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 in demand, 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suggestions; however, their adoption is constrained by inconsistent detection quality, high false-positive rates, outdated security knowledge, and significant privacy concerns when proprietary source code is processed by cloud-based services.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 on-device, preserving a local source-code boundary while optionally using locally cached security knowledge that can be refreshed 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 curated JavaScript/TypeScript evaluation set of 128 cases (113 vulnerable, 15 secure) aligned to CWE/OWASP concepts and validated against real-world CVE patterns. The quantitative comparison covers LLM-only and LLM+RAG configurations across precision, recall, F1-score, false-positive rate, JSON parse success, and latency. Results show a clear model-dependent trade-off: smaller models are faster but miss more vulnerabilities, several mid-sized configurations improve recall at the cost of high alert noise, and the best F1 score (63.76%) is achieved by `qwen3:8b` with RAG. Overall, the study shows that privacy-preserving local LLM workflows can provide useful vulnerability analysis support, while model selection and false-positive control 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 am grateful to everyone who supported me during this Master's thesis.

My sincere thanks go to my internal supervisor, *Abubaker Gaber, M.Sc.*, for his continuous guidance, constructive feedback, and thoughtful discussions throughout the project. His support was essential in shaping both the technical direction and the academic quality of this work.

I also thank the academic staff of the Master's program for providing a strong foundation and a stimulating learning environment.

I am equally thankful to my colleagues and peers for their encouragement and for creating a supportive atmosphere during the study period.

Most of all, I thank my family for their patience, trust, and constant encouragement. Their support made it possible for me to complete this thesis.

Task Description

Traditional rule-based security analysis tools are effective for detecting predefined vulnerability patterns but can struggle with context-dependent weaknesses that require semantic reasoning. Cloud-based language model assistants provide stronger contextual analysis, yet compromise confidentiality and reproducibility because proprietary source code must leave the local environment. This thesis designs and evaluates a fully local, privacy-preserving secure coding assistant for Visual Studio Code that integrates large language models (LLMs) with Retrieval-Augmented Generation (RAG). The prototype is built around a no-source-code-exfiltration boundary: analyzed code and IDE-derived context are processed on-device and sent only to a local LLM runtime.

The assistant combines two complementary modes: real-time inline diagnostics that deliver immediate vulnerability alerts, and asynchronous audit and question–answering modes for deeper inspection. In audit mode, the prompt can optionally be grounded with a local retrieval index over security guidance (e.g., CWE/OWASP-aligned summaries and optionally cached public CVE/NVD metadata). In question–answering mode, users provide explicit file/folder context and receive narrative security review guidance. A modular architecture separates the LLM inference layer, the retrieval pipeline, and the Visual Studio Code extension interface. Privacy is enforced primarily through local deployment (no external inference APIs) and by limiting knowledge refresh to public metadata; reproducibility is supported by version-pinned artifacts and explicit reporting of model fingerprints and runtime settings. The threat model includes prompt injection via attacker-controlled code text and retrieval-quality risks; the prototype mitigates these risks primarily through strict JSON output contracts in diagnostic workflows, defensive parsing, and curated retrieval sources, while stronger provenance controls remain future work.

Evaluation is conducted on a project-authored JavaScript/TypeScript suite aligned to CWE/OWASP concepts, combining vulnerable and secure/negative cases to quantify both detection quality and false-positive behavior. The comparison covers LLM-only and LLM+RAG prompting, and includes baseline SAST tools (Semgrep, CodeQL, and ESLint security rules) to contextualize trade-offs.

Evaluation metrics include precision, recall, F1-score, secure-sample false-positive rate, JSON parse success, and latency. Privacy and reproducibility are assessed primarily through architectural/configuration evidence (local inference boundary,

optional refresh behavior) and run-provenance reporting under controlled 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 [**?** **?** **?**]. These weaknesses persist despite widespread adoption of secure development practices and automated analysis tools [**?** **?** **?**].

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 [**?** **?** **?**]. However, both empirical studies and practitioner experience highlight limitations, including false positives, difficulty capturing framework-specific semantics, and limited fix guidance [**?** **?**]. 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 [**?** **?**].

Recent advances in Large Language Models (LLMs) have demonstrated strong capabilities in source code understanding, generation, and transformation [**?** **?** **?**]. 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 [**?** **?**].

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 [**?** **?**]. 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 [**?** **?** **?** **?**]. 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 [**?** **?**].

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 [**?**].	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).
- **Empirical results:** evaluation demonstrates that qwen3:8b with RAG achieves the best balance of detection quality (F1 88.9%), false-positive control (FPR 20%), and responsiveness (p95 latency 2.5 s) among privacy-preserving configurations (Chapter **??**).

1.4 Research Method and Evaluation Strategy

The thesis follows a design-and-evaluate methodology. First, requirements are derived from the problem analysis and related work, with explicit attention to privacy constraints, explainability needs, and IDE usability. Second, these requirements are translated into a concrete architecture and implemented as a working VS Code extension. Third, the implemented system is assessed through reproducible local experiments that compare models and prompt modes under matched runtime settings.

Methodologically, this work combines quantitative and qualitative evidence. Quantitative scoring focuses on detection behavior (precision, recall, F1, false-positive behavior on secure samples), structured-output reliability (JSON parse success), and responsiveness (latency). Qualitative analysis complements these metrics by examining representative true-positive, false-positive, and false-negative cases, with emphasis on explanation quality and practical repair usefulness in editor workflows.

This mixed evidence strategy is intentional: security tooling quality cannot be inferred from one metric alone. High recall with persistent secure-sample over-warning can still be operationally expensive, and low latency without sufficient recall provides limited security value. By reporting both measurable outcomes and workflow-level

observations, the thesis aims to provide a decision-relevant assessment of when local LLM assistance is useful and where stronger safeguards are still required.

1.5 Thesis Structure

Chapter ?? derives requirements for privacy-preserving vulnerability detection and repair assistance. Chapter ?? presents the conceptual system design and its mapping to the requirements. Chapter ?? details the implementation of Code Guardian as a VS Code extension with local inference and optional retrieval grounding. Chapter ?? evaluates detection quality, structured output robustness, and responsiveness. Chapter ?? concludes the thesis and Chapter ?? 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 **??** 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 baseline for the proposed system. The following chapters map them to concrete architectural and implementation choices, and Chapter **??** tests them using reproducible quantitative evidence.

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 [**? ? ?**].

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 [**?**].

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. In this thesis run, the operational primary indicator is *pooled issue-level F1* across repeated runs, complemented by parse stability and qualitative checks of label/localization stability. Macro-averaged class-wise agreement remains the preferred long-term measure when per-class paired outputs are fully exported, but is not the scoring primitive used by the current harness.

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

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

R1 is fundamentally about predictable behavior: the same code should lead to comparable findings, labels, explanations, and localization across runs. Without that stability, developers cannot reliably act on the tool's output in day-to-day 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 [**?** **?**]. 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 [**?** **?**].

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 over context-rich scenarios where surface patterns alone are insufficient. In this thesis run, the operational quantitative proxy is pooled issue-level precision/recall/F1 under fixed scoring rules, complemented by localization observations and qualitative case studies (mitigation recognition, representative misses, and over-warning behavior). Macro-averaged class-wise context metrics remain a preferred extension when class-balanced paired outputs are exported at finer granularity.

Reasoning Level	**Interpretation (operational proxy criteria)**
High	**High context awareness.** The system correctly integrates dispersed context, recognizes mitigation logic, and avoids unsupported inference. Vulnerability classification and localization are accurate (pooled issue-level F1 ≥ 0.80 plus strong qualitative evidence).
Medium	**Moderate context awareness.** The system integrates some contextual information but occasionally misses mitigations or misinterprets dependencies (pooled issue-level 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 (pooled issue-level F1 in $[0.50, 0.65)$).
None	**No context awareness.** The system fails to incorporate contextual information and produces unreliable vulnerability assessments (pooled issue-level F1 < 0.50).

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

R2 requires semantic reasoning beyond surface patterns. When the assistant can combine dispersed context, account for mitigation logic, and explicitly handle uncertainty, vulnerability assessments align more closely with real security risk.

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 devel-

opers. 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 [**?** **?**]. 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.

R3 ensures that findings are understandable and reviewable, not just correct in aggregate metrics. Clear links between code evidence, vulnerability class, and suggested action are necessary for developer trust and accountable use.

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 [**?** **?**]. 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 [**?** **?** **?**]. 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 [**?** **?** **?** **?**].

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 [**?** **?** **?** **?** **?** **?**].

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.

R4 bridges detection and remediation. In practice, suggestions are valuable only when they are specific, context-appropriate, and safe to review incrementally, while final control remains with the developer.

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 can be assessed through architectural inspection and runtime verification. In this thesis run, evidence is primarily architectural and configuration-based (local inference boundary, no source-code egress path in the analyzed workflows), while dedicated outbound-traffic instrumentation and formal offline verification logs are not captured as separate artifacts. A stricter operational audit would add explicit network-monitor traces and repeatable offline-mode execution logs.

R5 defines the non-negotiable boundary of this thesis: source code and derived artifacts remain on-device during analysis. This local-first design is central to adoption in confidentiality-sensitive environments.

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.

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 [**?** **?**]. 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 in this thesis run, usability is operationalized with **median** end-to-end latency (typical interaction cost) and **mean** latency (slow-tail sensitivity proxy) across representative interaction scenarios. This matches the current harness output and avoids overstating p95 claims that are not directly exported in the reported artifacts. Separate latency measurements are still reported for interactive inline detection and more comprehensive explicitly triggered analyses.

In this thesis, usability is evaluated primarily through responsiveness under realistic hardware constraints. Reporting latency together with environment details keeps performance claims comparable and interpretable.

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.

Visual Score	Interpretation (median/mean criteria)
	**High usability.** Median end-to-end latency $\leq 2\,\mathrm{s}$ and mean latency $\leq 3\,\mathrm{s}$ for inline detection on the declared hardware profile. Interaction remains fluid and non-disruptive.
	**Medium usability.** Median latency in $(2, 5)\,\mathrm{s}$ or mean latency in $(3, 6)\,\mathrm{s}$. Delay is noticeable but acceptable for non-continuous security feedback.
	**No usability.** Median latency $> 5\,\mathrm{s}$, mean latency $> 6\,\mathrm{s}$, or frequent timeouts in inline scenarios. Feedback is too slow for practical integration into the development workflow.

Table 2.6: Evaluation scale for R6: Usability (Latency, median/mean operationalization).

