed vulnerability detection system is derived systematically from the six requirements identified in Section 2.1 and the gaps observed in existing approaches reviewed in Section 2.2. This section traces how each architectural decision directly addresses specific limitations in current vulnerability detection systems, establishing the rationale for a local, RAG-augmented, IDE-integrated design.

3.1.1. From Cloud-Based to Privacy-Preserving Local Deployment

The analysis in Section 2.2 revealed that most LLM-based vulnerability detection systems rely on cloud-hosted models or external APIs, requiring transmission of source code to remote servers. This poses unacceptable privacy risks in industrial, governmental, and regulated settings where source code contains proprietary logic, intellectual property, or sensitive business information [15, 8, 67, 60].

These observations directly motivate the decision to deploy all components locally within the developer's environment. By running LLMs, retrieval systems, and analysis components entirely on local hardware, the system ensures that no source code, intermediate representations, or analysis results are transmitted beyond the developer's machine. This design directly satisfies R5 (privacy-preserving operation) and enables offline operation, making the system suitable for air-gapped or security-sensitive development environments.

In the Code Guardian prototype, the privacy boundary is defined around *source code*: analyzed code and IDE-derived context are sent only to a local LLM backend. The system may optionally refresh its *security knowledge base* from public vulnerability metadata sources (e.g., CVE/NVD descriptions and references). Such refresh operations do not transmit user source code and can be disabled to operate fully offline with cached or baseline knowledge.

The trade-off for local deployment is increased latency compared to cloud-based systems with dedicated accelerators. However, by carefully optimizing model selection, retrieval strategies, and context management, the system can achieve acceptable response times on standard developer hardware, addressing R6 (usability and responsiveness).

3.1.2. Retrieval-Augmented Generation for Security Knowledge Grounding

Requirement R1 demands consistent and accurate vulnerability detection across repeated analyses. Pure generative LLM approaches can suffer from hallucination and inconsistent classifications [22, 47]. Similarly, R2 requires context-aware reasoning that goes beyond surface-level pattern matching to understand data flow, control flow, and API semantics.

The system addresses these requirements through Retrieval-Augmented Generation (RAG), which grounds vulnerability detection in a locally maintained knowledge base of security information [29, 25]. This knowledge base includes:

- **CWE-aligned vulnerability patterns** and mitigation summaries [38]
- **OWASP guidance** for recurring web vulnerability categories [59, 55]
- **CVE/NVD summaries** (public descriptions and references) to keep the knowledge base current [37, 42]
- **Curated JavaScript/TypeScript security notes** (e.g., prototype pollution, unsafe deserialization patterns)

These sources are grounded in established taxonomies and practitioner guidance such as CWE and OWASP materials [38, 55, 59].

During analysis, relevant security knowledge is retrieved based on semantic similarity to the code under examination and provided as context to the LLM. This design reduces reliance on purely generative reasoning, improves detection consistency by anchoring outputs to established security knowledge, and enables the knowledge base to evolve independently of model parameters. By decoupling security knowledge from the model, the system supports continuous updates to vulnerability information without requiring model retraining.

3.1.3. IDE Integration for In-Context Security Assistance

Traditional SAST tools operate as separate processes in CI/CD pipelines, providing feedback only after code is committed. This delayed feedback loop increases cognitive load and reduces remediation rates, as developers must context-switch between writing code and reviewing security findings [23, 11].

The proposed system integrates directly into Visual Studio Code as an extension, providing security analysis during active development. This design supports two interaction modes:

- **Inline detection**: Real-time vulnerability annotations triggered by debounced document changes, providing immediate feedback similar to type errors or linting warnings
- **On-demand analysis**: Explicit analysis commands for comprehensive security audits of selected code regions or entire files

IDE integration directly addresses R3 (explainability and transparency) by enabling rich, interactive presentation of vulnerability findings, explanations, and repair suggestions within the developer's familiar environment. It also supports R4 (actionable repair suggestions) by allowing developers to preview, review, and apply suggested fixes with minimal friction.

3.1.4. Modular Component Architecture

The analysis in Section 2.2 showed that monolithic vulnerability detection systems lack transparency: when detection fails or produces unexpected results, it is difficult to diagnose whether the error originated in context extraction, retrieval, reasoning, or explanation generation.

The proposed system decomposes vulnerability detection into four core components, each with clearly defined responsibilities:

1. **Context Extraction**: Determines analysis scope (enclosing function, selection, file) and produces snippet text plus localization metadata
2. **Knowledge Retrieval**: Queries the local security knowledge base to retrieve relevant vulnerability information
3. **Vulnerability Detection**: Analyzes code context using the LLM, grounded in retrieved security knowledge
4. **Repair Generation**: Produces developer-controlled repair suggestions as optional patches (quick fixes)

This modular design enables component-level analysis, isolated improvement, and transparent error diagnosis. Each component can be tested, optimized, and replaced independently without destabilizing the overall architecture. Intermediate outputs (extracted context, retrieved knowledge, detection reasoning) remain visible for debugging and validation, supporting transparency and reproducibility.

3.1.5. Context-Aware Analysis with Code Understanding

Requirement R2 demands that the system correctly identify vulnerabilities based on semantic and structural context rather than syntactic patterns alone. Surface-level pattern matching produces false positives when secure code resembles vulnerable patterns, and false negatives when vulnerabilities depend on data flow or control flow relationships.

The system addresses this requirement primarily through *scope selection* and *prompt grounding*. In the current prototype, context extraction focuses on reliably selecting a coherent code region (e.g., the enclosing function) and mapping findings back to the editor. The LLM is then expected to reason about data flow and control flow *within the provided scope*, optionally grounded by retrieved security guidance.

Conceptually, context-aware reasoning benefits from multiple kinds of context, including:

- **Syntactic context**: function boundaries and code structure derived from AST parsing (used for scoping)
- **Local semantic context**: identifiers, API calls, and validation logic present in the analyzed region
- **Inferred data/control flow**: reasoning about sources, sinks, and guards within the snippet

This approach supports R2 for many localized vulnerability patterns, but it has inherent limitations when required evidence lies outside the analyzed scope (e.g., validation in a different file). Chapter 7 outlines extensions such as explicit taint-style traces and cross-file context extraction to address these cases.

3.1.6. Explainability Through Structured Reasoning

Requirement R3 demands transparent explanations that link detected vulnerabilities to concrete code regions and recognized security principles. Opaque "black box" detection undermines developer trust and makes it difficult to validate findings or apply fixes correctly [23, 11].

The system enforces explainability through structured reporting and IDE-native presentation. In the diagnostics pipeline, vulnerability reports include:

- **Code localization**: Specific lines or code fragments contributing to the vulnerability
- **Issue explanation**: A short message describing why the code is risky and what pattern is being flagged

- **Optional repair**: A suggested patch presented as a quick fix (developer-controlled)

In interactive webview modes, the system can provide longer, Markdown-formatted explanations, and (when enabled) can cite retrieved guidance to anchor the reasoning [38, 55]. The evaluation harness used in Chapter 5 additionally requests explicit `type` and `severity` fields to support quantitative scoring.

By grounding explanations in retrieved security knowledge (when used) and enforcing structured output formats where needed, the system produces actionable vulnerability reports that are auditable at the IDE level. This design directly supports R3 and improves developer trust by making detection outputs inspectable and reviewable.

This section has shown how each architectural decision maps to the requirements established in Chapter 2. The local, RAG-augmented, IDE-integrated design is not an arbitrary choice, but a direct consequence of the limitations observed in existing work and the operational constraints of real-world, privacy-sensitive development environments. The next section details the core components that instantiate this design, specifying their inputs, outputs, and processing logic.

3.2. System Components

The Code Guardian prototype is organized into a small set of components that map directly to the requirements from Chapter 2. The system is implemented as a VS Code extension that (i) extracts an appropriate analysis scope from the editor, (ii) invokes a local LLM for vulnerability analysis, (iii) optionally augments prompts with locally retrieved security knowledge (RAG), and (iv) renders findings and fix suggestions using IDE-native UI elements.

3.2.1. Context Extraction Component

The context extraction component determines *which* code is analyzed for a given interaction mode and provides the analyzer with enough metadata to localize findings in the editor.

Scopes. Code Guardian supports multiple scopes with different latency and completeness characteristics:

- **Function scope (real-time)** extracts the innermost function-like block at the cursor position (function declarations, arrow functions, methods, constructors). This is the default for continuous feedback because it keeps the prompt small.
- **File scope (on-demand)** analyzes the full current document and returns a comprehensive set of findings as diagnostics.

- **Selection scope (interactive)** sends a selected region (or current line) into a webview-based analysis view that supports follow-up questions.
- **Workspace scope (batch)** scans all JS/TS files and aggregates results into a security dashboard.

AST-based function extraction. Function scope extraction is implemented by parsing the current document with the TypeScript compiler API and selecting the smallest enclosing `FunctionLikeDeclaration` around the cursor. The extractor returns both the extracted snippet and its start-line offset (0-based) in the original document. This offset enables precise mapping of model-reported line numbers back to VS Code diagnostic ranges.

Design rationale. Function scoping is a practical mechanism to keep latency predictable for real-time use (R6), while the line-offset mapping improves explainability by ensuring that findings point to the correct source locations (R3). Deeper program context (imports, cross-file flows) is not extracted explicitly in the current prototype and is treated as a key future-work direction.

3.2.2. Knowledge Retrieval Component (RAG)

The retrieval component implements Retrieval-Augmented Generation (RAG) to ground LLM reasoning in local security knowledge [29, 25]. In Code Guardian, retrieval is implemented by a dedicated `RAGManager` that maintains a local knowledge base and a persistent vector index.

Knowledge base contents. The knowledge base is populated from a mix of curated and fetched *public* security metadata:

- **CWE-aligned entries** describing common weakness patterns and mitigations [38].
- **OWASP Top 10 guidance** for recurring web vulnerability categories [59].
- **CVE/NVD summaries** retrieved from the NVD API (descriptions and references) [42, 37].
- **JavaScript ecosystem advisories** for dependency and platform-specific risks.

Knowledge artifacts are cached on disk and reused across sessions. When network access is unavailable, the system falls back to a small baseline knowledge bundle so retrieval remains functional (with reduced coverage).

Indexing and retrieval. Knowledge entries are chunked using a recursive splitter (chunk size 1000, overlap 200) and embedded locally through an Ollama-served embedding model. Embeddings are stored in a local HNSW vector index [32] via LangChain tooling [27]. At query time, the top-k most similar chunks are retrieved

(default $k = 3$) and injected into the prompt as an explicit "relevant security knowledge" section.

Privacy boundary. Retrieval and embeddings are executed locally. Optional knowledge refresh operations fetch only public vulnerability metadata; user source code is not transmitted to those endpoints (R5).

3.2.3. Vulnerability Detection Component

The vulnerability detection component performs local LLM inference and returns findings in a format suitable for IDE integration.

Structured-output analyzer for diagnostics. For inline diagnostics, Code Guardian enforces a strict JSON-only output contract to reduce ambiguity and parsing failures. Each finding includes:

- **message**: a short description of the issue.
- **startLine/endLine**: 1-based line indices relative to the analyzed snippet.
- **suggestedFix** (optional): a replacement string that can be offered as a quick fix.

If no issues are found, the model must return an empty array `[]`. The analyzer performs defensive parsing by stripping code fences and extracting the first JSON array substring if the model emits additional text.

Failure handling and caching. Transient inference failures (timeouts, local server warm-up) are handled through retry with exponential backoff. To reduce repeated inference on unchanged code, results are cached with an LRU-style strategy keyed by a hash of (code, model), improving responsiveness during iterative edits.

Interactive analysis mode. In addition to the structured diagnostics flow, Code Guardian includes a webview-based analysis view for selected code. This mode uses Markdown-formatted responses and supports follow-up Q&A. When enabled, the RAG manager can be used to enrich the system prompt and user prompt for this interactive workflow.

3.2.4. Repair Generation Component

In the current prototype, repair generation is implemented as an *optional field* in the vulnerability report rather than as a separate autonomous patching system. When the analyzer includes a `suggestedFix`, the IDE integration layer exposes it as a quick fix action. Applying a fix is always user-initiated and can be reverted via the editor undo stack (R4). The system does not automatically validate functional correctness of repairs; this is treated as a major future improvement area.

3.2.5. IDE Integration Layer

The IDE integration layer connects the analysis pipeline to VS Code's APIs [36] and determines when analyses run and how results are presented.

Triggers. The extension supports both automatic and manual triggers:

- **Debounced real-time analysis** on document changes (default debounce: 800 ms) for JavaScript/TypeScript documents.
- **Manual file analysis** via a command palette command, with a size guard to avoid excessively large prompts.
- **Interactive selection analysis** and **contextual Q&A** views implemented as WebViews.
- **Workspace scanning** that aggregates results into a dashboard view.