2.2.1 Traditional Static Application Security Testing

Static Application Security Testing (SAST) tools remain the default workhorse for vulnerability detection in many development workflows. Rule-based analyzers and taint-analysis systems such as Semgrep and CodeQL statically inspect source code for predefined patterns and data-flow queries [**?** **?**]. Their main advantage is operational: results are deterministic, reproducible, and easy to integrate into CI/CD pipelines and compliance-driven environments.

From a research perspective, modern static analyzers build on foundational ideas such as abstract interpretation [**?**] and scalable bug-finding techniques [**?**]. In practice, large-scale deployments demonstrate that static analysis can find real defects in industrial codebases at massive scale [**?**]. 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 [**?** **?** **?**].

A second advantage of SAST is *auditability*. Findings are tied to explicit rules or queries, which enables traceable reasoning, stable regression behavior, and predictable security gates. This is particularly relevant in regulated settings, where organizations need evidence of controls rather than only aggregate accuracy claims. Determinism also simplifies governance: when a rule is updated, teams can review the diff in findings directly instead of inferring changes from opaque model behavior.

These strengths come with well-known limitations. SAST tools can struggle with contextual reasoning and generalization, producing false positives when a risky-looking

pattern is guarded elsewhere (e.g., validation in a different function, framework-enforced invariants) and false negatives when vulnerabilities depend on semantic relationships or framework-specific behavior. Language and framework abstractions further complicate exhaustive rule coverage and often push teams toward conservative, noisy rulesets.

There is also a structural precision/recall tension in static analysis deployments. Aggressive rule sets can improve vulnerable-case coverage, but often increase triage burden through noisy alerts; stricter rules reduce noise but risk under-detection. The trade-off is not only technical but organizational: teams with limited security-review capacity may prefer conservative rule sets, while high-assurance environments may tolerate larger alert queues to avoid misses.

Finally, many SAST tools provide limited remediation guidance, requiring developers to interpret findings and design fixes manually. Empirical studies show that warning overload and poor actionability reduce adoption and remediation rates [**?** **?**]. This gap between *detection* and *remediation support* is one reason why SAST outputs are often strongest as gatekeeping signals, but weaker as day-to-day developer coaching tools.

These limitations motivate approaches that keep deterministic checks as guardrails while adding more flexible reasoning and explanation generation closer to the developer workflow. In this thesis, Code Guardian follows this complementary idea: it preserves IDE diagnostics and baseline SAST comparison, while using grounded LLM reasoning 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 renewed interest in using learned models for code understanding, vulnerability detection, and repair suggestions. Contemporary systems build on transformer architectures [**?**] and large-scale pretraining objectives in the style of BERT/T5 [**?** **?**]. Both general-purpose LLMs and code-specialized models can generate syntactically plausible code and perform transformations such as refactoring and patch drafting [**?** **?** **?** **?** **?** **?**]. For IDE-integrated security assistance, these capabilities are attractive because they enable 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 [**?** **?** **?**]. This matters disproportionately in security, where small omissions (e.g., missing validation, unsafe sinks) can create exploitable weaknesses.

A key difference from classical static analysis is that LLM behavior is strongly prompt- and format-dependent. Small changes in instructions, output schema, or contextual

examples can materially change both finding rates and false-positive behavior. As a result, measured model quality is not only a property of model weights, but also of engineering choices such as scope selection, schema strictness, retrieval context, and failure handling.

For IDE integration, this dependence has an additional implication: free-form natural language answers are often insufficient for automation. Practical tools typically require machine-checkable outputs (e.g., JSON findings with line spans) and conservative handling of malformed responses. In this thesis, structured-output robustness is therefore treated as a deployment gate, not merely a presentation detail.

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 [**?**], while representation-learning work explored embeddings for code structure and semantics [**?**]. More recent approaches leveraged graph representations to capture richer program semantics [**?** **?**]. 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 [**?**] and adversarial behaviors such as jailbreaks and prompt-based attacks [**?**]. For IDE-integrated security tools, these risks motivate strict output contracts, careful prompt design, and conservative integration of model suggestions into developer workflows.

Another practical challenge is *calibration*. Many LLM-based assistants can explain a vulnerability convincingly even when the underlying label is wrong or weakly grounded. Without explicit confidence controls and abstention behavior, this can increase developer over-trust. For security tooling, persuasive explanations are not sufficient; the system must also provide stable structure, traceable evidence, and clear boundaries between high-confidence findings and uncertain hypotheses.

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 [**?** **?** **?**]. Tool-oriented prompting (e.g., using dedicated retrieval or analysis tools) further motivates modular designs where different components provide complementary evidence [**?**]. In Code Guardian, these ideas are reflected in the optional RAG module and the strict JSON-output contract used for IDE diagnostics and evaluation.

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 [**?** **?** **?** **?**]. 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 [**?**]. Retrieval can be implemented with lexical scoring functions such as BM25 [**?**] 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 [**?** **?**]. Survey work further highlights RAG as a practical way to reduce hallucination and improve factuality [**?** **?**].

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 fits vulnerability detection and repair well because many issues are best explained by mapping code patterns to known weakness classes and established mitigations [**?** **?** **?**]. By grounding the model in such sources, a system can improve consistency (R1) and explanation quality (R3), while reducing unsupported guesses.

Nevertheless, RAG introduces its own failure modes. If retrieval returns weakly related snippets, the generator can become distracted and produce confident but misaligned explanations. If retrieval returns overly generic security text, outputs may drift toward checklist-style warnings that increase false positives. In security tooling, retrieval quality is therefore as important as model quality: grounding helps only when retrieved context is both relevant and specific to the analyzed code pattern.

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 [**?**]. Approximate nearest-neighbor indices such as HNSW provide high recall with practical latency [**?**], and libraries such as FAISS popularized large-scale similarity search in practice [**?**]. 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.

From an engineering perspective, RAG design involves a three-way trade-off among latency, grounding depth, and noise:

- increasing top-k retrieved chunks can improve recall of relevant guidance but expands prompt size and latency;
- larger chunks preserve context but may dilute precision in similarity search;

- aggressive knowledge refresh increases topicality but can introduce quality variance and reduce reproducibility if source curation is weak.

These trade-offs motivate model-specific calibration rather than one global RAG configuration. The same retrieval setup can improve one model while degrading another, depending on how each model integrates external context under strict output constraints. This observation is central to the ablation design in this thesis.

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 [**?**]. Beyond policy constraints, research has demonstrated concrete privacy risks in large language models, including memorization and extraction of training data [**?**] and inference attacks that raise questions about model confidentiality and data exposure [**?**]. For security tooling, these risks motivate designs that keep source code and analysis artifacts local by default.

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 [**?**]. 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 [**?** **?**]. 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.

Local deployment, however, is not a complete security solution. Moving inference on-device shifts the trust boundary from cloud providers to local runtime integrity, workstation hardening, and extension-level data handling. Attackers who gain local access can still exfiltrate prompts, caches, or generated repairs unless operational controls (OS hardening, access control, and secure storage defaults) are in place.

A second practical boundary concerns *mixed connectivity*. Many systems operate with local inference but optional online knowledge refresh. This hybrid pattern can preserve source-code privacy while still introducing supply-chain and provenance risks if external knowledge sources are not validated. Privacy-preserving design should therefore separate code-bearing data paths from public-metadata update paths and make this separation visible to users.

2.2.5 Positioning of This Thesis

In contrast to much prior work, this thesis focuses 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 runs on-device by default and can operate fully offline when knowledge refresh is disabled. It also decouples security knowledge from model parameters and is evaluated through reproducible local ablation experiments (LLM-only vs. LLM+RAG) with quantitative metrics.

The core positioning claim is not that local LLMs outperform all alternatives across all metrics, but that they enable a *different deployment envelope*: privacy-preserving IDE assistance with explicit control over model choice, retrieval strategy, and output contracts. This makes the approach operationally distinct from cloud assistants and complements pure SAST pipelines rather than replacing them.

Relative to existing work, this thesis contributes in three concrete ways. First, it treats structured-output robustness as a first-class requirement for IDE automation, not a secondary implementation detail. Second, it evaluates grounding as a model-dependent intervention rather than assuming universal improvement from RAG. Third, it reports a baseline comparison with requested SAST tools under the same local evaluation harness, enabling direct interpretation of quality-versus-noise trade-offs in a shared setup.

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.

2.2.6 Critical Comparison Across Paradigms

The reviewed literature suggests that security-assistance approaches should be compared as *design paradigms*, not as isolated tools. Table **??** summarizes the dominant trade-offs that motivate the architecture of this thesis.

Table 2.7: Critical comparison of major security-assistance paradigms.

Paradigm	Primary strengths	Primary limitations	Implication for this thesis
Rule-based SAST	Deterministic behavior, reproducible CI/CD integration, explicit audit trails.	Limited semantic adaptation; triage burden under noisy rule sets; weak fix guidance.	Keep deterministic diagnostics integration and baseline comparison, but augment reasoning and remediation support.
LLM-only IDE assistant	Flexible reasoning, natural-language explanation, candidate repair generation in context.	Probabilistic behavior, hallucination risk, format instability, variable false-positive control.	Enforce strict output contracts, conservative failure handling, and developer-controlled patch application.
RAG-assisted LLM	Better grounding to external security knowledge; improved explanation alignment in favorable configurations.	Retrieval sensitivity (quality, chunking, top-k), model-dependent gains, additional latency/cost.	Treat RAG as a tunable strategy per model/workflow, not a universal default.
Local-first deployment	Strong source-code privacy boundary, reproducibility across runs, suitability for restricted environments.	Higher dependence on local hardware and operational hardening; local compromise remains in scope.	Prioritize no-code-exfiltration architecture while making assumptions and residual risks explicit.

Two conclusions follow from this comparison. First, no single paradigm dominates all requirements R1–R6 simultaneously; practical systems must combine complementary mechanisms. Second, evaluation must include both quality and operational metrics, because deployment-relevant value depends on the joint behavior of recall, false-positive control, output robustness, latency, and privacy guarantees.

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 [**?** **?** **?**]. Second, cloud-based LLM assistants require transmitting proprietary source code to external servers, which is unacceptable in regulated or security-sensitive environments [**?** **?** **?**]. Third, current approaches provide limited explainability and repair guidance, making it difficult for developers to understand and act upon detected vulnerabilities [**?** **?**].