Presentation. Findings are rendered as VS Code diagnostics (squiggles, Problems panel, hover tooltips). Suggested repairs are attached to diagnostics and exposed as a quick fix action.

3.2.6. Supporting Subsystems

Several additional subsystems are important for usability and reproducibility:

- **Model management** queries available local Ollama models, filters for suitable code models, and supports runtime model switching.
- **Analysis cache** is a bounded LRU cache (100 entries, 30-minute TTL) with user-visible hit/miss statistics.
- **Workspace dashboard** computes a coarse security score using issue density and keyword-based severity heuristics and visualizes the distribution across files.
- **Vulnerability data manager** caches public metadata (e.g., OWASP/CWE/CVE-derived entries) with a time-based expiry to support offline reuse after updates.

This section has described the core components that constitute Code Guardian. The next section describes how these components are orchestrated into workflows for real-time feedback, on-demand inspection, and batch scans.

3.3. Detection Workflows

While the component architecture defines what responsibilities exist in the system, IDE usability depends on how these components are orchestrated in concrete workflows. Code Guardian supports several workflows that trade off latency, context breadth, and output structure. This section describes the main workflows implemented in the prototype and relates them to requirements R1–R6.

3.3.1. Real-Time Function-Level Diagnostics

The real-time workflow provides continuous feedback while a developer edits JavaScript/TypeScript code. The key goal is to deliver timely, IDE-native warnings without disrupting flow (R6).

Trigger. The extension listens to document change events for JavaScript and TypeScript documents. Analysis is *debounced* (default: 800 ms) so the model is not invoked on every keystroke.

Scope and guards. When the debounce fires, Code Guardian extracts the innermost enclosing function at the cursor position and analyzes only that snippet. A size guard skips unusually large functions (default: 2000 characters) to bound worst-case latency and avoid overloading the local model.

Structured output for diagnostics. The analyzer is prompted to return a strict JSON array of issues. Each issue contains a message, a line range, and an optional fix string. The diagnostic adapter maps the snippet-relative, 1-based line numbers to VS Code ranges using the extractor's line offset and clamps ranges to valid document bounds. This enables stable rendering in the Problems panel, editor squiggles, and hover tooltips (R3).

Trade-offs. Function-level analysis improves responsiveness, but it can miss vulnerabilities whose evidence lies outside the current scope (e.g., validation in a different module). This limitation motivates the on-demand and workspace workflows and is addressed further in the future-work chapter.

3.3.2. On-Demand File Diagnostics

The file workflow provides broader coverage when a developer explicitly requests a deeper scan.

Trigger. A command palette action runs analysis over the full active document.

Scope guard. To avoid generating excessively large prompts, the prototype skips very large files (default: 20,000 characters) and warns the user instead.

Output. Findings are surfaced through the same structured diagnostics pipeline as the real-time workflow. This keeps the UI consistent, and it ensures that file scans can be reviewed in the Problems panel and navigated using standard IDE tooling.

3.3.3. Interactive Analysis and Follow-up Q&A

Some security questions require richer explanations than a single diagnostic message. Code Guardian therefore includes webview-based interaction modes.

Selection analysis view. The user can analyze a selected region (or the current line) and inspect the response in a dedicated analysis panel. This mode supports follow-up questions and streams Markdown-formatted responses for readability. Because it is user-initiated and not continuously triggered, it can tolerate higher latency than real-time diagnostics.

Contextual Q&A view. The contextual Q&A panel allows the user to select files and folders as context and ask security questions about that subset of the workspace. The extension reads the selected files locally and sends their contents only to the local LLM backend, preserving the no-exfiltration objective for source code.

RAG usage. When enabled, the RAG manager can enrich prompts for these interactive workflows by injecting retrieved security knowledge snippets (CWE/OWASP/CVE-derived guidance). Retrieval and embeddings are performed locally; optional knowledge refresh operations fetch only public metadata.

3.3.4. Workspace Scan and Security Dashboard

The workspace workflow supports periodic audits and prioritization across a repository.

File discovery and bounds. The scanner enumerates `*.js`, `*.jsx`, `*.ts`, and `*.tsx` files in the workspace while excluding dependency folders such as `node_modules`. Very large files are skipped (default: >500 KB) to keep runtime bounded.

Aggregation and scoring. Scan results are aggregated into a dashboard WebView. In addition to listing per-file issues, the dashboard computes a coarse security score based on issue density (issues per KLOC) and a keyword-based severity heuristic. This score is intended for prioritization rather than as a formal risk metric.

Developer interaction. The dashboard supports opening affected files directly from the report and rerunning scans. This workflow complements real-time diagnostics by helping developers understand which parts of the codebase concentrate the most findings.

3.3.5. Repair Suggestion Workflow

When the analyzer returns a `suggestedFix` for an issue, Code Guardian exposes it as a VS Code quick fix. Applying repairs is always user-initiated and integrates with the undo stack. This preserves developer control (R4) and reduces the risk of unintended behavioral changes from automatically applied patches.

3.3.6. Workflow Selection in Practice

In typical use, workflows complement each other:

1. Developers receive continuous feedback via debounced, function-level diagnostics while writing code.
2. Before committing or during review, developers run file scans for broader coverage.
3. For ambiguous findings or design-level questions, developers use the interactive analysis and contextual Q&A views to obtain richer explanations and mitigation guidance.
4. Periodically, workspace scans provide an aggregate view that supports prioritization and remediation planning.

Together, these workflows operationalize the core idea of Code Guardian: privacy-preserving local analysis combined with IDE-native feedback and developer-controlled remediation.

3.4. System Architecture and Design

This section presents the high-level architecture of Code Guardian using C4-style views (context, containers, and process). The goal is to make the privacy boundary and the main data flows explicit: code stays on-device, local inference is performed through a localhost LLM backend, and retrieval (when enabled) is backed by a local knowledge base and vector index.

3.4.1. Context View: System Boundary and External Interactions

Figure 3.1 positions Code Guardian within its operational environment. The primary actor is the developer working inside Visual Studio Code. The system executes in the VS Code extension host and communicates with a local LLM backend (Ollama) running on the same machine [46].

Local assets. The core local assets are:

- **Workspace source code** (JavaScript/TypeScript) being edited.
- **Local LLM runtime** (Ollama) used for analysis and (optionally) embeddings.
- **Local security knowledge base** and **vector index** used for retrieval when RAG is enabled [27, 32].
- **Local caches** for analysis results and vulnerability metadata.

Optional external data. Code Guardian may optionally fetch *public vulnerability metadata* to refresh the knowledge base (e.g., CVE records from the NVD API). This traffic does not include user source code and can be disabled by running in offline mode. After a refresh, cached knowledge can be reused without network access. This separation preserves the core privacy goal (no source-code exfiltration) while still allowing the knowledge base to evolve over time.

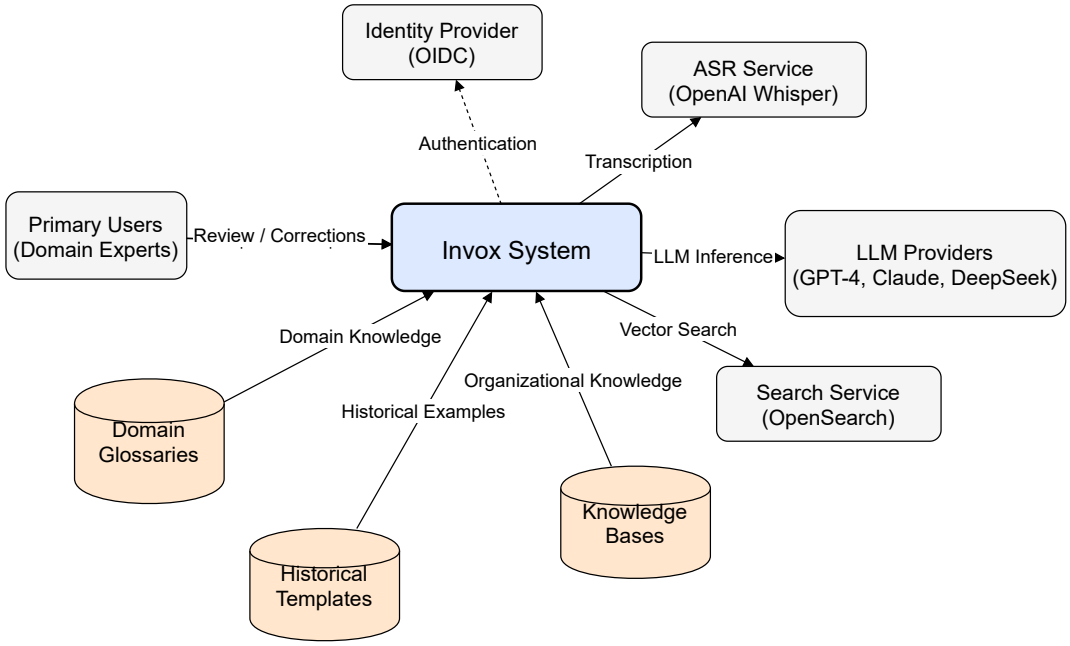

Figure 3.1.: Context diagram: Code Guardian runs locally in the VS Code extension host and calls a local LLM backend. A local knowledge base supports retrieval; optional refresh operations may fetch public vulnerability metadata, but source code remains local.

3.4.2. Container View: Internal Component Structure

Figure 3.2 summarizes the main internal containers and their responsibilities.

VS Code extension host. The extension is responsible for registering editor triggers, extracting analysis scopes, and rendering results using VS Code diagnostics and WebViews [36]. It provides commands for file analysis, selection analysis, contextual Q&A, model selection, cache inspection, and workspace scanning.

Structured diagnostics pipeline. For real-time and file-level diagnostics, the extension invokes a JSON-only analyzer that returns a list of issues with line ranges and optional fixes. These results are mapped into VS Code diagnostics and quick fixes. A bounded analysis cache reduces redundant LLM calls for unchanged snippets.

Interactive analysis pipeline. For selection analysis and contextual Q&A, the extension opens a WebView panel and streams Markdown-formatted responses from the local model. This mode supports conversational follow-up and can incorporate retrieved knowledge when RAG is enabled.

RAG manager and data manager. When enabled, the RAG manager maintains a local knowledge base (serialized entries) and a persistent vector store. A vulnerability data manager refreshes public metadata (CWE-/OWASP-aligned curated entries and optional CVE/advisory sources) and caches results on disk. The vector store is rebuilt or updated based on the knowledge base content.

3.4.3. Process View: End-to-End Workflows

Figure 3.3 illustrates the dynamic execution flow for the main workflows.

Real-time diagnostics (function scope).

1. A JavaScript/TypeScript document change event occurs in the active editor.
2. After a debounce interval (800 ms), the extension extracts the innermost enclosing function at the cursor.
3. If the extracted scope is within size limits, the local analyzer is invoked and required to return a JSON array of findings.
4. Findings are parsed defensively, cached, mapped to document ranges, and rendered as diagnostics. Optional `suggestedFix` strings are surfaced as quick fixes.

On-demand file diagnostics.

1. The developer invokes the “analyze full file” command.
2. The full document text is analyzed (subject to a size guard).
3. Findings are mapped to diagnostics and rendered in the editor.

Interactive analysis (selection) and contextual Q&A.

1. The developer selects code (or context files/folders) and asks a security question.

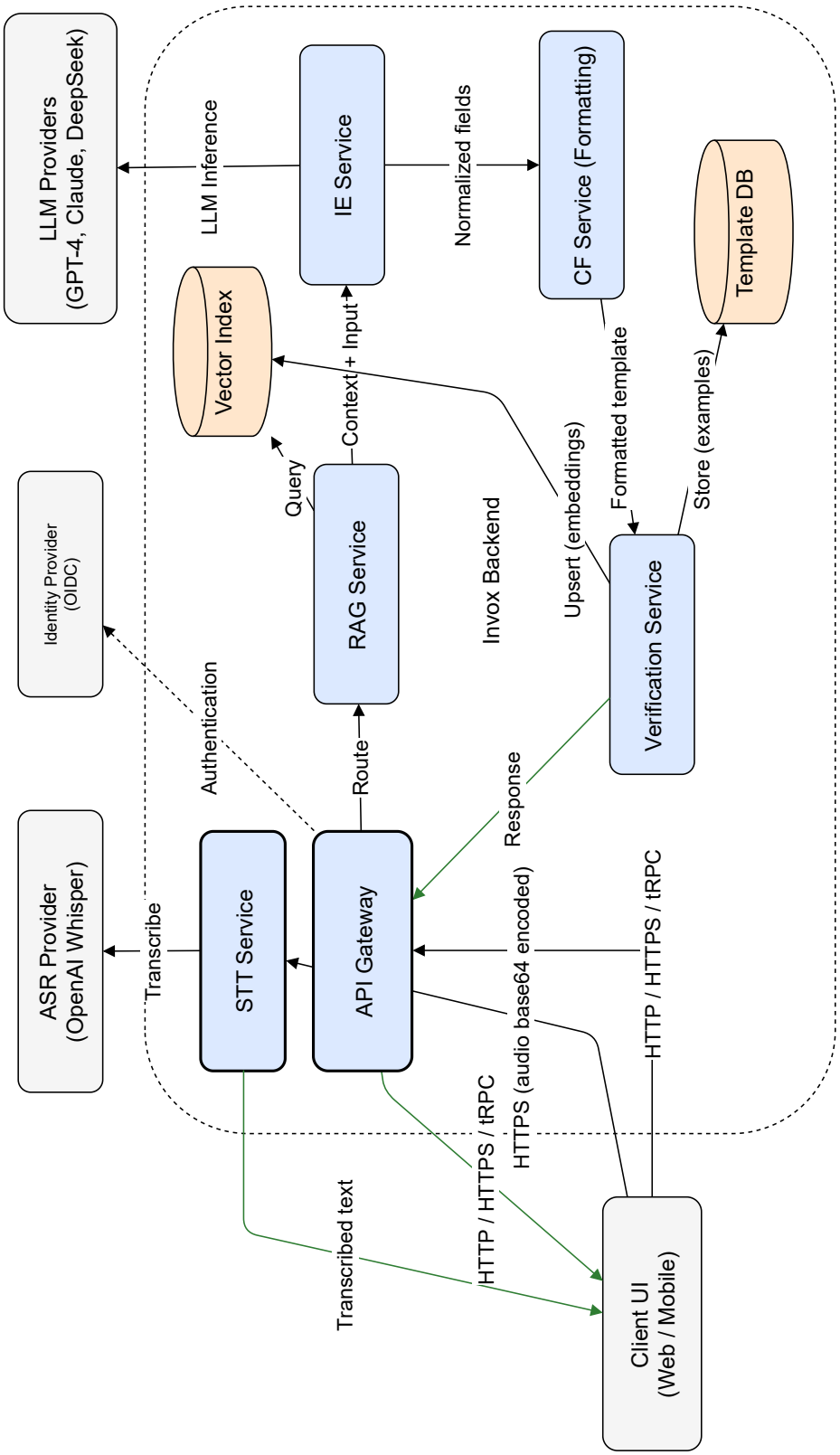

Figure 3.2.: Container diagram: The VS Code extension orchestrates context extraction, local LLM analysis, optional retrieval augmentation, and IDE-native rendering. Knowledge and caches are stored locally; optional knowledge refreshes fetch only public metadata.

2. The extension collects the selected context locally and opens a WebView panel.
3. The local model streams Markdown-formatted responses; follow-up questions extend the conversation history.
4. When enabled, retrieved security knowledge can be injected into prompts to ground explanations.

Workspace scan and dashboard.

1. The developer starts a workspace scan from the command palette.
2. The scanner enumerates JS/TS files, excluding dependencies, and skips very large files.
3. Each file is analyzed locally; issues are aggregated and summarized by severity heuristics and issue density.
4. Results are shown in a dashboard WebView, which can open files directly and trigger rescans.

Privacy considerations in the process view. Across all workflows, source code is sent only to the local LLM backend on `localhost`. Optional knowledge refresh operations fetch only public metadata and are cached; disabling refresh yields a fully offline analysis mode.

These architecture views make explicit how Code Guardian operationalizes the conceptual design: modular responsibilities, local inference as the default deployment, optional retrieval grounding, and IDE-native presentation with developer-controlled remediation.

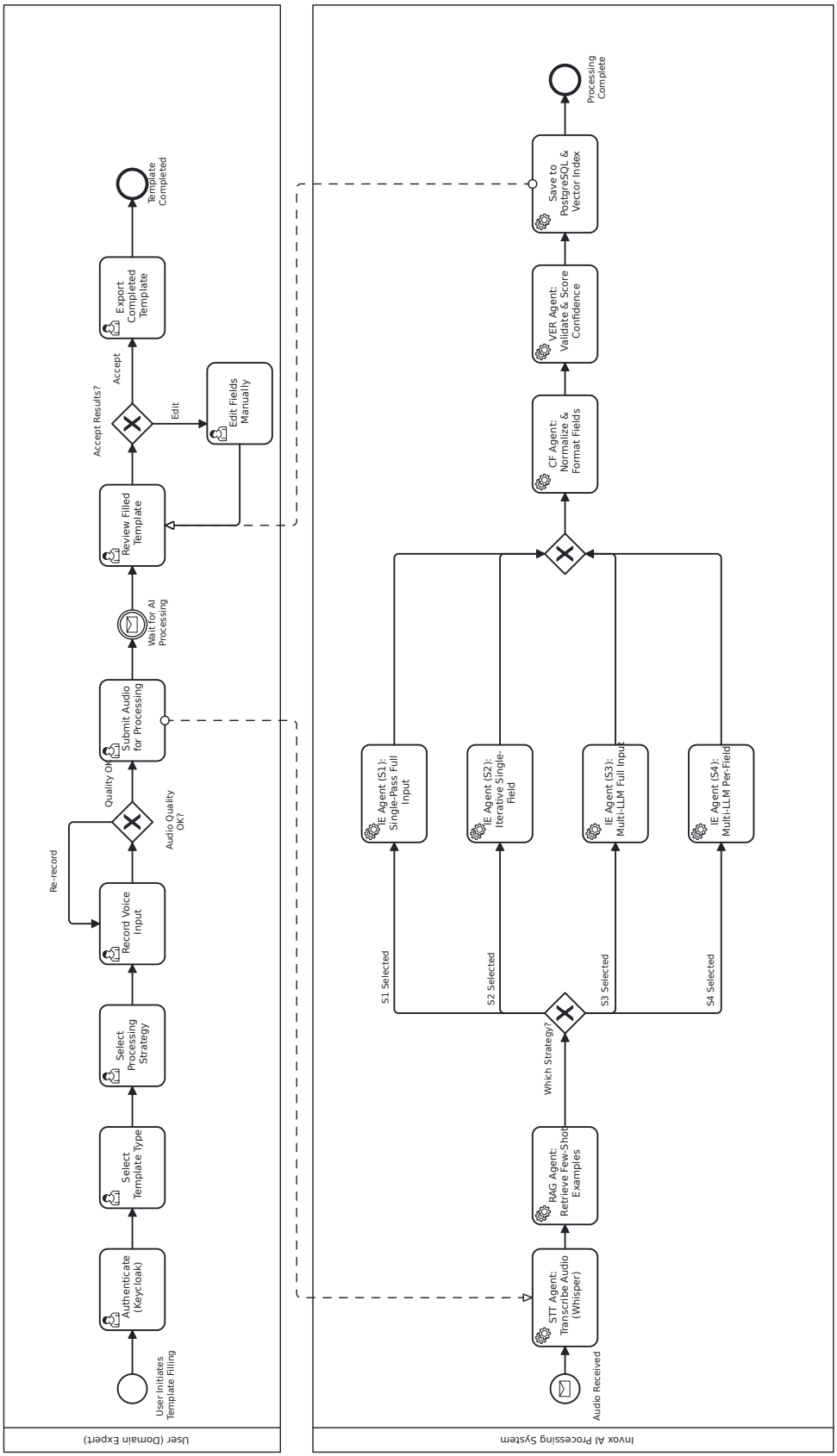

Figure 3.3.: Process view: debounced real-time function analysis, on-demand file analysis, interactive analysis/Q&A, and workspace scanning. The privacy boundary is maintained by keeping code on-device and using local inference.

4. Implementation

Building on the conceptual architecture introduced in the previous chapter, this chapter describes the implementation of *Code Guardian*: a privacy-preserving vulnerability detection and repair assistant integrated into Visual Studio Code. It details the technology stack, end-to-end workflow, core component implementation, orchestration modes (LLM-only vs. LLM+RAG), and the user interface exposed inside the IDE.

4.1. Technology Stack and Rationale

Code Guardian is implemented as a Visual Studio Code extension that performs security analysis and repair suggestion *locally* on the developer machine. The design goal is to integrate vulnerability detection into the IDE workflow while maintaining a strict *no source-code exfiltration* property (R5) and practical responsiveness for interactive use (R6).

VS Code Extension (TypeScript)

The extension is implemented in **TypeScript** using the VS Code Extension API [36]. Core responsibilities include registering editor events (on-change and on-command triggers), mapping analysis findings to VS Code diagnostics, and exposing quick fixes through code actions. The extension also provides two WebView-based interfaces: an interactive analysis view for selected code and a workspace security dashboard that aggregates findings across the project. Where IDE integration benefits from standardized editor tooling, the Language Server Protocol provides a relevant reference point for diagnostics-style interactions in modern IDEs [35].

Local LLM Inference via Ollama

All model inference is performed through **Ollama** running on `localhost:11434` [46]. Code Guardian supports multiple local code models (e.g., Qwen2.5-Coder, CodeLlama) and allows switching models at runtime through settings. Requests are configured for low-temperature decoding to improve determinism and schema

adherence, and retry logic with exponential backoff is applied to handle transient failures (e.g., model warm-up, timeouts).

Retrieval-Augmented Generation (RAG)

Optional RAG is implemented with **LangChain** and a local vector index [27]. Security knowledge entries (CWE patterns, OWASP Top 10 guidance, selected CVE summaries, and JavaScript/TypeScript secure coding notes) are embedded using a lightweight local embedding model (`nomic-embed-text` via Ollama) and stored in an **HNSW** vector store. At analysis time, relevant knowledge snippets are retrieved and injected into the prompt to ground the model's reasoning and reduce hallucinations (R1–R3).

Dynamic Vulnerability Knowledge Updates

The knowledge base is designed to be refreshable. Code Guardian supports updating OWASP and CVE information and persists cached data on disk. Importantly, only public vulnerability metadata is fetched; source code and extracted context remain local.

Performance Mechanisms

To support near-real-time feedback, the system uses (i) debounced analysis triggers during editing, (ii) a lightweight function-level analysis mode for continuous feedback, and (iii) an LRU-style analysis cache that avoids repeated model calls for unchanged snippets. These mechanisms directly support R6 while keeping the user experience consistent.

4.2. System Workflow

This section describes the operational workflow of the Code Guardian system, tracing how JavaScript/TypeScript code inside Visual Studio Code is transformed into vulnerability findings and repair suggestions. The workflow operationalizes the conceptual design from Chapter 3 and aligns the implementation with the requirements defined in Chapter 2, especially privacy-preserving operation (R5), responsiveness (R6), and explainability (R3).

End-to-End Processing Flow

Code Guardian exposes multiple analysis triggers (real-time and explicit commands). Two execution paths are particularly important in the prototype: (i) a *structured diagnostics* path that produces JSON findings suitable for editor diagnostics, and (ii) an *interactive analysis* path that produces Markdown explanations in a WebView and supports follow-up questions.

1. **Trigger and scope selection**: An analysis run is triggered either (i) automatically while the user edits code (debounced), or (ii) explicitly via a command (analyze selection, analyze file, scan workspace). The selected scope determines how much code context is included (function vs. selection vs. full document vs. workspace batch).
2. **Code context extraction**: The extension extracts the relevant code fragment and its location (start line offset). For real-time diagnostics, the default unit is the *enclosing function* at the cursor position. For selection analysis, the unit is the selected region (or the current line if no selection exists). For file analysis, the entire document text is used.
3. **Optional retrieval (RAG)**: If RAG is enabled and initialized, the system retrieves security knowledge snippets relevant to the analyzed code. Retrieval uses a local embedding model and a local vector index; retrieved guidance is appended to prompts to ground explanations and recommendations. In the current prototype, RAG is primarily used in the interactive WebView workflows and can be used for batch scanning; the real-time diagnostics path is kept lightweight to preserve responsiveness.
4. **LLM analysis (local)**: The local model (Ollama) is invoked with different output constraints depending on the workflow.
   - **Structured diagnostics**: the model is required to return *only* a JSON array of issues (message, line range, optional fix). Light post-processing removes Markdown fences and extracts the first JSON array substring if needed.
   - **Interactive analysis**: the model streams Markdown-formatted explanations to a WebView, enabling richer narratives and follow-up questions.
5. **Diagnostics and UI rendering**: Findings are converted into VS Code diagnostics and rendered inline (squiggles), in the Problems panel, and via hover tooltips. When a suggested fix is available, a quick-fix code action is offered so the developer can apply a patch after review.
6. **Caching and statistics**: Results are cached by (code snippet, model) to avoid repeated inference. The cache uses LRU-style eviction and a time-to-live policy,

and cache statistics can be inspected from the UI to validate responsiveness improvements under repeated edits.

Interactive vs. Batch Workflows

Real-time workflow (function-level). When editing, Code Guardian runs on a small scope and uses debouncing to avoid overwhelming the local model. A size threshold skips unusually large functions (2000 characters) to bound worst-case latency. This mode is optimized for R6 and is intended to surface common, localized patterns (e.g., injection sinks, insecure randomness) with minimal disruption.