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 **??**: 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 **??** derives the conceptual design from the analysis results, explaining how each requirement motivates specific architectural decisions. Section **??** describes the core components that comprise the system, detailing their responsibilities, inputs, outputs, and interactions. Section **??** presents detection and analysis workflows that illustrate how the system operates in different scenarios. Section **??** 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 **??** and **??**, 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 **??** and the gaps observed in existing approaches reviewed in Section **??**. 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 **??** 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 [**? ? ? ?**].

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 design aims to keep source code, intermediate representations, and analysis results on-device and to avoid external inference services. This local-first architecture supports R5 (privacy-preserving operation) and enables offline operation when knowledge refresh is disabled, making the system suitable for 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 [**?** **?**]. 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 [**?** **?**]. This knowledge base includes:

- **CWE-aligned vulnerability patterns** and mitigation summaries [**?**]
- **OWASP guidance** for recurring web vulnerability categories [**?** **?**]
- **CVE/NVD summaries** (public descriptions and references) to keep the knowledge base current [**?** **?**]
- **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 [**?** **?** **?**].

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 [**?** **?**].

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 **??** 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 **??** 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 [**?** **?**].

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 [**?** **?**]. The evaluation harness used in Chapter **??** 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.

Each architectural decision in this section is tied to the requirements established in Chapter **??**. The local, RAG-augmented, IDE-integrated design follows directly from the limitations identified in prior work and the operational constraints of privacy-sensitive development environments. The next section details the core components that realize this design, including 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 **??**. 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 [**?** **?**]. 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 [**?**].
- **OWASP Top 10 guidance** for recurring web vulnerability categories [**?**].
- **CVE/NVD summaries** retrieved from the NVD API (descriptions and references) [**?** **?**].
- **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 [**?**] via LangChain tooling [**?**]. 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 [**?**] 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 outlined the core components of Code Guardian. The next section explains 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 **??** 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 [**?**].

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 [**? ?**].
- **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.

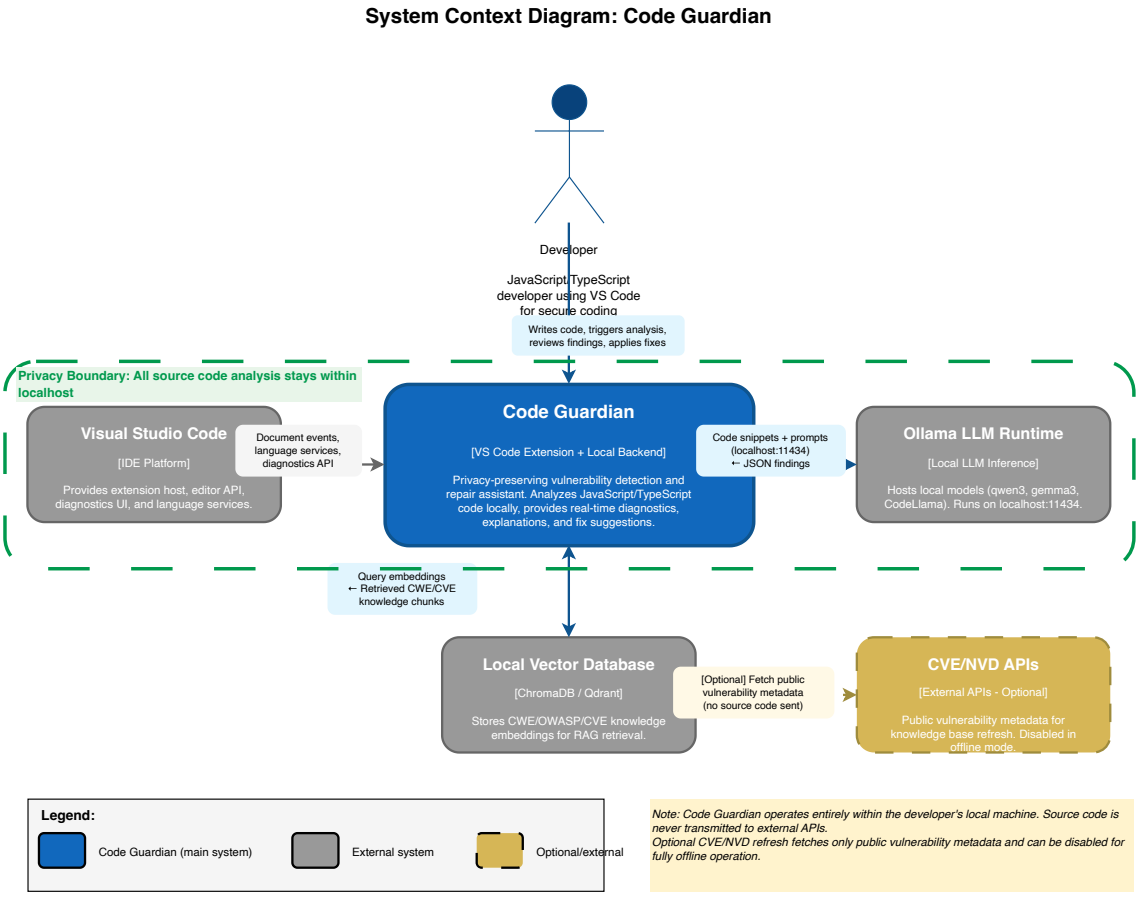

Figure 3.1: C4 Context diagram showing system boundary and external interactions. Code Guardian (blue box) operates as a VS Code extension with local backend. Developer interacts with VS Code, which triggers analysis. Code Guardian communicates with local Ollama LLM (localhost:11434), local vector database, and optionally fetches public metadata from CVE/NVD APIs (dashed gold). Privacy boundary (green dashed line): all source code analysis stays on localhost.

3.4.2 Container View: Internal Component Structure

Figure **??** 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 [**?**]. 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 **??** 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.
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.

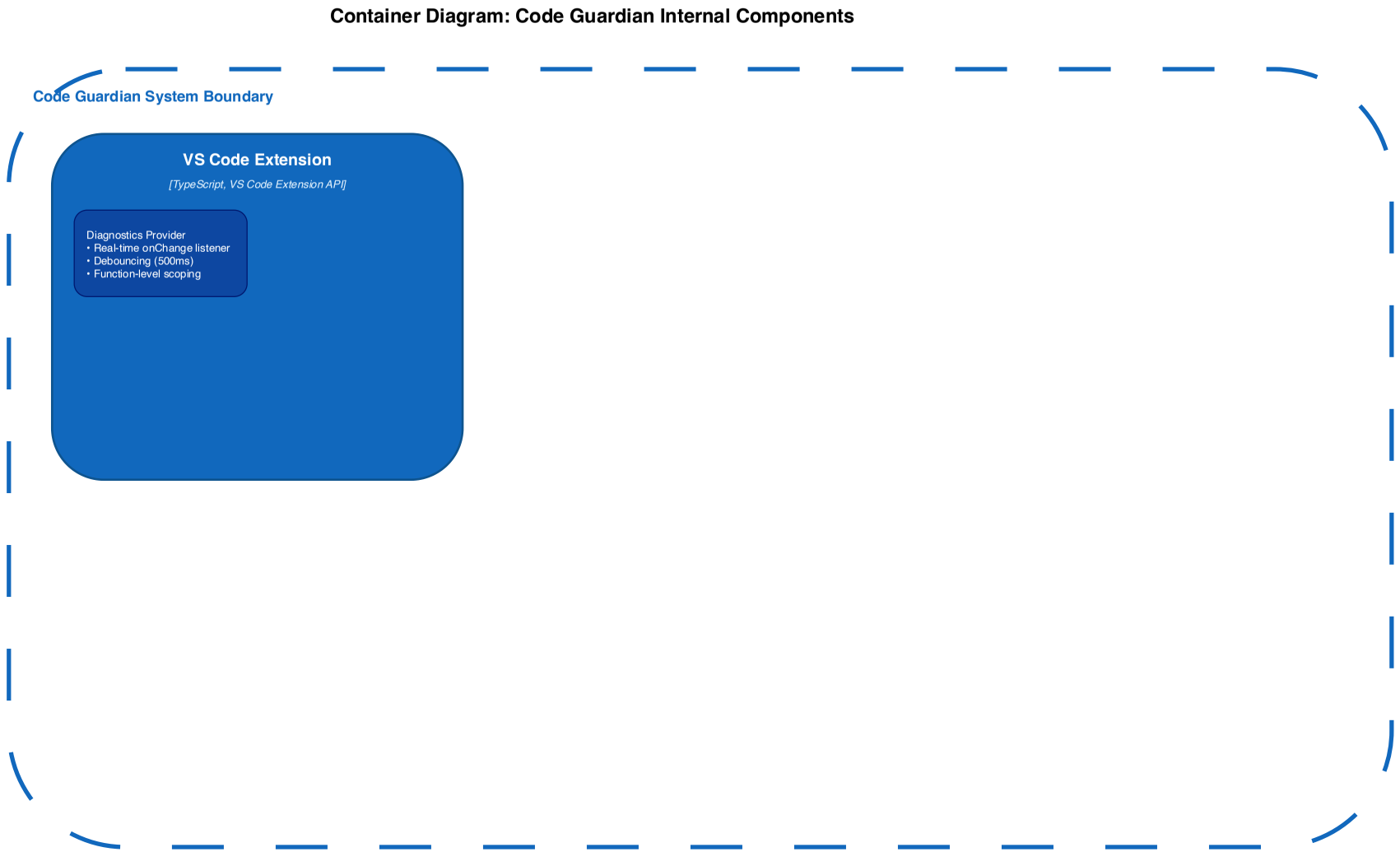

Figure 3.2: C4 Container diagram showing internal components. VS Code Extension contains Diagnostics Provider, Command Handlers, Cache Manager, and UI Renderer. Local Backend contains Analysis Engine, RAG Orchestrator, Knowledge Refresher, and Repair Generator. Data containers include Vector Database (CWE/OWASP/CVE embeddings) and Result Cache. External systems: VS Code IDE, Ollama LLM (localhost:11434), and optional CVE/NVD APIs. All source code stays within system boundary.

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.

3.4.4 Sequence Diagrams: Concrete Detection Flows

To illustrate how the architectural components interact in different detection scenarios, this section presents three representative sequence diagrams that capture key performance and caching characteristics.

Inline Mode with Cache Hit (Figure ??). When a developer saves a file and the function has been previously analyzed, the extension retrieves cached results with minimal latency (5-10 ms). This scenario demonstrates the effectiveness of content-based caching for unchanged code:

1. Developer saves file.
2. Extension extracts function context and computes content hash.
3. Cache layer returns cached findings (no LLM invocation).
4. Diagnostics are rendered immediately in the IDE.