On-demand workflow (selection/file). The developer can explicitly request analysis of selected code (interactive WebView) or the full file (structured diagnostics). Full-file analysis includes a conservative size guard (20,000 characters) to avoid excessively large prompts.

Workspace workflow (dashboard scan). The workspace scanner iterates through JavaScript/TypeScript files (excluding `node_modules`) and aggregates findings into a dashboard. Very large files are skipped (500 KB) to keep scans bounded. This supports risk overview and prioritization by surfacing the most affected files and a coarse severity distribution.

User-Controlled Remediation

Repair suggestions are never applied automatically. Instead, the system presents a quick-fix option that inserts the suggested patch only after explicit user confirmation. This design preserves developer control (R4), prevents silent behavior changes, and keeps responsibility for functional correctness with the human developer.

4.3. Core Component Implementation

This section details the concrete implementation of Code Guardian's main components. The prototype is implemented as a single VS Code extension located in `code-guardian/`. Its primary tasks are (i) extracting relevant code context inside the IDE, (ii) invoking local LLM inference, (iii) optionally grounding the prompt with retrieved security knowledge (RAG), and (iv) presenting findings and repairs in an IDE-native way.

4.3.1. Extension Entry Point and Event Wiring

The extension entry point registers event listeners and commands. Real-time analysis is triggered on document changes and is debounced (default: 800 ms) to avoid excessive inference calls during typing. On-demand commands support analyzing a selection, a full file, or scanning the workspace. Findings are reported through the VS Code diagnostics API so they appear inline and in the Problems panel.

4.3.2. Context Extraction and Scoping

Code Guardian extracts the smallest useful unit of code for interactive analysis: the enclosing function at the cursor position. This design reduces prompt size and improves responsiveness without requiring full-program analysis. Function extraction is implemented as a lightweight syntactic pass (`code-guardian/src/functionExtractor.ts`) and returns both the extracted snippet and its start-line offset within the original document. When a snippet is analyzed, the offset is later used to map model-reported line numbers back to the full file, so that diagnostics are placed correctly in the editor.

For explicit commands, the scope expands to (i) a selected region (or current line if no selection exists) and (ii) the full file. These scopes prioritize completeness and are typically used for deeper inspection or before committing changes.

4.3.3. LLM Analyzer (Local JSON-Only Output)

The analyzer performs local inference through Ollama and requires the model to return *only* a JSON array of findings. The output schema is intentionally minimal to keep parsing robust and IDE rendering straightforward.

Listing 4.1: Security issue output schema used by Code Guardian

```
interface SecurityIssue {
  message: string;
  startLine: number;    // 1-based
  endLine: number;      // 1-based
  suggestedFix?: string;
}
```

To increase reliability, the implementation includes:

- **Schema-constrained prompting:** the system prompt instructs strict JSON-only output.
- **Defensive parsing:** Markdown fences and stray quotes are removed, and the first JSON array substring is extracted when needed.

- **Retry logic:** transient errors (timeouts, temporary unavailability) are retried with exponential backoff.

Failure handling and safe defaults. In a developer tool, a false crash is often worse than a missed warning because it interrupts the workflow. Therefore, non-recoverable failures (e.g., repeated timeouts, model-not-found) are handled by showing a user-facing message and returning an empty issue list. This keeps the editor responsive while preserving explicit control over model configuration (e.g., prompting the user to pull a missing model).

4.3.4. Diagnostics Mapping and Localization

The diagnostic adapter converts the model's 1-based line numbering into VS Code's 0-based ranges and clamps indices to valid document bounds (`code-guardian/src/diagnostic.ts`). When a function snippet is analyzed, the previously computed start-line offset is added to each finding's range. If a suggested fix is included, it is attached to the diagnostic as related information and surfaced as a quick-fix action.

4.3.5. RAG Manager (Local Retrieval)

When enabled, the RAG manager maintains a local security knowledge base and a persistent vector index. Knowledge items are embedded using a local embedding model accessed through Ollama (`nomic-embed-text`) and stored in an HNSW vector store. For each analysis request, the manager retrieves the top-k relevant snippets and augments the prompt with (i) short vulnerability definitions and (ii) mitigation guidance. This grounding supports consistency and explainability (R1–R3).

Indexing and chunking. Knowledge entries are chunked using a recursive text splitter to balance semantic coherence and retrieval recall. The index is persisted under the extension's storage path so it can be reused across sessions without rebuilding. RAG initialization is performed lazily at runtime to avoid slowing down extension activation when retrieval is not used.

4.3.6. Vulnerability Knowledge Updates

To keep retrieved content current, the vulnerability data manager periodically refreshes public security sources (e.g., OWASP Top 10 entries, a curated set of CWE patterns, and a configurable number of recent CVEs). Retrieved metadata is cached on disk and only public vulnerability information is fetched. No user source code or extracted code context is transmitted off-device, preserving the privacy goal (R5).

Offline fallback. Because network access may be restricted in sensitive environments, the system includes a minimal baseline knowledge bundle that is used when updates fail. This ensures that RAG-enabled prompting remains functional offline, even though coverage is reduced relative to a refreshed knowledge base.

4.3.7. Diagnostics and Quick Fixes

Findings are mapped to VS Code diagnostics with appropriate text ranges. If the model provides a `suggestedFix`, a quick-fix code action is offered. Fixes are never applied automatically: the developer must confirm application, which maintains human control and reduces the risk of unintended behavioral changes (R4).

4.3.8. Caching and Responsiveness Mechanisms

To avoid repeated inference on unchanged snippets, Code Guardian caches analysis results keyed by the code snippet and active model (`code-guardian/src/analysisCache.ts`). In addition to caching, real-time analysis is debounced to reduce request volume during rapid edits. Together, these mechanisms reduce redundant local LLM calls and make continuous feedback feasible on developer hardware (R6), while also improving run-to-run consistency by limiting stochastic re-analysis.

4.3.9. Workspace Scanner and Dashboard

For project-level visibility, Code Guardian includes a workspace scanner that analyzes JavaScript and TypeScript files in batch and aggregates results into a dashboard WebView. The dashboard summarizes severity distribution and surfaces the most vulnerable files, supporting risk-based prioritization and iterative hardening.

The scanner enumerates files by extension, excludes dependency folders (e.g., `node_modules`), and skips very large files to bound runtime. Because the JSON-only analyzer does not mandate a severity field, the scanner applies a conservative keyword-based severity heuristic to support coarse prioritization in the dashboard; this is treated as a presentation aid rather than a ground-truth classifier.

4.3.10. Privacy Boundary and Threat Model Considerations

Code Guardian's privacy boundary is defined at the point of inference: analyzed code and any extracted context are sent only to the local Ollama server. The system does not transmit source code to external services. When knowledge updates are

enabled, only public vulnerability metadata is fetched; this data is cached locally and then used to ground prompts.

From a threat-model perspective, the main risks are *prompt injection* (attacker-controlled comments or strings that attempt to override the analysis instruction) and *retrieval poisoning* (malicious or misleading knowledge entries). The current prototype mitigates these risks primarily through strict output contracts and by keeping retrieval sources scoped to curated security data; Chapter 7 outlines stronger mitigations such as provenance tracking, allowlisting, and retrieval sanitization.

4.4. Analysis Modes and Orchestration

Code Guardian supports multiple analysis modes that trade off latency, context breadth, and user disruption. Rather than implementing multiple independent pipelines, the system follows a single orchestration pattern: extract code context → (optional) retrieve security knowledge → run local LLM analysis → render diagnostics and optional fixes.

4.4.1. Scopes: Function, Selection, File, Workspace

Function scope (real-time). During editing, Code Guardian analyzes the enclosing function at the cursor position. This reduces prompt size and supports low-latency feedback. A size threshold is applied to skip unusually large functions to avoid blocking the editor.

Selection scope. Developers can explicitly analyze selected code (or the current line if no selection exists). This mode is used for focused investigation and interactive follow-up.

File scope. Full-file analysis runs on demand and produces a complete set of findings for the document. This mode can tolerate higher latency but provides broader context for classification and repair suggestions.

Workspace scope. Workspace scanning analyzes multiple files and aggregates findings into the dashboard. The output is intended for prioritization (which files and categories dominate) rather than line-by-line interaction.

4.4.2. LLM-Only vs. RAG-Enhanced Execution

The system offers two operational configurations:

- **LLM-only:** prompt contains the analyzed code plus strict JSON-output constraints. This configuration minimizes overhead and is suitable for real-time use.
- **LLM+RAG:** the prompt is augmented with retrieved CWE/OWASP/CVE knowledge snippets and secure coding guidance. This improves grounding and can reduce false positives or unsupported suggestions, at the cost of additional embedding and retrieval time.

RAG can be toggled at runtime through settings and is surfaced via a status-bar indicator in the IDE.

4.4.3. Caching and Responsiveness

To reduce repeated inference calls, Code Guardian caches analysis results keyed by (code snippet, model). Under typical editing patterns, small changes frequently return to previously seen states (undo/redo, formatting), making caching effective. Debouncing and caching together aim to keep real-time diagnostics responsive (R6) while maintaining stable behavior across repeated runs (R1).

4.4.4. Repair Suggestion Handling

When the model returns a suggested fix, Code Guardian exposes it as a quick fix action. The extension does not auto-apply modifications; instead it requires explicit user confirmation and integrates with the editor undo stack. This design ensures that repair suggestions remain *advisory* and that final responsibility for functional correctness stays with the developer (R4).

4.5. User Interface

Code Guardian is designed to integrate security feedback into the existing Visual Studio Code workflow rather than introducing a separate application. The interface emphasizes (i) low-friction detection during coding, (ii) actionable remediation via quick fixes, and (iii) workspace-level prioritization.

4.5.1. Inline Diagnostics and Hover Explanations

Detected issues are rendered as standard VS Code diagnostics. This provides familiar affordances: squiggles in the editor, a centralized list in the Problems panel, and hover tooltips that summarize the finding. This presentation supports R3 by making explanations available at the point of code, without requiring context switching.

Real-time behavior. For JavaScript and TypeScript documents, diagnostics can be produced during normal editing via a debounced trigger. To preserve responsiveness, the default real-time scope is the enclosing function at the cursor, and unusually large functions are skipped. Developers can therefore treat Code Guardian feedback similarly to linting warnings: it appears close to the code being written and is automatically updated as the local scope changes.

4.5.2. Quick Fixes for Repair Suggestions

When a finding includes a suggested fix, Code Guardian offers a *Quick Fix* action. Fix application is always user-initiated and confirmed, which preserves developer control and reduces the risk of unintended behavior changes (R4). Fixes integrate with the editor undo stack so developers can revert and iterate.

Developer control. The prototype does not apply patches automatically. Instead, suggested fixes are attached to diagnostics and surfaced through the standard lightbulb UI. This design reflects the practical risk that an LLM-generated patch may be security-improving but behavior-changing; keeping the developer in the loop is therefore essential for safe adoption.

4.5.3. Interactive Analysis View and Contextual Q&A

For deeper inspection, the extension provides WebView-based views. A selection analysis view presents the model output in a dedicated panel for longer explanations and follow-up questions (e.g., “why is this vulnerable?”, “what is the safer alternative?”). In addition, Code Guardian provides a contextual Q&A view in which the developer can select files or folders as context and ask security-related questions about a codebase segment. Both views support switching the local model at runtime and (when enabled) benefit from retrieval-augmented grounding.

Model controls inside WebViews. The WebView views include a model selector that lists locally available Ollama models and supports refreshing the list at runtime. This enables quick comparison of smaller models (fast feedback) versus larger models (more detailed explanations) without restarting the extension.

4.5.4. Workspace Security Dashboard

The workspace dashboard aggregates findings across the codebase. It provides a severity breakdown and highlights the most vulnerable files, enabling developers to prioritize remediation work. This mode is complementary to inline diagnostics: it supports periodic security reviews and progress tracking after fixes.

Score and prioritization. To support triage, the dashboard computes a coarse security score based on issue density and severity heuristics, and surfaces the files with the highest concentration of findings. The score is intended as a prioritization aid rather than a formal risk metric; the underlying findings should still be inspected and validated by the developer.

4.5.5. Model and RAG Controls

Because Code Guardian is intended to run on developer hardware with varying resource constraints, model selection is exposed as a first-class UI feature. The extension provides commands to list available local Ollama models and switch the active model without restarting the extension. RAG can be toggled through settings and is also surfaced as a lightweight status indicator to make the current analysis mode explicit during development sessions. This transparency supports reproducibility (R1) and reduces user confusion about why results differ across runs.

RAG and cache management. The prototype also exposes maintenance commands for (i) updating vulnerability metadata used for the local knowledge base, (ii) rebuilding or searching the knowledge base, and (iii) inspecting and clearing the analysis cache. These controls help developers validate performance improvements (cache hit rates) and keep the retrieval corpus current without sacrificing source-code privacy.

5. Evaluation

This chapter evaluates Code Guardian with respect to the requirements defined in Chapter 2. The evaluation focuses on three aspects: (i) detection quality (precision/recall trade-offs and false positives), (ii) robustness of structured output (JSON parse success), and (iii) responsiveness (latency under IDE usage patterns). In addition, we report qualitative observations on repair suggestions and developer-facing usability.

5.1. Datasets

The evaluation dataset is a project-authored curated set of JavaScript and TypeScript security cases maintained alongside the Code Guardian prototype in `code-guardian/evaluation/datasets/`. Cases are aligned to CWE/OWASP concepts. Each test case consists of (i) a short code snippet, (ii) a list of expected vulnerabilities with CWE identifiers and severities, and (iii) an expected remediation description. Two datasets are provided:

- **Core dataset:** `vulnerability-test-cases.json` contains representative examples for common vulnerability classes (e.g., SQL injection, XSS, command injection, path traversal) as well as a small number of secure examples to measure false positives.
- **Advanced dataset:** `advanced-test-cases.json` contains additional vulnerability types and more nuanced patterns (e.g., insecure CORS configuration, timing attacks, mass assignment).

In the evaluated prototype version used for this thesis run, the scored dataset (`vulnerability-test-cases.json`) contains **33** test cases (**18** vulnerable and **15** secure). Expected findings are annotated with CWE identifiers to support aggregation by vulnerability class.

Which dataset is scored in this chapter. The repository-contained evaluation harness used in this thesis (`code-guardian/evaluation/evaluate-models.js`) was run in ablation mode with `-ablation -runs=3`. Unless stated otherwise, quantitative results in Chapter 5 refer to this 33-case dataset.

Vulnerability classes are aligned with the Common Weakness Enumeration taxonomy [38], and severities are recorded as coarse labels to support prioritization and reporting. Where applicable, severity can be related back to standardized scoring systems such as CVSS and public vulnerability databases such as CVE/NVD [17, 37, 42].

This dataset targets requirement R2 (context-aware vulnerability reasoning) by including vulnerability instances that cannot be resolved purely by superficial keyword matching (e.g., sink usage that requires reasoning about tainted input). It also supports R1 by enabling reproducible comparisons across repeated runs and across different local models.

Dataset Schema

Each record follows a uniform schema:

- `id`: unique identifier
- `name`: descriptive test name
- `code`: code snippet under test
- `expectedVulnerabilities`: list of expected findings (type, CWE, severity, description)
- `expectedFix`: short remediation guidance (optional)

The dataset is intentionally small and human-auditable to facilitate iteration on prompts, retrieval, and output validation. Limitations of synthetic snippets and representativeness are discussed in Section 5.8.

Toward larger benchmarks. While this thesis focuses on a curated, auditable suite to support rapid iteration, standard benchmark suites such as the OWASP Benchmark and NIST Juliet can be used to broaden coverage and stress-test generalization in future work [54, 41]. In addition, secure coding checklists and verification frameworks such as OWASP ASVS can guide the selection of security requirements and test categories when scaling the evaluation [53].

5.2. Evaluation Metrics

We evaluate Code Guardian along complementary axes aligned with Chapter 2: detection quality (R1–R2), explainability/structure (R3), and responsiveness (R6).

Detection Quality

Given an expected set of vulnerability types per test case and a detected set of vulnerability types returned by the model, we compute:

- **Precision:** $\mathrm{TP}/(\mathrm{TP}+\mathrm{FP})$
- **Recall:** $\mathrm{TP}/(\mathrm{TP}+\mathrm{FN})$
- **F1 score:** harmonic mean of precision and recall
- **False positive rate (FPR):** $\mathrm{FP}/(\mathrm{FP}+\mathrm{TN})$ on secure examples
- **Accuracy:** $(\mathrm{TP}+\mathrm{TN})/N$

Matching is performed at the vulnerability-type level. To account for naming variation (e.g., "Cross-Site Scripting" vs. "XSS"), the evaluation script applies case-insensitive substring matching between expected and detected type strings.

Secure examples. The scored dataset used in this chapter includes **15** intentionally secure snippets. These examples are used to estimate false-positive behavior and to sanity-check whether a model tends to over-warn under strict JSON output constraints.

Structured Output Robustness

Because Code Guardian integrates findings into IDE diagnostics, structured output is essential. We therefore report:

- **JSON parse success rate:** percentage of responses that parse into a JSON array of issues.

How parse failures are treated. In the evaluation harness, responses that do not parse as a JSON array are counted as parse failures and are scored as producing an empty set of issues. For vulnerable test cases, this manifests as false negatives (missed findings); for secure test cases, it can appear as a true negative but still indicates that the system is not usable for IDE automation. Reporting parse success rate separately is therefore important to avoid over-interpreting accuracy numbers when a model frequently produces malformed output.

Responsiveness

We measure:

- **Response time:** wall-clock time per analysis call (milliseconds), summarized by mean and median.

Latency is interpreted in the context of IDE workflows: function-level analysis has tighter budgets than file-level or workspace scans.

Limits of the scoring setup. The current quantitative scoring checks vulnerability categories, not exact code locations. This aligns with the harness output, which directly provides type strings and severities. Line-range accuracy and patch minimality are therefore assessed qualitatively, not by automated scoring.

5.3. Experimental Setup

All experiments are executed locally using the Code Guardian evaluation harness in `code-guardian/evaluation/evaluate-models.js`. The harness iterates over the curated dataset described in Section 5.1 and invokes Ollama with a strict JSON-only system prompt. Model decoding is configured for low randomness (`temperature=0.1`) to improve determinism and parsing stability.

Response Schema for Scoring

For evaluation purposes, the harness requests a richer structured output than the minimal in-editor diagnostics flow. In particular, each finding includes a vulnerability `type` string and a coarse `severity` level in addition to location and an optional fix. This enables type-level scoring (precision/recall) and per-category analysis. In the extension UI, vulnerability categories may appear within the message text and are used for downstream aggregation (e.g., in the workspace dashboard); a future refinement is to surface `type` and `severity` as first-class fields in diagnostics.

Compared Configurations

Two configurations are evaluated quantitatively:

- **LLM-only:** JSON-only analyzer without retrieval context.
- **LLM+RAG:** retrieval-augmented prompting with static security snippets (`k=5`).

Where relevant, results can also be stratified by model size or model family to study the trade-off between accuracy and latency.

Models and Hardware

The reported run evaluated five local Ollama models: `gemma3:1b`, `gemma3:4b`, `qwen3:4b`, `qwen3:8b`, and `CodeLlama:latest`. The execution environment was macOS (Darwin 25.3.0, arm64) on Apple M4 Max (16 CPU cores, 64 GB RAM), Node.js v20.19.5, and Ollama 0.17.0.

Runtime Parameters

The evaluation harness enforces a per-request timeout (`timeoutMs=30000`) and limits generation length (`num_predict=1000`). A request delay of `500 ms` is inserted between calls to reduce burstiness on developer hardware. Runs were executed sequentially with no warm-up and no retries.

Reproducibility Snapshot

Table 5.1.: Run configuration used for reported results.

Item	Value
Dataset file	`vulnerability-test-cases.json` (33 cases: 18 vulnerable, 15 secure)
Runs per sample	3
Prompt modes	LLM-only, LLM+RAG
RAG settings	static security snippets, `k=5`
Generation settings	`temperature=0.1`, `num_predict=1000`
Request controls	`timeoutMs=30000`, `requestDelayMs=500`
Execution mode	sequential, no warm-up, no retries
Models requested	`gemma3:1b`, `gemma3:4b`, `qwen3:4b`, `qwen3:8b`, `CodeLlama:latest`
Model resolution	All models resolved to the same tags as requested in this run
Software	Ollama 0.17.0, Node.js v20.19.5
Hardware/OS	Apple M4 Max (16 CPU cores, 64 GB RAM), macOS Darwin 25.3.0 (arm64)

Artifact Provenance

To document run provenance, Table 5.2 records the metadata fingerprint for the reported run.

Table 5.2.: Artifact provenance manifest for this thesis run.

Field	Value
Report repository commit	not captured by the harness in this run
Run window (UTC)	2026-02-25 18:26:56 to 2026-02-25 20:00:54
Total runtime	5 638 211 ms
Execution policy	sequential order, no retries, no warm-up
Dataset path	`code-guardian/evaluation/datasets/vulnerability-test-cases.json`
Invocation command	`node evaluation/evaluate-models.js -ablation -runs=3 -rag-k=5 -temperature=0.1 -num-predict=1000 -timeout-ms=30000 -delay-ms=500`
Model fingerprint (`gemma3:1b`)	digest prefix `8648f39daa8f`
Model fingerprint (`gemma3:4b`)	digest prefix `a2af6cc3eb7f`
Model fingerprint (`qwen3:4b`)	digest prefix `359d7dd4bcda`
Model fingerprint (`qwen3:8b`)	digest prefix `500a1f067a9f`
Model fingerprint (`CodeLlama:latest`)	digest prefix `8fdf8f752f6e`

For archival reproducibility, the recommended artifact bundle includes: (i) raw JSON output from the harness, (ii) the exact run configuration object, (iii) model digests, and (iv) generated summary tables. These artifacts should be stored as supplementary material together with the thesis PDF.

Run-to-Run Variability (Descriptive)

Each sample is evaluated three times in each configuration. Reported values are descriptive percentages and counts over pooled evaluations (e.g., recall over vulnerable evaluations, parse success over total requests). Because repeated runs reuse the same snippets and paired per-sample mode outputs were not exported by the harness, mode-to-mode differences are interpreted descriptively for this run only.

Response parsing. The harness strips Markdown code fences if present and then attempts to parse the remaining content as JSON. Responses that fail to parse are counted toward the parse failure rate and are scored as producing no issues, which impacts recall on vulnerable samples.

Running the Evaluation

The evaluation can be reproduced by running:

Listing 5.1: Running the Code Guardian evaluation harness

```
cd code-guardian
node evaluation/evaluate-models.js --ablation --runs=3 --rag-k=5 --temp
  --num-predict=1000 --timeout-ms=30000 --delay-ms=500
```

The script produces per-model metrics and can be extended to emit JSON/Markdown reports under `code-guardian/evaluation/logs/` for inclusion in the thesis appendix. For strict comparability with the reported tables, the evaluated model list should match the run configuration (`gemma3:1b`, `gemma3:4b`, `qwen3:4b`, `qwen3:8b`, and `CodeLlama:latest`).

The reported results in this thesis were produced with `runsPerSample=3` for both prompt modes.

5.4. Results: LLM-Only Configuration

This section reports results for Code Guardian when operating without retrieval augmentation. The goal is to establish a baseline for detection quality and latency when the model receives only the analyzed code and strict JSON-output constraints.

Detection Quality

Table 5.3 summarizes precision, recall, and F1 score on the curated dataset. These values should be reported per model, as different local models can show distinct precision/recall trade-offs.

Table 5.3.: LLM-only evaluation metrics on the curated dataset.

Model	Precision (%)	Recall (%)	F1 (%)	FPR (%)	Parse rate (%)
`gemma3:1b`	100.00	5.56	10.53	0.00	100.00
`gemma3:4b`	23.96	42.59	30.67	100.00	96.97
`qwen3:4b`	0.00	0.00	0.00	0.00	1.01
`qwen3:8b`	38.30	33.33	35.64	39.73	77.78
`CodeLlama:latest`	40.00	3.70	6.78	6.25	5.05

Latency

For interactive usage, response time is critical. Table 5.4 reports median and mean latency per analysis call for each model under the evaluation harness.

Table 5.4.: LLM-only latency metrics.

Model	Median (ms)	Mean (ms)
`gemma3:1b`	178	207
`gemma3:4b`	1415	1466
`qwen3:4b`	9549	9612
`qwen3:8b`	8774	9774
`CodeLlama:latest`	3987	4632

Discussion

The LLM-only results show heterogeneous behavior across model families. `gemma3:1b` remains very fast and conservative (high precision, low recall), while `qwen3:8b` provides the strongest LLM-only F1 but at much higher latency and lower parse reliability. `gemma3:4b` achieves higher recall than `gemma3:1b` but over-warns on secure samples (FPR 100%). `qwen3:4b` and `CodeLlama:latest` show severe parse instability in this run, which corresponds to near-zero detection performance.

5.5. Results: LLM+RAG Configuration

This section reports Code Guardian with retrieval augmentation enabled. In this configuration, the prompt is enriched with locally retrieved security knowledge (CWE/OWASP/CVE-derived summaries and mitigation guidance) to ground the model's output.

Detection Quality

Table 5.5 summarizes detection metrics for the RAG-enhanced configuration. Comparing Table 5.5 against Table 5.3 isolates the empirical impact of retrieval augmentation on precision/recall trade-offs.