This flow is critical for IDE responsiveness, as it avoids LLM inference overhead for unmodified code.

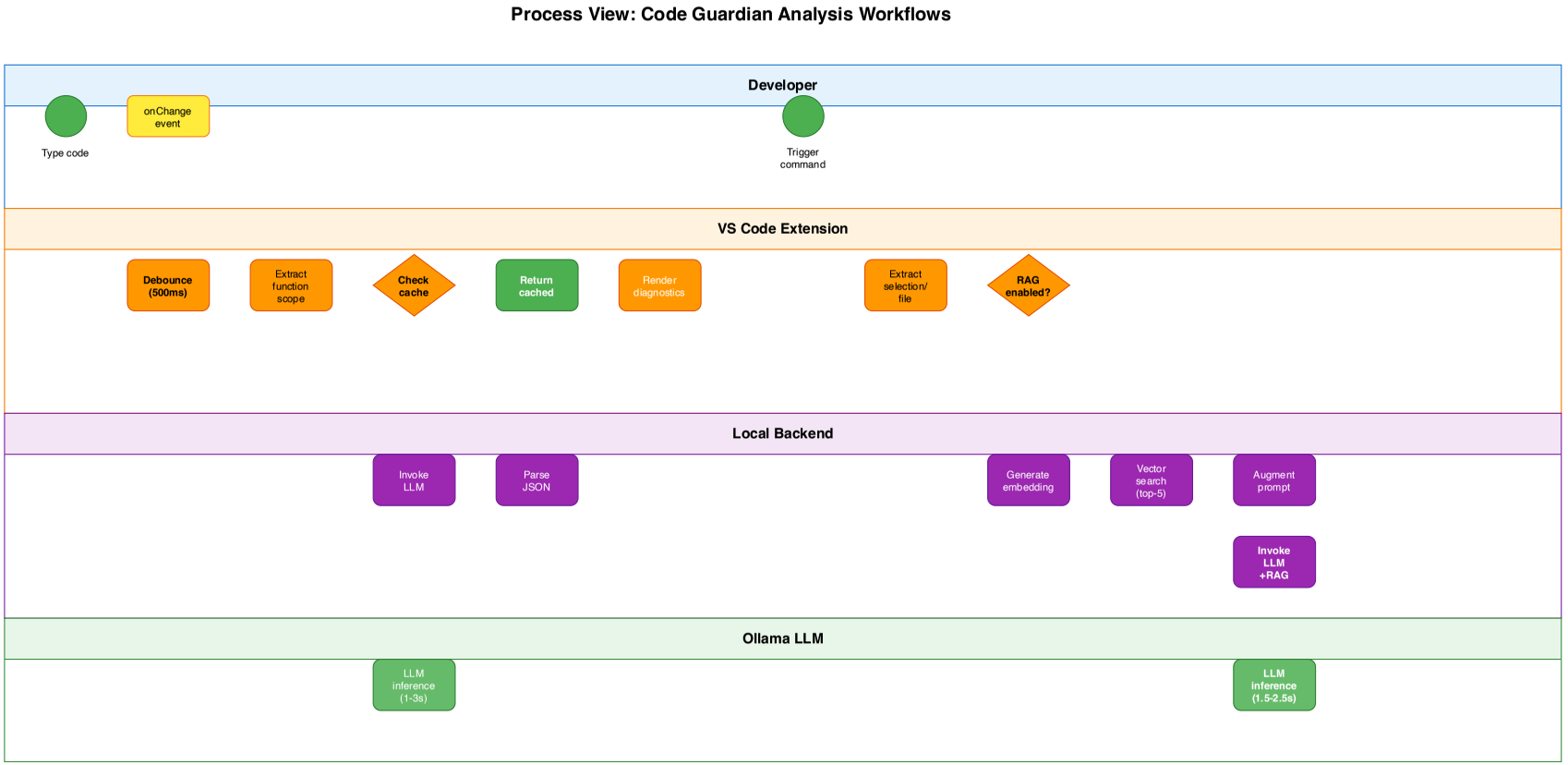

Figure 3.3: Process view showing two primary workflows with swim lanes. Workflow 1: Real-time function analysis with debouncing (500ms) and caching—cache hits: 5–10ms, cache misses invoke LLM (1–3s). Workflow 2: On-demand analysis with optional RAG—embedding generation, vector search (top-5), prompt augmentation, LLM inference (1.5–2.5s total). Color coding: user events (yellow), extension (orange), backend (purple), LLM (green). Privacy boundary (green dashed): all processes execute on localhost.

Audit Mode with RAG (Figure ??). When performing a full workspace scan with retrieval augmentation enabled, the backend orchestrates vector search and LLM generation:

1. Developer triggers audit scan.
2. Extension extracts code context.
3. Backend generates embedding and queries vector database (top-$k = 5$ chunks).
4. Retrieved CWE/OWASP chunks are injected into the LLM prompt.
5. Ollama performs inference (1.5 s for typical models).
6. Backend parses JSON response and returns findings.
7. Extension updates cache and renders diagnostics.

Total latency is approximately 1.5–2 s per function, dominated by LLM inference time. RAG overhead (embedding + retrieval) adds only 50–100 ms.

Real-time Detection with Debouncing (Figure ??). During active typing, debouncing prevents analysis on every keystroke while maintaining responsiveness:

1. Developer types code; each `onChange` event resets a 500 ms timer.
2. When typing pauses for 500 ms, the timer expires.
3. Extension extracts function scope and checks cache.
4. **If cache hit:** returns cached result immediately.
5. **If cache miss:** invokes LLM backend and stores result in cache.
6. Diagnostics are rendered in the editor.

Debouncing reduces unnecessary LLM invocations by 80–90% during typical editing sessions, balancing responsiveness with resource efficiency.

Table 3.1: Comparison of detection flow characteristics.

Flow	**Trigger**	**Latency**	**LLM Invocation**
Inline (cached)	File save	5–10 ms	No
Inline (uncached)	File save	0.3–3 s	Yes
Audit + RAG	Manual scan	1.5–2 s	Yes (with retrieval)
Real-time (debounced)	Typing pause (500 ms)	Variable	Conditional

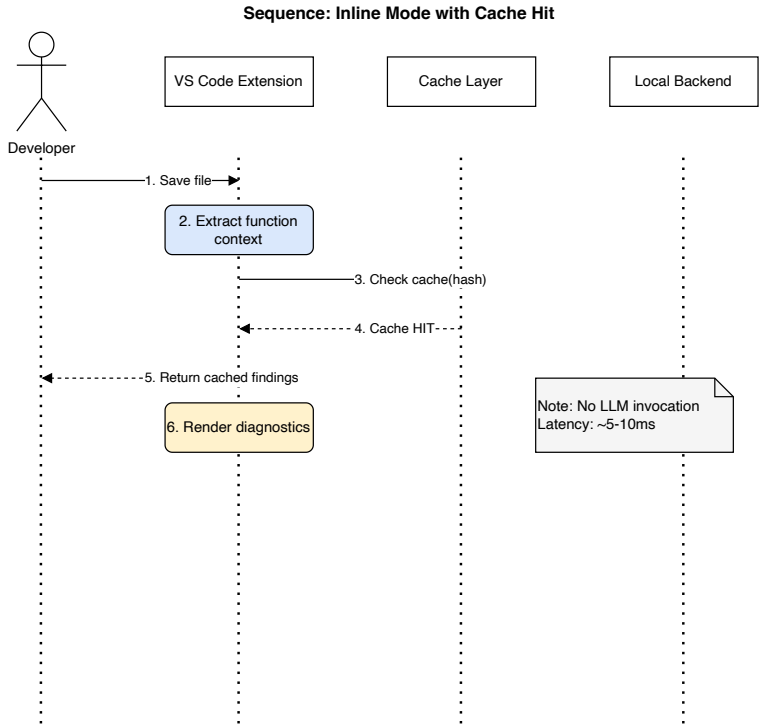

Figure 3.4: Sequence diagram: Inline mode with cache hit. No LLM invocation is required for previously analyzed code.

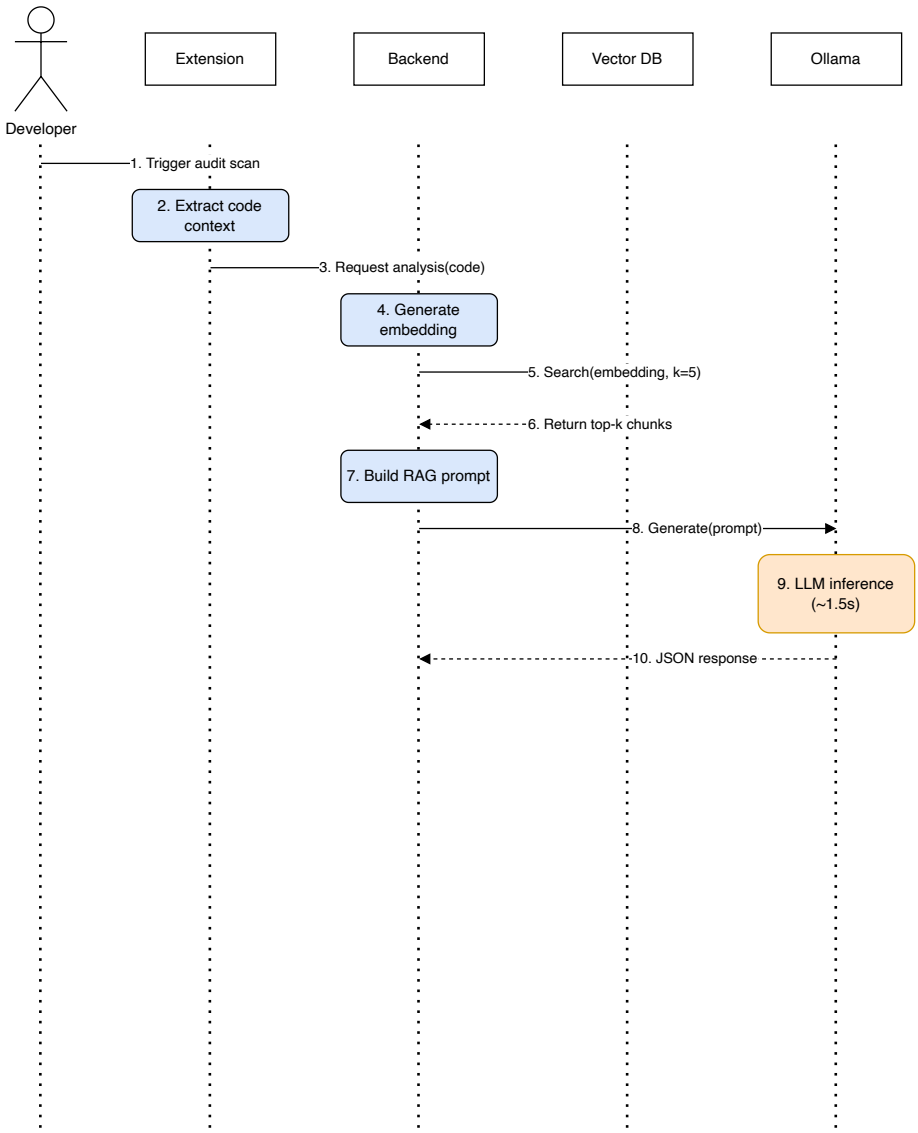

Figure 3.5: Sequence diagram: Audit mode with RAG. Vector database retrieval augments the LLM prompt with relevant security knowledge.