Table 5.5.: LLM+RAG evaluation metrics on the curated dataset.

Model	Precision (%)	Recall (%)	F1 (%)	FPR (%)	Parse rate (%)
`gemma3:1b`	0.00	0.00	0.00	0.00	100.00
`gemma3:4b`	24.24	44.44	31.37	100.00	100.00
`qwen3:4b`	0.00	0.00	0.00	0.00	4.04
`qwen3:8b`	54.29	35.19	42.70	27.12	84.85
`CodeLlama:latest`	0.00	0.00	0.00	10.00	1.01

Latency Overhead

Retrieval introduces overhead from embedding, vector search, and prompt expansion. Table 5.6 reports the latency impact of enabling RAG under the same evaluation harness.

Table 5.6.: LLM+RAG latency metrics.

Model	Median (ms)	Mean (ms)
`gemma3:1b`	180	180
`gemma3:4b`	1335	1334
`qwen3:4b`	9852	9959
`qwen3:8b`	7809	9012
`CodeLlama:latest`	5478	5768

Discussion

RAG effects are model-dependent in this run. For `qwen3:8b`, RAG improved precision and F1, reduced FPR, and reduced median latency compared to LLM-only. `gemma3:4b` showed a small F1 gain but retained an FPR of 100%. For `gemma3:1b`, RAG reduced recall to zero. `qwen3:4b` and `CodeLlama:latest` remained dominated by parse failures in both prompt modes.

5.6. Model Comparison and Category Breakdown

Code Guardian supports multiple local models through Ollama. In practice, model choice affects not only detection quality but also structured-output robustness and latency. This section summarizes the measured ablation results and category-level behavior observed in the run logs.

Ablation Comparison

Table 5.7 compares all measured model/prompt-mode combinations.

Table 5.7.: Ablation summary across model and prompt mode.

Configuration	Precision (%)	Recall (%)	F1 (%)	FPR (%)	Parse (%)	Median (ms)
`qwen3:8b` (LLM+RAG)	54.29	35.19	42.70	27.12	84.85	7809
`qwen3:8b` (LLM-only)	38.30	33.33	35.64	39.73	77.78	8774
`gemma3:4b` (LLM+RAG)	24.24	44.44	31.37	100.00	100.00	1335
`gemma3:4b` (LLM-only)	23.96	42.59	30.67	100.00	96.97	1415
`gemma3:1b` (LLM-only)	100.00	5.56	10.53	0.00	100.00	178
`CodeLlama:latest` (LLM-only)	40.00	3.70	6.78	6.25	5.05	3987
`gemma3:1b` (LLM+RAG)	0.00	0.00	0.00	0.00	100.00	180
`qwen3:4b` (LLM-only)	0.00	0.00	0.00	0.00	1.01	9549
`qwen3:4b` (LLM+RAG)	0.00	0.00	0.00	0.00	4.04	9852
`CodeLlama:latest` (LLM+RAG)	0.00	0.00	0.00	10.00	1.01	5478

Per-Category Observations

Based on detailed run logs, injection-style cases (SQL injection and XSS) were consistently detected by `qwen3:8b` in the best-performing configuration. At the same time, representative misses remained (e.g., NoSQL injection), and secure samples still produced false alarms in several configurations. The most extreme alert-noise behavior appears in `gemma3:4b`, where secure evaluations were flagged in every run (FPR 100% in both modes).

Interpretation

These results motivate two practical follow-ups: stronger output-contract enforcement for models with low parse reliability, and stricter abstention behavior on secure samples for high-noise configurations. They also support differentiated defaults: a

fast conservative model for inline checks and a higher-accuracy model for audit mode when higher latency is acceptable.

5.7. Qualitative Case Studies and Repair Suggestions

Quantitative metrics summarize overall detection behavior, but IDE usefulness also depends on explanation clarity and repair quality. This section reports qualitative observations from representative cases in the recorded ablation run.

Representative Cases

Case A (True Positive with direct remediation): SQL injection. Sample ID: `sql-injection-1`. In the best-performing configuration (`qwen3:8b` with RAG), the model flagged direct string concatenation in the query and suggested parameterized queries. The suggested remediation aligned with the expected fix and was minimal.

Case B (False Positive on secure sample): safe process invocation. Sample ID: secure snippet with `execFile` array arguments. In `qwen3:8b` (LLM+RAG), this secure sample was still flagged, with a broad path-sanitization recommendation. This illustrates residual over-warning even in the strongest configuration.

Case C (False Negative): NoSQL injection. Sample ID: NoSQL injection representative case. In `qwen3:8b` (LLM+RAG), this vulnerable sample was missed in a representative run and no remediation suggestion was produced.

Case D (True Positive): cross-site scripting via `innerHTML`. Sample ID: XSS representative case. In `qwen3:8b` (LLM+RAG), the model correctly flagged unsafe HTML sink usage and proposed replacing `innerHTML` with safer rendering behavior.

Repair Suggestion Quality (R4)

Repair suggestions are evaluated qualitatively against three criteria:

- **Security adequacy:** does the fix mitigate the vulnerability class (e.g., parameterization for SQL injection)?
- **Minimality:** does it avoid unnecessary refactoring?
- **Applicability:** does it fit the code context and available APIs?

Because Code Guardian operates inside the IDE, repair suggestions are intentionally presented as *optional* quick fixes, enabling developer review and preventing silent modifications.

Observed Failure Modes

Observed failure modes in the run include:

- **Parse instability:** some model/mode combinations return malformed or non-JSON outputs at high rates.
- **Over-warning:** secure patterns flagged as vulnerable, especially with `gemma3:4b`.
- **Under-warning:** conservative configurations (notably `gemma3:1b` with RAG) missing most vulnerable samples.
- **High-latency audit modes:** strongest F1 configurations may still be too slow for inline workflows.

Error Taxonomy from Run Logs

Table 5.8.: Observed error taxonomy in the ablation run.

Error pattern	Representative evidence	Quantitative impact in this run
Parse-collapse regimes	`qwen3:4b` and `CodeLlama:latest` (both modes)	Parse rates dropped to 1.01–5.05%, and recall remained 0.00% in all but one configuration (`CodeLlama:latest` LLM-only: 3.70%).
Secure-sample over-warning	`gemma3:4b` (LLM-only and LLM+RAG)	FPR reached 100% in both modes (45/45 secure evaluations flagged).
Conservative under-detection	`gemma3:1b` (especially LLM+RAG)	Recall was 5.56% (3/54) in LLM-only and 0.00% (0/54) with RAG.
High-latency high-F1 mode	`qwen3:8b` (LLM+RAG)	Best F1 (42.70%) and lower FPR (27.12%) came with high latency (median 7809 ms; mean 9012 ms).

These observations motivate the future work directions summarized in Chapter 7.

5.8. Summary and Discussion

The evaluation framework provides reproducible measurements of detection quality, structured output robustness, and latency for Code Guardian. The curated dataset

enables rapid iteration on prompts, retrieval, and output validation, while the model-comparison view supports selecting an appropriate default model for real-time IDE use.

Key Takeaways

- **R1–R2 (quality and consistency):** Model behavior differs strongly. In this run, `qwen3:8b` (LLM+RAG) achieved the highest F1 (42.70%) with lower FPR (27.12%) than other high-recall configurations, while `gemma3:1b` remained highly conservative (precision 100.00%, recall 5.56%).
- **RAG impact is model-dependent:** RAG improved `qwen3:8b` (F1: 35.64% → 42.70%; median latency: 8774 → 7809 ms) and slightly improved `gemma3:4b` (F1: 30.67% → 31.37%), but reduced `gemma3:1b` recall to zero. `qwen3:4b` and `CodeLlama:latest` remained ineffective due to parse instability.
- **R3 (explainability):** IDE-native rendering (diagnostics, hovers) keeps explanations close to code, but message quality depends on prompt design and knowledge coverage.
- **R4 (repairs):** Quick fixes are effective as an assistive mechanism when suggestions are minimal and context-appropriate; developer confirmation remains essential.
- **R5 (privacy):** All source code analysis and model inference remain local; only public vulnerability metadata may be fetched for knowledge updates.
- **R6 (responsiveness):** Responsiveness depends heavily on model profile (about 178–180 ms median for `gemma3:1b`, about 1.3–1.4 s for `gemma3:4b`, and about 7.8–9.9 s for `qwen3` variants).

Claim-to-Evidence Map

Table 5.9 links the thesis research questions to the concrete measurements and sections used as evidence.

Table 5.9.: Claim-to-evidence map for core research questions.

RQ	Claim tested	Evidence used	Outcome in this thesis run
RQ1 (Feasibility)	Local IDE assistant can provide useful security findings without source-code exfiltration.	Privacy boundary and local deployment design (Chapter 3); detection/latency measurements (Sections 5.4–5.6).	Supported with caveats: useful findings are produced locally, but model choice strongly affects recall and false positives.
RQ2 (Grounding)	Retrieval augmentation improves quality and consistency versus LLM-only.	Ablation results across prompt modes and models (Section 5.6); descriptive mode-to-mode comparison in this section.	Partially supported and model-dependent: clear gains for `qwen3:8b`, marginal gain for `gemma3:4b`, degradation for `gemma3:1b`.
RQ3 (Practicality)	Real-time IDE use is achievable with scoping/caching/debouncing.	Runtime design and guardrails (Chapter 4); measured latency distributions (Sections 5.4, 5.5).	Supported for small models and constrained workflows; high-performing large models remain primarily audit-mode due to multi-second latency.

Run Variability (Descriptive)

Each configuration was evaluated three times per sample. Reported percentages are run-level descriptive values over pooled evaluations (e.g., recall over 54 vulnerable evaluations and parse rate over 99 total requests). Because repeated runs reuse the same snippets and paired per-sample mode outputs were not exported by the harness, this thesis reports directional mode-to-mode differences descriptively and does not make inferential claims.

Latency was measured across repeated calls per sample. For high-latency configurations (`qwen3:8b`, `qwen3:4b`, and `CodeLlama:latest`), mean latency remained above median in both modes, indicating right-skewed response-time tails under local inference.

Mode-to-Mode Recall Differences (Descriptive)

Table 5.10 summarizes recall changes between LLM-only and LLM+RAG for each model in percentage points.

Table 5.10.: Descriptive recall differences (LLM-only vs. LLM+RAG).

Model	Recall LLM (%)	Recall RAG (%)	Δ Recall (pp)
`gemma3:1b`	5.56 (3/54)	0.00 (0/54)	-5.56
`gemma3:4b`	42.59 (23/54)	44.44 (24/54)	+1.85
`qwen3:4b`	0.00 (0/54)	0.00 (0/54)	+0.00
`qwen3:8b`	33.33 (18/54)	35.19 (19/54)	+1.86
`CodeLlama:latest`	3.70 (2/54)	0.00 (0/54)	-3.70

These differences are reported as run-level directional deltas. Because repeated evaluations reuse the same snippets and paired per-sample contingency outputs were not exported by the harness, these values should be interpreted as descriptive measurements for this run.

Limitations

The dataset is intentionally small and synthetic; therefore, it may not capture the full diversity of real-world codebases (framework-specific patterns, multi-file flows, and complex validation logic). Future work should include evaluation on larger benchmarks and real repositories, as discussed in Chapter 7.

Negative Results and Boundary Conditions

To avoid overstating system performance, the following negative outcomes are explicit boundary conditions of this thesis run:

- **RAG is not uniformly beneficial:** for `gemma3:1b`, enabling RAG reduced recall from 5.56% to 0.00%.
- **Higher recall can come with high alert noise:** `gemma3:4b` flagged all secure evaluations in both modes (FPR 100%).
- **Parse robustness can dominate measured quality:** `qwen3:4b` and `CodeLlama:latest` showed parse rates between 1.01% and 5.05%, with near-zero recall.

- **Run-level deltas are descriptive only:** mode-to-mode changes should be interpreted as measurements from this run, not inferential effects.

These results indicate that practical deployment requires explicit tuning of model, prompting, and scoring ontology rather than assuming a single universally best configuration.

Threats to Validity

Table 5.11.: Threats-to-validity summary and mitigations.

Threat type	Main risk	Mitigation in this thesis	Residual risk
Internal	Label-matching and issue-level FP counting can bias precision/FPR.	Fixed JSON schema, repeated runs, explicit reporting of scoring rules.	Partial label mismatches still distort measured precision/recall.
Construct	Core metrics may miss practical developer value (localization, fix quality).	Added qualitative case studies and repair-quality criteria.	No fully automated metric for fix correctness/minimality yet.
External	Curated synthetic snippets may not reflect real multi-file systems.	Included diverse CWE/OWASP-aligned classes and secure cases.	Generalization to production repositories remains uncertain.
Measurement interpretation	Repeated measurements on the same snippets reduce independence of pooled counts.	Reported raw counts/percentages and explicit mode-to-mode recall deltas.	Paired per-sample mode comparison remains limited by current exported outputs.

Internal validity. The evaluation harness scores detections at the vulnerability-type level using substring matching between expected and predicted category names. This reduces brittleness to naming variation but can also over-credit partially correct labels (e.g., a broad "Injection" label matching multiple injection subclasses) or under-credit semantically correct but differently phrased labels. In addition, false positives are counted at issue level, so precision/FPR denominators are sensitive to how many issues a model emits per sample. A stricter label ontology plus CWE-level normalization would improve precision of the evaluation itself.

Construct validity. Precision/recall and parse success rate measure important properties for IDE integration, but they do not fully capture developer value. In practice, usefulness also depends on (i) localization quality (correct lines), (ii) explanation clarity, and (iii) the security adequacy and minimality of suggested fixes. These are addressed in part through qualitative case studies (Section 5.7) but are not yet fully quantified.

External validity. Synthetic snippets are easier than real codebases with frameworks, configuration files, and cross-file flows. Results on the curated datasets should

therefore be interpreted as an estimate of core capability under controlled conditions rather than a definitive measure of real-world performance. The most likely failure modes in practice are missing context when only a small scope (e.g., a single function) is analyzed, and output-structure instability for certain model families.

Reproducibility Notes

To support reproducibility, the benchmark suite and evaluation harness are kept alongside the prototype in the thesis workspace, and run-specific environment/configuration details are reported in Section 5.3. Repeating runs remains important for estimating variability due to runtime effects (warm-up, caching, and transient system load) even under low-temperature decoding.

6. Conclusion

This chapter summarizes the contributions of this thesis, highlights key findings, discusses limitations, and outlines implications for practice. The goal of the thesis was to design and implement a privacy-preserving vulnerability detection and repair assistant for Visual Studio Code that leverages locally deployed LLMs and retrieval-augmented grounding without transmitting source code to external services.

6.1. Summary of Contributions

Code Guardian: an IDE-integrated, local security assistant. This thesis delivers **Code Guardian**, a VS Code extension that performs on-device vulnerability analysis for JavaScript and TypeScript projects. Findings are presented using IDE-native diagnostics, and optional quick fixes provide repair suggestions while keeping developers in full control of code changes.

Privacy-preserving LLM inference with optional RAG. All code analysis runs locally via Ollama. To improve grounding, the system optionally augments prompts with locally retrieved security knowledge (CWE/OWASP/CVE guidance) using a local vector index and local embeddings. This architecture supports explainability and consistency while preserving the no-exfiltration requirement.

Practical performance mechanisms. To remain usable during development, Code Guardian combines debounced triggers, function-level scoping for real-time use, and caching of repeated analyses. These mechanisms reduce unnecessary inference calls and support responsive IDE feedback.

Reproducible evaluation harness. The prototype includes a curated benchmark of security test cases and a local evaluation script for comparing models and configurations using standard detection metrics, parse robustness, and latency.

6.2. Answers to Research Questions

RQ1 (Feasibility). The thesis supports feasibility with caveats. Code Guardian demonstrates that useful vulnerability analysis and repair suggestions can be generated inside VS Code using fully local inference, satisfying the no-code-exfiltration

objective. However, practical usefulness depends on selecting a model profile that matches the deployment context (interactive editing vs. audit workflows).

RQ2 (Grounding). Retrieval augmentation is beneficial only under model-dependent conditions. In this run, RAG improved `qwen3:8b` (F1 35.64% → 42.70%) and slightly improved `gemma3:4b` (F1 30.67% → 31.37%), but reduced `gemma3:1b` recall to zero true positives. Grounding should therefore be treated as a tunable strategy rather than a universally positive default.

RQ3 (Practicality). IDE practicality is achievable when inference is constrained by function-level scoping, debounced triggers, and caching. These mechanisms keep local analysis responsive for small models, while larger models remain better suited to on-demand scans where higher latency is acceptable.

6.3. Deployment Implications

Table 6.1 summarizes pragmatic deployment choices based on the measured trade-offs.

Table 6.1.: Recommended deployment profiles from thesis results.

Use case	Preferred configuration	Main advantage	Main risk
Real-time editor feedback	`gemma3:1b` (LLM-only)	Very low latency (about 178 ms median) and stable parsing	Misses many vulnerabilities (recall 5.56%)
Moderate-latency audit pass	`gemma3:4b` (LLM+RAG)	Higher recall (44.44%) with about 1.3 s median latency	Severe alert noise on secure samples (FPR 100%)
Higher-precision deep audit	`qwen3:8b` (LLM+RAG)	Best F1 (42.70%) and lower FPR (27.12%) than other high-recall modes	Multi-second latency (median 7809 ms) and non-perfect parse rate (84.85%)
Unstable model profiles	`qwen3:4b` / `CodeLlama:latest`	N/A in this run	Parse-collapse regimes (1.01–5.05% parse), yielding near-zero recall

6.4. Limitations

Scope limits and contextual depth. While the system can flag common vulnerability patterns, deep semantic reasoning across files (e.g., source-to-sink flows spanning modules) is limited by the analysis scope and the absence of full static data-flow analysis.

Evaluation representativeness. The curated dataset is intentionally small and human-auditable, but it does not fully reflect the diversity and ambiguity of real-world codebases. Results should therefore be interpreted as indicative rather than definitive.

Repair correctness. Repair suggestions are model-generated and may affect behavior beyond security hardening. The system mitigates this by requiring explicit user review, but comprehensive functional validation remains outside the scope of the extension.

6.5. Responsible Use

This thesis treats Code Guardian as a decision-support system, not an autonomous security verifier. Both false negatives and false positives were observed, and some findings used semantically plausible but ontology-mismatched labels. For this reason, findings and repair suggestions should remain reviewable artifacts under developer control, with conventional testing and security review retained as mandatory safeguards before release.

6.6. Conclusion

Privacy-preserving secure coding assistance is feasible within the IDE when local LLM inference is combined with careful prompt structuring, optional retrieval grounding, and developer-controlled remediation workflows. The evaluation shows a clear quality-latency-robustness trade-off: `gemma3:1b` remained fast (about 178–180 ms median) but had very low recall, `gemma3:4b` increased recall at around 1.3–1.4 s median latency but over-warned on secure code, and `qwen3:8b` achieved the best F1 with markedly higher multi-second latency.

The results also show that retrieval augmentation is not universally beneficial; its effect is model-dependent. In this study, RAG improved `qwen3:8b` and marginally improved `gemma3:4b`, but reduced `gemma3:1b` to zero true positives and did not recover parse-collapse regimes for `qwen3:4b` and `CodeLlama:latest`. This indicates that RAG integration must be calibrated per model and prompt format, not assumed to improve security detection by default.

Overall, Code Guardian demonstrates that locally deployed LLMs can provide useful vulnerability detection and repair suggestions without transmitting source code off-device, but practical deployment requires explicit configuration choices for model class, latency budget, and acceptable false-positive behavior.

7. Future Work

Building on the contributions and limitations above, several next steps stand out for improving Code Guardian's effectiveness, robustness, and evaluation depth.

Deeper program analysis and cross-file context. Add lightweight static analysis to extract taint-style source-to-sink traces across functions and files, and feed these traces into the LLM prompt. This can reduce false positives from missing context and improve recall for vulnerabilities that span modules (e.g., validation in one file and sink usage in another).

Hybrid integration with traditional SAST. Integrate rule-based baselines (e.g., Semgrep) as an additional signal. A hybrid approach can use SAST findings as candidate locations and let the LLM focus on contextual reasoning and repair generation, improving both precision and developer trust.

Repair validation and safer patching. Extend repair suggestions with syntactic checks and minimal local validation (e.g., TypeScript typechecking on modified regions). Provide diff previews by default and track when suggested fixes introduce new warnings.

Adversarial robustness. Study prompt-injection and retrieval-poisoning risks within the IDE context (e.g., attacker-controlled comments or dependency code). Add provenance and filtering for retrieved knowledge, and implement safe prompt templates that explicitly treat code as data.

Broader evaluation on standard benchmarks and real repositories. Complement the curated dataset with larger benchmarks (e.g., OWASP Benchmark, Juliet-style suites) and selected real-world CVE cases. Export paired per-sample outputs for both modes and report mode-to-mode deltas and per-category breakdowns under matched conditions.

User studies in realistic IDE workflows. Run a developer study to measure time-to-fix, perceived usefulness, trust calibration, and false-positive tolerance. Standard usability and workload instruments such as SUS and NASA-TLX can complement objective metrics [5, 20]. Compare LLM-only vs. LLM+RAG configurations under real editing sessions to validate R6 and practical adoption constraints.

A. Detailed Implementation Results

A.1. Prompt Contract (JSON-Only Output)

Code Guardian relies on a strict output contract so findings can be converted into IDE diagnostics reliably. The analyzer instructs the local model to return *only* a JSON array of security issues.

Listing A.1: JSON-only response schema (conceptual)

```
[
  {
    "message": "Issue description",
    "startLine": 1,
    "endLine": 3,
    "suggestedFix": "Optional secure alternative"
  }
]
```

Robustness in practice

Even with explicit instructions, some models occasionally emit Markdown fences or additional text. The prototype therefore applies defensive parsing: it removes common code-block markers and extracts the first JSON array substring before attempting to parse. This is a pragmatic mechanism to improve structured-output robustness for IDE integration.

A.2. Runtime Guardrails

To keep the extension usable on developer hardware (R6), Code Guardian includes conservative guardrails:

- **Debounce interval:** 800 ms for real-time analysis after document changes.
- **Function scope size limit:** extracted functions larger than 2000 characters are skipped in real-time mode.

- **File scope size limit:** full-file analysis is skipped above 20,000 characters.
- **Workspace scan file size limit:** files larger than 500 KB are skipped in batch scans.

In addition, analysis results are cached using an LRU-style cache (100 entries, 30-minute TTL) to reduce redundant inference calls during iterative edits.

A.3. Key Configuration Options

The extension exposes user configuration through VS Code settings. The most relevant options are:

- `codeGuardian.model`: default Ollama model for analysis
- `codeGuardian.ollamaHost`: Ollama base URL (default: `http://localhost:11434`)
- `codeGuardian.enableRAG`: enable/disable retrieval augmentation

A.4. VS Code Commands (Prototype)

The prototype exposes the following main commands via the Command Palette:

- **Analyze selected code with AI**: opens the interactive analysis view for a selection or current line.
- **Analyze full file**: runs the structured diagnostics pipeline over the active document.
- **Contextual Q&A**: opens a WebView for asking security questions with user-selected file/folder context.
- **Select AI model**: lists available local Ollama models and switches the active model.
- **Manage RAG knowledge base**: view/add/search knowledge, rebuild the vector store, and update vulnerability data.
- **Toggle RAG**: enables/disables retrieval augmentation through settings.
- **Update vulnerability data**: refreshes cached public metadata used for the knowledge base.
- **View cache statistics**: inspects the analysis cache and supports clearing/resetting statistics.

- **Workspace security dashboard**: performs a batch scan and displays an aggregate dashboard.

A.5. Evaluation Harness Location

The evaluation script used in Chapter 5 is located at `code-guardian/evaluation/evaluate-models.js`. The curated datasets are located in `code-guardian/evaluation/datasets/`.

B. Detailed Experimental Results

B.1. Curated Test Suite Overview

The curated evaluation suite maintained in `code-guardian/evaluation/datasets/` contains representative vulnerability snippets for JavaScript/TypeScript secure coding. Expected vulnerabilities include CWE identifiers and severities so results can be aggregated by category.

Dataset sizes

For the scored thesis run in Chapter 5, the harness uses:

- **Primary scored dataset:** `vulnerability-test-cases.json` with 33 test cases (18 vulnerable, 15 secure)

Additional datasets can be added for broader coverage, but all quantitative results in this thesis are derived from the 33-case scored dataset above.

Representative Vulnerability Classes

The dataset includes (non-exhaustive) examples for:

- SQL injection (CWE-89)
- Cross-site scripting (CWE-79)
- Command injection (CWE-78)
- Path traversal (CWE-22)
- Insecure randomness (CWE-338)
- Hardcoded credentials (CWE-798)
- CSRF and authentication-flow weaknesses (CWE-352, CWE-287)
- Prototype pollution / unsafe reflection patterns (CWE-1321, CWE-470)

Test Case Record Format

Each test case contains:

- a code snippet (`code`),
- a list of expected findings (`expectedVulnerabilities`),
- and optional remediation guidance (`expectedFix`).

Reproducing the harness run

The evaluation script is executed locally:

Listing B.1: Running the evaluation script

```
cd code-guardian
node evaluation/evaluate-models.js --ablation --runs=3 --rag-k=5 --temp
  --num-predict=1000 --timeout-ms=30000 --delay-ms=500
```

The script prints per-model precision/recall/F1, false positive rate, average response time, and JSON parse success rate. Models to test are specified in the script and can be edited to match the locally installed Ollama models.