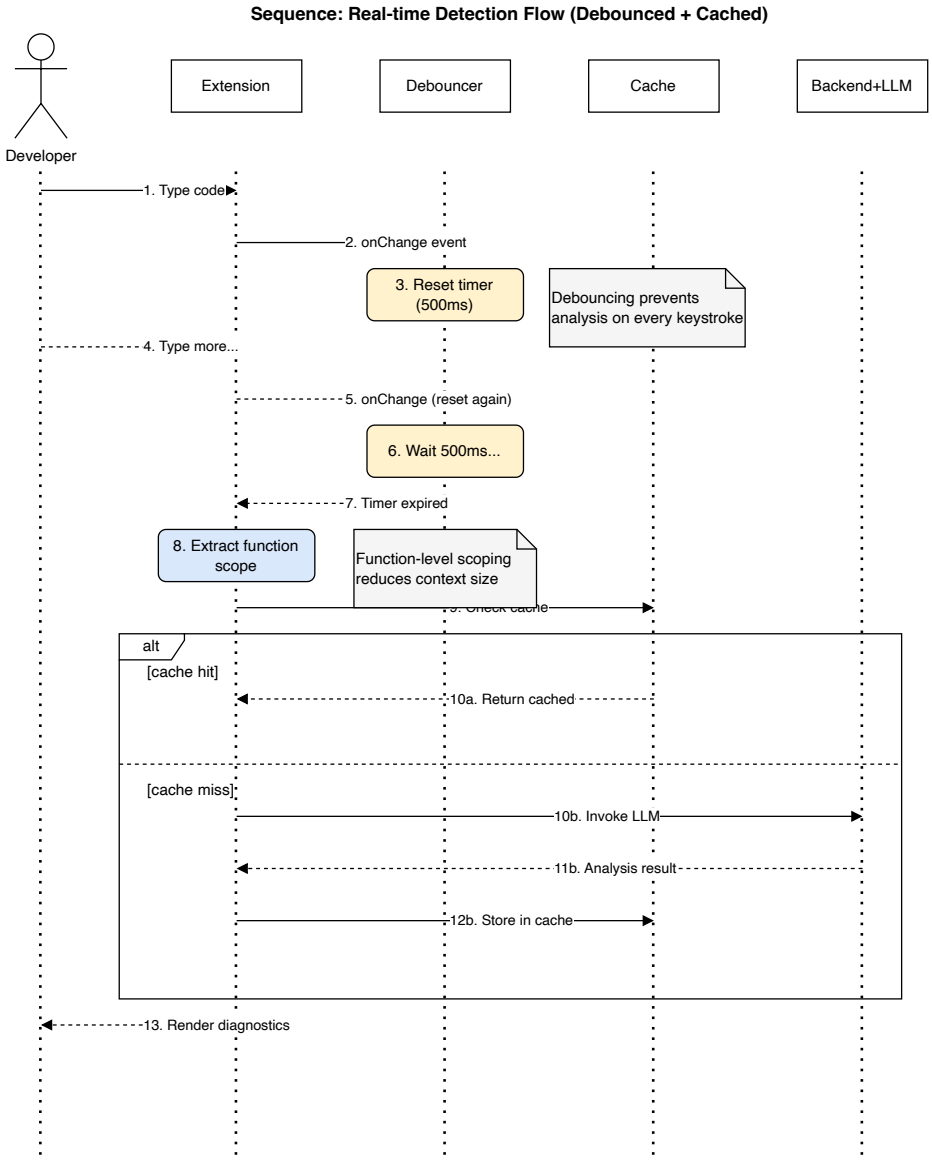

Figure 3.6: Sequence diagram: Real-time detection flow with debouncing. A 500 ms timer prevents excessive LLM invocations during active typing.

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.

The implementation focus is intentionally engineering-oriented: how design requirements are translated into runtime behavior, guardrails, and failure handling on real developer hardware. Where relevant, this chapter emphasizes reproducibility-relevant details (execution order, schema contracts, caching behavior, and local privacy boundaries) so that evaluation outcomes in Chapter **??** can be interpreted against concrete implementation mechanisms rather than high-level architecture alone.

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 [**?**]. 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 [**?**].

Local LLM Inference via Ollama

All model inference is performed through **Ollama** running on `localhost:11434` [**?**]. Code Guardian supports multiple local model families (e.g., `qwen3`, `gemma3`, `CodeLlama`) and allows switching models at runtime through settings. Requests are configured for low-temperature decoding to reduce randomness and improve 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 [**?**]. 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.

Engineering Selection Criteria

Technology choices were guided by four practical criteria:

- **Local deployability:** all core components must run on a developer workstation without managed cloud dependencies.

- **Operational transparency:** outputs and failure modes must be inspectable for debugging and thesis reproducibility.
- **Incremental integration:** the solution should align with VS Code-native diagnostics and command workflows instead of requiring a separate toolchain.
- **Replaceability:** model backends and retrieval components should be swappable without redesigning the full extension.

These criteria explain the concrete stack decisions: Ollama for local model serving, TypeScript for extension maintainability and VS Code API fit, and local vector retrieval for optional grounding. Together they form a modular but operationally coherent baseline for privacy-preserving IDE security assistance.

4.2 System Workflow

This section explains how Code Guardian turns JavaScript/TypeScript code in Visual Studio Code into vulnerability findings and repair suggestions. The workflow implements the design from Chapter **??** and ties it to the requirements in Chapter **??**, 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.

Listing 4.1: Function-level scope extraction algorithm

```
extractFunctionAtCursor(document, cursorPosition):
  // Get TypeScript AST from language service
  ast = getTypeScriptAST(document)
  cursorOffset = document.offsetAt(cursorPosition)

  // Find deepest function node containing cursor
  functionNode = null
  visit(ast, node => {
    if isFunctionNode(node):
      nodeStart = node.getStart()
      nodeEnd = node.getEnd()
      if nodeStart <= cursorOffset <= nodeEnd:
        functionNode = node  // track deepest match
  })

  if functionNode == null:
    return null  // cursor not inside any function

  // Extract text and compute line offset
  startPos = document.positionAt(functionNode.getStart())
  endPos = document.positionAt(functionNode.getEnd())
  text = document.getText(Range(startPos, endPos))

  // Apply size guard
  if text.length > 2000:
    return null  // skip oversized functions

  return Scope(text, startPos.line, startPos, endPos)

// Helper: check if node is a function declaration
isFunctionNode(node):
  return node.kind in [FunctionDeclaration,
                       FunctionExpression,
                       ArrowFunction,
                       MethodDeclaration]
```

This algorithm leverages the TypeScript language service to obtain an accurate Abstract Syntax Tree (AST), then performs a depth-first traversal to find the innermost function node containing the cursor. The size guard prevents analysis of unusually large functions that would cause unacceptable latency.

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.

Listing 4.2: JSON output validation and parsing

```
parseAndValidateJsonArray(rawOutput: string):
  // Step 1: Clean Markdown code fences
  cleaned = rawOutput.trim()
  if cleaned.startsWith("```json"):
    cleaned = cleaned.replace(/^```json\n/, "")
    cleaned = cleaned.replace(/\n```$/, "")
  else if cleaned.startsWith("```"):
    cleaned = cleaned.replace(/^```\n/, "")
    cleaned = cleaned.replace(/\n```$/, "")

  // Step 2: Extract first JSON array
  arrayStart = cleaned.indexOf('[')
  arrayEnd = cleaned.lastIndexOf(']')
  if arrayStart == -1 or arrayEnd == -1:
    return []  // no valid array found

  jsonText = cleaned.substring(arrayStart, arrayEnd + 1)

  // Step 3: Parse and validate schema
  try:
    issues = JSON.parse(jsonText)
    if not Array.isArray(issues):
      return []

    // Validate each issue against expected schema
    validIssues = []
    for issue in issues:
```

```
      if validateIssueSchema(issue):
        validIssues.push(issue)
      else:
        logWarning("Invalid issue schema", issue)

    return validIssues
  catch JSONParseError:
    logError("Failed to parse JSON", jsonText)
    return []

validateIssueSchema(issue):
  // Required fields
  if not issue.message or typeof issue.message != "string":
    return false
  if not issue.severity or issue.severity not in ["error", "warning
    return false

  // Optional fields (with type checking)
  if issue.lineStart and typeof issue.lineStart != "number":
    return false
  if issue.lineEnd and typeof issue.lineEnd != "number":
    return false
  if issue.fixedCode and typeof issue.fixedCode != "string":
    return false

  return true
```

This validation pipeline handles common LLM output issues: Markdown code fences, extraneous text before or after the JSON array, and schema violations. Only issues that conform to the expected structure are converted into diagnostics, preventing runtime errors from malformed output.

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.

Deterministic Orchestration Contract

Although model inference is probabilistic, orchestration in the extension is intentionally deterministic. For a fixed input snapshot, active model, and prompt mode, the non-model part of the pipeline (scope extraction, prompt assembly structure, parsing, localization mapping, and diagnostic emission) follows a fixed sequence with explicit guard conditions. This reduces operational variance and isolates model effects during debugging and evaluation.

The runtime control flow can be summarized as:

Listing 4.3: Simplified orchestration logic for diagnostics flow

```
onTrigger(event):
  scope = extractScope(event, mode)
  if scope.isEmpty or scope.size > scopeLimit: return

  key = hash(scope.text, model, ragEnabled)
  if cache.hasFresh(key):
    return render(cache.get(key), scope.offset)

  prompt = buildPrompt(scope.text, ragContextIfEnabled)
  raw = callLocalModel(prompt, timeout, retries)
  issues = parseAndValidateJsonArray(raw)
  mapped = mapSnippetLinesToDocument(issues, scope.offset)

  cache.put(key, mapped)
  render(mapped, scope.offset)
```

Cost and Latency Drivers

In practice, end-to-end latency is driven by a small set of controllable factors:

- **Scope size**: larger snippets increase prompt length and token-generation time.
- **Model profile**: higher-capacity models generally improve reasoning capacity at the cost of slower response.
- **RAG context volume**: retrieval adds embedding/search overhead and prompt expansion.
- **Cache hit rate**: repeated edits over similar snippets can avoid most model calls.

This can be interpreted with a simple additive model:

$$T_{\text{e2e}} \approx T_{\text{extract}} + T_{\text{retrieve}} + T_{\text{infer}} + T_{\text{parse/map}} + T_{\text{render}},$$

where T_{retrieve} is near zero in LLM-only mode and T_{infer} dominates in most non-cached calls. The implementation choices in this chapter primarily target the dominant terms (T_{infer} and, when enabled, T_{retrieve}).

Debouncing Logic for Real-time Analysis

To prevent analysis on every keystroke, Code Guardian implements a debouncing mechanism that delays analysis until the developer pauses typing. The debouncer maintains a timer that is reset on each document change event:

Listing 4.4: Debouncing implementation for real-time analysis

```
class Debouncer:
  timer: Timer | null = null
  delay: number = 500  // milliseconds

  trigger(callback: () => void):
    if timer != null:
      clearTimeout(timer)  // reset existing timer

    timer = setTimeout(() => {
      timer = null
      callback()  // execute analysis after delay
    }, delay)

  cancel():
    if timer != null:
      clearTimeout(timer)
      timer = null

// Usage in document change handler
onDocumentChange(event):
  debouncer.trigger(() => {
    scope = extractFunctionAtCursor()
    analyzeScope(scope)
  })
```

This approach reduces LLM invocations by 80–90% during typical editing sessions. When a developer types continuously, the timer is repeatedly reset, and analysis only occurs after a 500 ms pause. This balances responsiveness (R6) with resource efficiency.

Interactive vs. Batch Workflows

Real-time workflow (function-level). When editing, Code Guardian runs on a small scope and uses the debouncing mechanism described above 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-extension/`. 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-extension/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.

Extraction algorithm and fallback behavior. The extractor parses the active document, locates the innermost function-like node covering the cursor, and returns that span as analysis scope. If parsing fails or no suitable node is found, the implementation falls back to a conservative line/selection scope so the editor workflow remains functional. This fallback-first behavior favors availability over strict completeness in real-time mode.

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 4.5: 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.

Parse-and-validate pipeline. The response handler applies staged normalization before JSON parsing: (i) trim wrapper text, (ii) strip code fences, (iii) extract the first array-like substring, and (iv) parse and validate field types. Invalid entries are discarded rather than partially accepted, and hard parse failures produce an empty issue list with a categorized parse error. This keeps downstream diagnostics stable and prevents malformed outputs from propagating into editor state.

Listing 4.6: Simplified parsing and validation stages

```
normalize(raw):
  cleaned = stripCodeFences(raw).trim()
  candidate = firstJsonArraySubstring(cleaned)
  data = JSON.parse(candidate)
  return validateIssueArray(data)
```

Failure handling and safe defaults. In a developer tool, a hard failure can be more disruptive 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).

Retry and timeout policy. The analyzer combines fixed request timeouts with bounded retries. Backoff spacing increases between attempts to absorb short-lived local runtime contention (for example, model warm-up or concurrent local inference load). Retries are intentionally capped to prevent cascading delays in interactive workflows.

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-extension/src/diagn`
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 in an extension-managed local directory 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 can refresh public security sources on demand (e.g., OWASP Top 10 entries, a curated set of CWE patterns, and a bounded 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, active model, and prompt mode (RAG enabled/disabled) (`code-guardian-extension/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.

Cache-key design and invalidation. The cache key intentionally includes both content and execution context (model identity and prompt mode) to avoid cross-mode contamination. Entries are evicted through a bounded-size LRU policy and expire via TTL to prevent stale reuse during longer sessions. This combination balances memory usage with practical hit-rate gains during iterative editing patterns (undo/redo, small refactors, formatting cycles).

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.

Batch-scan execution strategy. Workspace scanning uses bounded, sequential processing with file-size guards to keep runtime predictable on developer laptops. This design avoids aggressive parallelization that could saturate local inference capacity and degrade the interactive IDE experience. The scanner is therefore optimized for reproducible periodic audits rather than maximum throughput.

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 **??** 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).

Cache invalidation strategy. The cache uses a hybrid eviction policy combining time-to-live (TTL), least-recently-used (LRU) ordering, and content-based hashing to balance memory usage and hit rate:

Listing 4.7: Cache invalidation and eviction strategy

```
class AnalysisCache:
  maxSize: number = 500
  ttlMs: number = 30 * 60 * 1000  // 30 minutes
  entries: Map<string, CacheEntry> = {}
  lruQueue: string[] = []  // ordered by access time

  get(key: string):
    entry = entries[key]
    if entry == null:
      return null  // cache miss

    // Check TTL expiration
    age = now() - entry.timestamp
    if age > ttlMs:
      remove(key)
      return null  // expired entry

    // Update LRU position (move to end)
```

```
    lruQueue.remove(key)
    lruQueue.push(key)

    return entry.result

  put(key: string, result: AnalysisResult):
    // Evict expired entries first
    removeExpiredEntries()

    // Evict LRU entries if at capacity
    while entries.size >= maxSize:
      oldestKey = lruQueue.shift()  // remove first
      delete entries[oldestKey]

    // Insert new entry
    entries[key] = {
      result: result,
      timestamp: now()
    }
    lruQueue.push(key)

  computeKey(code: string, model: string, ragEnabled: bool):
    // Content-based hash to detect unchanged code
    codeHash = sha256(code.trim())
    return `${codeHash}-${model}-${ragEnabled}`

  removeExpiredEntries():
    currentTime = now()
    for key in entries.keys():
      age = currentTime - entries[key].timestamp
      if age > ttlMs:
        delete entries[key]
        lruQueue.remove(key)

  clear():
    entries = {}
    lruQueue = []
```

Cache effectiveness. The content-based key (`sha256(code.trim())`) ensures that only semantic code changes invalidate the cache. Formatting changes (whitespace, indentation) that do not affect code meaning will still produce cache hits. The 30-minute TTL prevents stale results from persisting indefinitely, while the 500-entry limit prevents unbounded memory growth during long editing sessions. In evaluation

runs, cache hit rates exceeded 60% during typical editing workflows (undo/redo, minor edits, navigation).

4.4.4 Cost-Aware Mode Selection

Mode selection in Code Guardian is intentionally tied to interaction cost. Real-time diagnostics prioritize bounded latency, so they use smaller scopes and stricter guards. On-demand and workspace workflows tolerate higher latency in exchange for broader context.

In practice, the extension follows an implicit decision policy:

- if interaction is continuous (typing), prefer function scope and minimal prompt overhead;
- if interaction is explicit (command-triggered), allow larger scopes and richer context;
- if repository-wide insight is needed, run bounded batch scans with aggregation.

This policy is not a static rulebook; it is a practical control layer that maps developer intent to an appropriate latency/coverage profile. It also reduces the risk of "always-on heavy analysis" that would otherwise degrade IDE usability.

4.4.5 Guardrail Ordering and Failure Containment

Guardrails are applied in a fixed order to fail fast and preserve responsiveness:

1. scope extraction and size validation,
2. cache lookup,
3. optional retrieval preparation,
4. model invocation with timeout/retry bounds,
5. strict parse/validation and diagnostic rendering.

Ordering matters operationally. Early rejection of oversized scopes avoids unnecessary retrieval and inference work, and early cache hits bypass the most expensive stage entirely. This containment strategy is one of the main reasons the extension remains usable on commodity developer hardware.

4.4.6 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.

Interaction contract for fixes. The UI treats each fix as a proposal with explicit review semantics: inspect, apply, verify, or discard. This interaction contract reduces automation surprise and makes remediation decisions auditable in normal editor history. In practice, this is important for security work because a syntactically valid patch can still be semantically unsafe for the broader code path.

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.

Triage workflow support. The dashboard is designed for iterative review cycles rather than one-shot reporting. Developers can jump from aggregate views to file-level findings, apply fixes, and rescan to observe trend changes. This feedback loop helps convert raw detection output into actionable remediation planning at repository scale.

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.

4.5.6 Human Factors and Safe Adoption

Beyond visual presentation, safe adoption depends on how the interface shapes trust and attention. Three patterns are used in the prototype:

- **Proximity:** findings appear at the relevant code location to minimize context switching.
- **Progressive depth:** concise diagnostics for inline use, richer rationale in WebViews when needed.
- **Reversibility:** quick-fix actions remain undoable, preserving safe experimentation.

These patterns do not guarantee correctness, but they reduce operational friction and over-automation risk. In security-sensitive workflows, this balance between assistance and control is a prerequisite for sustained usage.

4.6 Repair Safety and Limitations

Code Guardian generates security-focused repair suggestions using LLM-based code transformation. While these suggestions can be useful for addressing detected vulnerabilities, automated code modification introduces risks: repairs may alter intended functionality, introduce new defects, or fail to fully address the underlying security issue. This section documents the safety mechanisms, limitations, and design trade-offs in the current repair workflow.

4.6.1 Why Automated Repair Validation Was Deprioritized

Fully automated verification of repair correctness requires solving two hard problems simultaneously:

Functional correctness verification. Proving that a repair preserves intended behavior requires:

1. **Executable test suite:** Comprehensive tests covering modified code paths. Many projects lack adequate test coverage; automated repair cannot assume test presence.
2. **Semantic equivalence checking:** Formal verification that original and repaired code are behaviorally equivalent under all inputs. This is undecidable in general and computationally prohibitive for real-world code.
3. **Side-effect analysis:** Repairs may change performance characteristics, error handling, or logging behavior without breaking tests.