Recommended Artifact Checklist

For full reproducibility and auditability, the following files should be archived with the thesis:

- Raw run output JSON from the harness (all configurations and per-case results)
- Exact run configuration object (models, prompt modes, runtime parameters)
- Model digests / quantization metadata used for inference
- Environment metadata (OS, Node.js, Ollama versions, hardware)
- Generated tables or scripts used to derive reported descriptive metrics and mode-to-mode recall deltas

This appendix is intentionally concise: the full dataset is machine-readable and can be inspected directly in the repository.

Bibliography

[1] Allamanis, M., Brockschmidt, M., Khademi, M.: Learning to represent programs with graphs. In: Proceedings of the International Conference on Learning Representations (ICLR) (2018)

[2] Alon, U., Zilberstein, M., Levy, O., Yahav, E.: Code2vec: Learning distributed representations of code. In: Proceedings of the ACM on Programming Languages (POPL) (2019)

[3] Bender, E.M., Gebru, T., McMillan-Major, A., Shmitchell, M.: On the dangers of stochastic parrots: Can language models be too big? In: Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency (FAccT) (2021)

[4] Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S., Engler, D.: A few billion lines of code later: Using static analysis to find bugs in the real world. Communications of the ACM **53**(2), 66–75 (2010)

[5] Brooke, J.: SUS: A quick and dirty usability scale. Usability Evaluation in Industry pp. 189–194 (1996)

[6] Brown, T.B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al.: Language models are few-shot learners. In: Advances in Neural Information Processing Systems (NeurIPS) (2020)

[7] Card, S.K., Moran, T.P., Newell, A.: The Psychology of Human-Computer Interaction. CRC Press, Boca Raton, FL, USA (1983), reprint edition 1991

[8] Carlini, N., Tramer, F., Wallace, E., Jagielski, M., Herbert-Voss, A., Lee, K., Roberts, A., Brown, T., Song, D., Erlingsson, Ú., Oprea, A., Raffel, C.: Extracting training data from large language models. In: Proceedings of the 30th USENIX Security Symposium (2021)

[9] Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H.P.d.O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Hilton, J., Nakano, R., Hesse, C., Chen, J., Plappert, M., Beard, M., Voss, C., Radford, A., Sutskever, I.: Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374 (2021)

[10] Chess, B., McGraw, G.: Static analysis for security. IEEE Security & Privacy **2**(6), 76–79 (2004)

[11] Christakis, M., Bird, C.: What developers want and need from program analysis: An empirical study. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 332–343. Singapore (2016)

[12] Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. Proceedings of the 4th ACM Symposium on Principles of Programming Languages (POPL) pp. 238–252 (1977)

[13] Devlin, J., Chang, M.W., Lee, K., Toutanova, K.: BERT: Pre-training of deep bidirectional transformers for language understanding. In: Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics (NAACL-HLT) (2019)

[14] Engler, D., Chen, D.Y., Hallem, S., Chou, A., Chelf, B.: Bugs as deviant behavior: A general approach to inferring errors in systems code. In: Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP). pp. 57–72 (2001)

[15] European Union: Regulation (eu) 2016/679 (general data protection regulation). `https://eur-lex.europa.eu/eli/reg/2016/679/oj` (2016), accessed: 2026-01-24

[16] Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., Zhou, M.: Codebert: A pre-trained model for programming and natural languages. In: Findings of the Association for Computational Linguistics: EMNLP (2020)

[17] FIRST: Common vulnerability scoring system v3.1: Specification document. `https://www.first.org/cvss/specification-document`, accessed: 2026-01-24

[18] GitHub: Codeql documentation. `https://codeql.github.com/docs/`, accessed: 2026-01-24

[19] Guu, K., Lee, K., Tung, Z., Pasupat, P., Chang, M.W.: REALM: Retrieval-augmented language model pre-training. In: Proceedings of the 37th International Conference on Machine Learning (ICML) (2020)

[20] Hart, S.G., Staveland, L.E.: Development of NASA-TLX (task load index): Results of empirical and theoretical research. In: Advances in Psychology. vol. 52, pp. 139–183. North-Holland (1988)

[21] Howard, M., Lipner, S.: The Security Development Lifecycle. Microsoft Press (2006)

[22] Ji, Z., Lee, N., Frieske, R., Yu, T., Su, D., Xu, Y., Ishii, E., Bang, Y., Madotto, A., Fung, P.: Survey of hallucination in natural language generation. ACM Computing Surveys **55**(12) (2023)

[23] Johnson, B., Song, Y., Murphy-Hill, E., Bowdidge, R.: Why don't software developers use static analysis tools to find bugs? In: Proceedings of the 35th International Conference on Software Engineering (ICSE). pp. 672–681. San Francisco, CA, USA (2013)

[24] Johnson, J., Douze, M., Jégou, H.: Billion-scale similarity search with GPUs. arXiv preprint arXiv:1702.08734 (2017)

[25] Karpukhin, V., Oguz, B., Min, S., Wu, L., Edunov, S., Chen, D., Yih, W.t.: Dense passage retrieval for open-domain question answering. In: Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP) (2020)

[26] Kim, D., Nam, J., Song, J., Kim, S.: Automatic patch generation learned from human-written patches. In: Proceedings of the 2013 International Conference on Software Engineering (ICSE). pp. 802–811 (2013)

[27] LangChain: Langchain documentation. `https://python.langchain.com/`, accessed: 2026-01-24

[28] Le Goues, C., Nguyen, T., Forrest, S., Weimer, W.: Genprog: A generic method for automatic software repair. IEEE Transactions on Software Engineering **38**, 54–72 (2012)

[29] Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.t., Rocktäschel, T., Riedel, S., Kiela, D.: Retrieval-augmented generation for knowledge-intensive NLP tasks. In: Advances in Neural Information Processing Systems (NeurIPS) (2020)

[30] Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., Chen, Z., Wang, S.: Vuldeepecker: A deep learning-based system for vulnerability detection. In: Proceedings of the Network and Distributed System Security Symposium (NDSS) (2018)

[31] Livshits, B., Lam, M.S.: Finding security vulnerabilities in java applications with static analysis. In: Proceedings of the 14th USENIX Security Symposium (2005)

[32] Malkov, Y.A., Yashunin, D.A.: Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. IEEE Transactions on Pattern Analysis and Machine Intelligence **42**(4), 824–836 (2020)

[33] McGraw, G.: Software Security: Building Security In. Addison-Wesley (2006)

[34] Mialon, G., Dessì, R., Lomeli, M., Nalmpantis, C., Pasunuru, R., Raileanu, R., Rozière, B., Schick, T., Scialom, T., Suau, X., et al.: Augmented language models: A survey. arXiv preprint arXiv:2302.07842 (2023)

[35] Microsoft: Language server protocol specification. https://microsoft.github.io/language-server-protocol/, accessed: 2026-01-24

[36] Microsoft: Visual studio code extension API. https://code.visualstudio.com/api, accessed: 2026-01-24

[37] MITRE: Common vulnerabilities and exposures (CVE). https://www.cve.org/, accessed: 2026-01-24

[38] MITRE: Common weakness enumeration (CWE). https://cwe.mitre.org/, accessed: 2026-01-24

[39] MITRE: CWE top 25 most dangerous software weaknesses. https://cwe.mitre.org/top25/, accessed: 2026-01-24

[40] Monperrus, M.: A critical review of automatic patch generation learned from human-written patches: Essay on the problem statement and the evaluation of automatic software repair. In: Proceedings of the 36th International Conference on Software Engineering (ICSE). pp. 234–242 (2014)

[41] National Institute of Standards and Technology: Juliet test suite for C/C++ and Java. https://samate.nist.gov/SARD/test-suites/, accessed: 2026-01-24

[42] National Institute of Standards and Technology: National vulnerability database (NVD). https://nvd.nist.gov/, accessed: 2026-01-24

[43] National Institute of Standards and Technology: Secure software development framework (SSDF) version 1.1. Tech. Rep. SP 800-218, NIST (2022), accessed: 2026-01-24

[44] Nielsen, J.: Usability Engineering. Morgan Kaufmann, San Francisco, CA, USA (1994)

[45] Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., Savarese, S.: Codegen: An open large language model for code with multi-turn program synthesis. arXiv preprint arXiv:2203.13474 (2022)

[46] Ollama: Ollama documentation. https://ollama.com/, accessed: 2026-01-24

[47] OpenAI: GPT-4 technical report. arXiv preprint arXiv:2303.08774 (2023)

[48] OWASP Foundation: Cross-site request forgery prevention cheat sheet. https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html, accessed: 2026-01-24

[49] OWASP Foundation: Cross site scripting (xss) prevention cheat sheet. https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html, accessed: 2026-01-24

[50] OWASP Foundation: Deserialization cheat sheet. https://cheatsheetseries.owasp.org/cheatsheets/Deserialization_Cheat_Sheet.html, accessed: 2026-01-24

[51] OWASP Foundation: Input validation cheat sheet. https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html, accessed: 2026-01-24

[52] OWASP Foundation: Logging cheat sheet. https://cheatsheetseries.owasp.org/cheatsheets/Logging_Cheat_Sheet.html, accessed: 2026-01-24

[53] OWASP Foundation: Owasp application security verification standard (ASVS). https://owasp.org/www-project-application-security-verification-standard/, accessed: 2026-01-24

[54] OWASP Foundation: Owasp benchmark project. https://owasp.org/www-project-benchmark/, accessed: 2026-01-24

[55] OWASP Foundation: Owasp cheat sheet series. https://cheatsheetseries.owasp.org/, accessed: 2026-01-24

[56] OWASP Foundation: Password storage cheat sheet. https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html, accessed: 2026-01-24

[57] OWASP Foundation: Sql injection prevention cheat sheet. https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html, accessed: 2026-01-24

[58] OWASP Foundation: Unrestricted file upload cheat sheet. https://cheatsheetseries.owasp.org/cheatsheets/File_Upload_Cheat_Sheet.html, accessed: 2026-01-24

[59] OWASP Foundation: Owasp top 10 – 2021. https://owasp.org/www-project-top-ten/ (2021), accessed: 2026-01-24

[60] OWASP Foundation: Owasp top 10 for large language model applications. https://owasp.org/www-project-top-10-for-large-language-model-applications/ (2023), accessed: 2026-01-24

[61] Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., Karri, R.: Asleep at the keyboard? assessing the security of GitHub Copilot's code contributions. In: Proceedings of the IEEE Symposium on Security and Privacy (S&P) (2022)

[62] Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., Liu, P.J.: Exploring the limits of transfer learning with a unified text-to-text transformer. Journal of Machine Learning Research **21** (2020)

[63] Reimers, N., Gurevych, I.: Sentence-bert: Sentence embeddings using siamese bert-networks. In: Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing (EMNLP-IJCNLP) (2019)

[64] Robertson, S., Zaragoza, H.: The probabilistic relevance framework: BM25 and beyond. In: Foundations and Trends in Information Retrieval, vol. 3, pp. 333–389. Now Publishers (2009)

[65] Schick, T., Dwivedi-Yu, J., Dessì, R., Raileanu, R., Lombardi, M., Zettlemoyer, L., Cancedda, N., Scialom, T.: Toolformer: Language models can teach themselves to use tools. In: Proceedings of the International Conference on Learning Representations (ICLR) (2023)

[66] Semgrep, Inc.: Semgrep documentation. `https://semgrep.dev/docs/`, accessed: 2026-01-24

[67] Shokri, R., Stronati, M., Song, C., Shmatikov, V.: Membership inference attacks against machine learning models. In: Proceedings of the 2017 IEEE Symposium on Security and Privacy (S&P) (2017)

[68] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., Polosukhin, I.: Attention is all you need. In: Advances in Neural Information Processing Systems (NeurIPS) (2017)

[69] Wang, Y., Wang, W., Joty, S., Hoi, S.C.H.: Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In: Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP) (2021)

[70] Weidinger, L., Mellor, J., Rauh, M., Griffin, C., Uesato, J., Huang, P.S., Cheng, M., Balle, B., Kasirzadeh, A., et al.: Ethical and social risks of harm from language models. arXiv preprint arXiv:2112.04359 (2021)

[71] Weimer, W., Nguyen, T., Le Goues, C., Forrest, S.: Automatically finding patches using genetic programming. In: Proceedings of the 31st International Conference on Software Engineering (ICSE) (2009)

[72] Zhou, Y., Liu, S., Siow, J.K., Du, X., Liu, Y.: Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In: Advances in Neural Information Processing Systems (NeurIPS) (2019)

[73] Zou, A., Wang, Z., Kolter, J.Z., Fredrikson, M.: Universal and transferable adversarial attacks on aligned language models. arXiv preprint arXiv:2307.15043 (2023)