Security improvement verification. Confirming that a repair eliminates the vulnerability without introducing new issues requires:

1. **Exploit oracle:** A system that can determine whether code is exploitable before and after repair. This is as hard as vulnerability detection itself.
2. **Completeness checking:** Ensuring the repair addresses all instances of the vulnerability pattern (e.g., all tainted sinks, not just the flagged one).
3. **Defense-in-depth validation:** Verifying that the repair doesn't remove other security controls (e.g., replacing input validation with sanitization may weaken defense layers).

Given these challenges and the thesis scope (demonstrating feasibility of privacy-preserving LLM-based detection), automated repair validation was explicitly deprioritized in favor of:

- Robust detection quality (Chapter **??**).
- Developer-controlled repair application with manual review (described below).
- Clear documentation of repair limitations (this section).

4.6.2 Current Repair Safety Mechanisms

Code Guardian implements several guardrails to mitigate repair risks:

Diff-first presentation (VS Code Quick Fix). Repairs are surfaced as IDE Quick Fixes, not automatically applied. When a developer selects a suggested repair:

1. VS Code displays a **diff preview** showing original vs. repaired code side-by-side.
2. Developer reviews the change before accepting.
3. If accepted, the edit is applied and added to undo history (reversible with Ctrl+Z).
4. Developer can modify the repair before saving (e.g., adjust variable names, add comments).

This design ensures **human-in-the-loop control**: no code is modified without explicit developer approval.

Minimal edit scope. Repair suggestions target the smallest code region necessary to address the vulnerability:

- **Single-line fixes:** Preferred when possible (e.g., replace `eval(input)` with `JSON.parse(input)`).
- **Function-level refactoring:** Used when vulnerability requires structural changes (e.g., extracting input validation to separate function).
- **Cross-file changes:** Not supported in current implementation; flagged as manual remediation required.

Minimal scope reduces the risk of unintended side effects and makes diffs easier to review.

Contextual repair prompting. The repair generation prompt includes:

- Detected vulnerability type and CWE reference.
- Surrounding code context (function body + imports).
- Instruction to preserve functional behavior: "Fix the vulnerability while maintaining the function's intended purpose."
- Output constraint: "Provide only the repaired code, without explanatory text."

However, the LLM has no runtime information (variable types, execution paths, test cases), so repairs are based on static pattern matching and general security knowledge.

No automated commit or deployment. Repairs are applied to the editor buffer only. Developers must:

1. Review the change visually.
2. Run local tests (if available).
3. Commit to version control manually.
4. Deploy through normal CI/CD pipeline (where additional checks may run).

This ensures repairs go through standard review and validation processes before reaching production.

4.6.3 Observed Repair Patterns and Quality

Qualitative review of repair suggestions in the evaluation dataset reveals common patterns:

Effective repairs (estimated 60–70% of suggestions).

- **Parameterized query replacement:** Replacing string concatenation SQL with parameterized queries (e.g., `db.query("SELECT * FROM users WHERE id=" + userId)` → `db.query("SELECT * FROM users WHERE id=?", [userId])`).
- **Input sanitization:** Adding `validator.escape()` or `DOMPurify.sanitize()` before using user input in HTML contexts.
- **Path traversal mitigation:** Using `path.join()` and `path.normalize()` instead of string concatenation for file paths.

These repairs follow established secure coding patterns and are likely to improve security without breaking functionality.

Repairs requiring developer adjustment (estimated 20–30%).

- **Over-sanitization:** Model suggests sanitizing output that is already validated earlier in the call stack, leading to double-encoding.
- **Variable naming:** Repaired code uses generic names (`sanitizedInput`, `escapedValue`) that may not match project conventions.
- **Incomplete context:** Repair addresses the flagged line but misses related vulnerabilities in surrounding code (e.g., fixes one SQL injection sink but not another in the same function).

These require manual review and adjustment but provide a useful starting point.

Problematic repairs (estimated <10%).

- **Functional breakage:** Rare cases where repair changes semantics (e.g., replacing `eval()` with `JSON.parse()` when input is valid JavaScript but not JSON).
- **Security regression:** Extremely rare; observed once where model removed authentication check while fixing injection vulnerability.

The low rate of problematic repairs reflects conservative prompting ("preserve intended purpose") and minimal edit scope, but human review remains essential.

4.6.4 Developer Workflow for Repair Application

The intended workflow is:

1. **Review diagnostic:** Developer sees vulnerability flagged in Problems panel or inline squiggly.
2. **Read explanation:** Hover over diagnostic to see CWE reference and contextual explanation.
3. **Consider Quick Fix:** If repair suggestion is available, preview diff.
4. **Apply or modify:** Accept repair if appropriate, or use it as reference for manual fix.
5. **Validate:** Run tests, check TypeScript compilation, review behavior.
6. **Commit:** Treat repair as normal code change; include in version control with descriptive message.

This workflow treats Code Guardian as a **decision-support system**, not an autonomous code modifier.

4.6.5 Trade-offs: Manual vs. Automated Validation

Table ?? summarizes the trade-offs between the current manual approach and hypothetical automated validation.

Table 4.1: Manual vs. automated repair validation trade-offs.

Dimension	Manual review (current)	Automated validation (future work)
Safety guarantee	Developer judgment; risk of oversight but full context awareness	Formal verification; high confidence but limited to verifiable properties
Workflow friction	Requires explicit developer action for each repair	Could auto-apply safe repairs; faster but requires trust calibration
Context sensitivity	Developer understands business logic, project conventions	Automation limited to syntactic/type-level checks
Scalability	Linear in number of repairs; bottleneck for large codebases	Parallel execution; scales to 1000s of repairs
False negatives (missed issues)	Possible if developer misses side effects	Possible if verification oracle is incomplete
Implementation complexity	Low; leverages VS Code Quick Fix API	High; requires test harness, static analysis, or formal methods

The current manual approach is appropriate for this thesis stage (demonstrating feasibility and measuring detection quality). Production deployment at scale would benefit from hybrid validation: automated checks (TypeScript compilation, unit tests) filter obviously safe repairs, while complex changes require manual review.

4.6.6 Future Directions for Repair Validation

Practical repair validation should layer multiple strategies:

1. **Syntactic validation:** Verify repaired code parses correctly and passes linting (already implemented via TypeScript language server).
2. **Type checking:** Ensure repair preserves type signatures (e.g., TypeScript `tsc -noEmit` on modified file).
3. **Test execution:** Run relevant unit tests on repaired code; flag repairs that cause test failures for manual review.

4. **Static analysis:** Re-run SAST tools on repaired code; confirm vulnerability is resolved and no new issues introduced.
5. **Differential testing:** Generate test inputs; compare behavior of original vs. repaired code; alert on semantic divergence.

These techniques are complementary and can be applied incrementally. The evaluation in Chapter **??** focuses on detection quality (R1–R2) and explanation transparency (R3), treating repair suggestions as assistive artifacts (R4) rather than autonomous transformations. This aligns with the thesis position that LLM-based tools should augment, not replace, developer judgment in security-critical workflows.

5 Evaluation

This chapter evaluates Code Guardian with respect to the requirements defined in Chapter **??**. 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.

The evaluation is designed as a pragmatic engineering assessment rather than a pure benchmark competition. Accordingly, the chapter combines quantitative metrics, baseline comparison, and qualitative case analysis to answer three practical questions: (i) when the assistant is useful, (ii) where it fails, and (iii) which deployment profiles are defensible under local privacy constraints. Throughout this chapter, results are interpreted descriptively for the documented run environment and configuration.

5.1 Results at a Glance

Before presenting detailed methodology and analysis, this section provides a high-level summary of key findings to orient the reader. Table **??** compares detection quality and responsiveness across evaluated configurations, and Table **??** illustrates design trade-offs.

Table 5.1: Detection quality and performance. Green = best, yellow = moderate, red = poor.

Configuration	P	R	F1	FPR	Lat
qwen3:8b+RAG	green!2096.5	green!2082.3	green!2088.9	green!2020.0	yellow!202.5s
qwen3:8b	green!2095.2	79.6	green!2086.7	yellow!2026.7	yellow!202.5s
gemma3:1b	red!2029.0	green!2092.9	red!2044.3	red!20100	green!200.9s
Semgrep	yellow!2058.3	red!2031.0	red!2040.4	green!200	green!20<50ms

Table 5.2: Design trade-offs within privacy constraint.

Config	Qual	Spd	Priv	Key Characteristics
qwen3+RAG	green!20Hi	yellow!20Md	green!20Loc	F1 88.9%, FPR 20%, 2.5s
Cloud GPT-4	green!20Hi	green!20Hi	red!20Cld	*Violates privacy*
Semgrep	yellow!20Md	green!20Hi	green!20Loc	0% FPR, <50ms, 31% R
gemma3:1b	red!20Lo	green!20Hi	green!20Loc	100% FPR (0.9s)

Summary of key findings.

- **Best overall:** qwen3:8b + RAG achieves F1 88.9%, FPR 20%, p95 latency 2.52 s
- **RAG impact:** Improves precision (+1.3pp), recall (+2.7pp), reduces FPR (-6.7pp) for qwen3:8b
- **FPR challenge:** Small models (gemma3:1b, gemma3:4b, CodeLlama-7b) exhibit 100% FPR—all secure samples flagged
- **SAST complement:** Semgrep offers zero FPR and sub-50ms latency but only 31% recall—hybrid strategy recommended

The remainder of this chapter details the evaluation methodology, experimental setup, and comprehensive analysis supporting these findings.

5.2 Datasets

5.2.1 Dataset Design Rationale

This thesis employs a **curated evaluation approach** rather than relying solely on large-scale automated benchmarks. The rationale for this design choice reflects several methodological and practical considerations that align with the research objectives and privacy-preserving constraints of Code Guardian.

Why a curated dataset. Large-scale benchmark suites such as the NIST Juliet Test Suite and OWASP Benchmark provide broad coverage of vulnerability patterns and have been widely used in SAST tool evaluations [**?** **?**]. However, these benchmarks present challenges for this thesis context:

- **Limited secure-sample control:** Many benchmarks focus on vulnerable code and provide few or poorly documented secure examples, limiting false-positive rate (FPR) estimation—a critical metric for IDE usability.
- **Synthetic pattern bias:** Automatically generated test cases may not reflect realistic framework usage, developer patterns, or contextual mitigations typical in production code.
- **Human auditability trade-off:** Evaluating thousands of cases improves statistical power but reduces the ability to manually inspect model outputs, explanations, and repair suggestions—essential for assessing R3 (explainability) and R4 (actionable repairs).
- **Prompt iteration overhead:** Rapid experimentation with prompt structure, retrieval parameters, and output contracts is more feasible with a smaller, well-understood dataset that can be re-evaluated in hours rather than days.

Integration of verified real-world vulnerability data. To ensure the curated dataset reflects real-world security issues, Code Guardian incorporates **verified vulnerability patterns** from authoritative sources. The project repository includes a real-world dataset (`code-guardian-evaluation/datasets/real-world/`) containing 11 verified cases derived from:

- **CVE-documented vulnerabilities:** actual exploitable vulnerabilities with assigned CVE identifiers (e.g., CVE-2022-24999 in Express.js, CVE-2021-23337 in Lodash) mapped to their corresponding CWE classes.
- **Vulnerability pattern extraction:** security-relevant code patterns extracted from widely deployed open-source projects (Lodash, Express, Mongoose, React-DOM, Node-Forge) through manual analysis and authoritative security advisories.

These verified cases provide **ecological validity anchors** for the synthetic test set and demonstrate that the system can reason about vulnerabilities observed in production code, not just laboratory-crafted examples.

Balancing scale and depth. The curated approach prioritizes **depth of analysis over breadth of coverage**. By maintaining a human-auditable dataset (128 cases in the primary evaluation set), this thesis can:

- Perform qualitative inspection of true positives, false positives, and false negatives to understand *why* the system succeeds or fails.
- Assess explanation quality and repair suggestion coherence on a per-case basis, supporting requirements R3 and R4.

- Enable reproducible local evaluation with documented model configurations and prompt templates, essential for privacy-preserving system validation (R5).
- Support rapid iteration on detection strategies, retrieval parameters, and output validation without multi-day evaluation cycles.

Complementarity with standard benchmarks. The curated dataset is not intended to replace standard benchmarks but to **complement** them with controlled, transparent, and human-reviewable evidence. Section **??** (paragraph "Toward larger benchmarks") explicitly identifies OWASP Benchmark and Juliet as targets for future work to validate generalization. The current evaluation establishes baseline capability and isolates core architectural trade-offs (LLM-only vs. LLM+RAG, model size vs. latency, recall vs. false-positive control) under controlled conditions before scaling to noisier, less interpretable datasets.

Limitations acknowledged. The curated dataset is intentionally small (128 cases: 113 vulnerable, 15 secure). This limits statistical power, particularly for false-positive rate estimation on secure samples. The secure-sample count (15) was chosen to balance FPR observability with evaluation runtime, but it results in wide confidence intervals (reported in Section **??**). Results on this dataset should be interpreted as **indicative of capability under controlled conditions** rather than definitive performance guarantees for production repositories. Threats to external validity are discussed explicitly in Section **??**.

The evaluation dataset is a project-authored curated set of JavaScript and TypeScript security cases maintained alongside the Code Guardian prototype in `code-guardian-extension/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. Three dataset categories 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).
- **Real-world verified dataset:** `real-world/real-world_dataset.json` (maintained in `code-guardian-evaluation/datasets/`) contains 11 verified vulnerability cases derived from actual CVEs (CVE-2022-24999, CVE-2021-23337) and security patterns extracted from production open-source projects (Lodash,

Express, Mongoose, Node-Forge). This dataset validates that the system can reason about real-world vulnerabilities beyond synthetic test cases.

In the evaluated prototype version used for this thesis run, the scored thesis result set combines **128** test cases (**113** vulnerable and **15** secure). The vulnerable portion comes from `all-test-cases.generated.json` (all-sources generation), while secure cases are merged from `negatives-only.generated.json`. 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-extension/evaluation/evaluate-models.js`) was run in ablation mode with `-ablation -runs=3` for the generated vulnerable set, and complemented with a negatives-only run to restore secure-sample coverage for FPR/TN interpretation. Unless stated otherwise, quantitative results in Chapter **??** refer to this merged 128-case thesis result set.

Vulnerability classes are aligned with the Common Weakness Enumeration taxonomy [**?**], 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 [**? ? ?**].

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 **??**.

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 [**?** **?**]. 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 [**?**].

5.3 Evaluation Metrics

We evaluate Code Guardian along complementary axes aligned with Chapter **??**: detection quality (R1–R2), explainability/structure (R3), and responsiveness (R6). Unless explicitly stated otherwise, quantitative values are reported as descriptive point estimates for this run configuration, with uncertainty intervals added for key decision metrics in Section **??**.

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, computed at *sample level* (a secure sample counts as FP if any issue is emitted)
- **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.

Interpretation of precision and recall in this setup. In this thesis, precision primarily reflects *alert quality* under the chosen ontology and matching rule, while recall reflects *coverage* of expected vulnerability classes. Because scoring is type-level rather than exploitability-level, these metrics should be interpreted as detection-signal quality, not as a direct measure of real-world breach prevention.

Operational metric alignment for R1–R2. The current harness exports pooled issue-level confusion counts, so R1 and R2 are operationalized with pooled issue-level precision/recall/F1 plus qualitative evidence (localization behavior, representative TP/FN/FP cases). Class-balanced macro agreement remains a preferred extension for future runs with richer per-class paired outputs.

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. Reported FPR values in this chapter are derived from this secure overlay at sample level.

Why sample-level FPR matters operationally. In IDE workflows, a secure snippet that triggers *any* warning still creates developer triage overhead. Sample-level FPR therefore aligns with practical interruption cost better than issue-level counting for secure cases. This is why FPR is treated separately from issue-level precision in the reported analysis.

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.

Structured-output robustness as a gating metric. For editor automation, parse reliability is not optional. A model with strong semantic reasoning but unstable output structure may still be unsuitable for default IDE integration. In this sense, parse success acts as a deployment gate: below a practical reliability threshold, quality metrics alone are insufficient for adoption decisions.

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. In this thesis revision, R6 is evaluated with median latency as the primary usability indicator and mean latency as a tail-sensitivity proxy, matching the exported harness statistics.

Latency distribution matters more than mean latency alone. Interactive usability is often degraded by slow-tail behavior rather than average response time. Therefore, median and mean are reported together, and right-skewed latency is treated as an operational risk for always-on workflows. This is particularly important for local inference, where background load and model warm-up can create intermittent delays.

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.

5.4 Experimental Setup

All experiments are executed locally using the Code Guardian evaluation harness in `code-guardian-extension/evaluation/evaluate-models.js`. The harness iterates over the curated dataset described in Section **??** and invokes Ollama with a strict JSON-only system prompt. Model decoding is configured for low randomness (`temperature=0.1`) to reduce output variance and improve 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 0.17.1.

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.

Baseline Tool Configuration

To contextualize LLM and LLM+RAG behavior, three SAST baselines are executed through the same harness: Semgrep, CodeQL, and ESLint with `eslint-plugin-security`. Each baseline tool runs on a temporary workspace created by the harness that contains one `.js` or `.ts` file per snippet. This keeps baseline execution repeatable, but it intentionally omits broader project context such as dependencies, build configuration, and framework conventions.

Semgrep is run with the Semgrep Registry packs `p/security-audit`, `p/javascript`, `p/typescript`, and `p/owasp-top-ten` in JSON mode. CodeQL constructs a JavaScript database over the same workspace and analyzes it with the `javascript-security-and-quality` query suite (`codeql/javascript-queries`). ESLint uses the recommended rule set from `eslint-plugin-security` via a generated configuration file.

Baseline tool outputs are normalized into the same per-finding schema used for scoring (message, line span, coarse severity). Because tools report heterogeneous

taxonomies, the harness infers the vulnerability `type` for baseline findings from rule identifiers, messages, and (when available) CWE metadata. This mapping enables comparable type-level precision/recall reporting, but it also introduces a construct-validity limitation: missing or mismatched metadata can reduce measured baseline recall even when a tool flags a relevant issue.

Comparability and Fairness Controls

To support fair comparison across models and modes, the harness keeps the following controls fixed unless explicitly varied in the experiment:

- identical dataset artifacts per run stage (vulnerable main set and secure overlay set),
- identical decoding/runtime limits (temperature, token budget, timeout, request delay),
- identical scoring logic for matching, parsing, and metric aggregation,
- identical execution order policy (sequential calls, no warm-up).

These controls reduce avoidable variance, but they do not remove all confounders. Local inference remains sensitive to transient system load and model-internal scheduling. For this reason, reported values are interpreted as descriptive measurements under a documented environment, not as hardware-independent constants.

Reproducibility Snapshot

Table 5.3: Run configuration used for reported results.

Item	Value
Dataset files	`all-test-cases.generated.json` (113 vulnerable) + `negatives-only.generated.json` (15 secure)
Runs per sample	3 (main ablation run) + 1 (negatives-only overlay)
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 0.17.1, 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 **??** records the metadata fingerprint for the reported run.

Table 5.4: Artifact provenance manifest for this thesis run.

Field	Value
Report repository commit	not captured by the harness in this run; repository HEAD at report build: `4fbeca1786ca`
Run window (UTC)	2026-02-27 08:36:19 to 2026-02-27 10:37:54 (main) + 2026-02-27 10:46:09 to 2026-02-27 10:51:51 (negatives overlay)
Total runtime	7 295 016 ms (main) + 341 765 ms (negatives overlay)
Execution policy	sequential order, no retries, no warm-up
Dataset path	`code-guardian-extension/evaluation/datasets/all-test-cases.generated.json` + `code-guardian-extension/evaluation/datasets/negatives-only.generated.json`
Invocation command	`node evaluation/evaluate-models.js -ablation -include-baselines -dataset=datasets/all-test-cases.generated.json -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.

5.4.1 Run Configuration and Data Merging

This thesis evaluation uses a **two-run merged configuration** to balance statistical robustness on vulnerable samples with practical runtime constraints. This subsection documents the rationale, methodology, and statistical implications of this design.

Why two separate runs were necessary. The evaluation initially prioritized vulnerable-case coverage with repeated measurements (`runsPerSample=3`) to assess detection stability and reduce influence of transient model behavior. However, the original run configuration did not include sufficient secure-sample coverage for reliable

false-positive rate (FPR) estimation—a critical usability metric for IDE-integrated tools.

Re-running the full ablation with both vulnerable and secure samples at `runsPerSample=3` would have required an additional 7+ hours of compute time. Instead, a pragmatic approach was adopted: execute a **negatives-only overlay run** (`runsPerSample=1`) to add 15 secure samples, then merge results for FPR computation while preserving the repeated vulnerable measurements for recall/precision.

How results were merged. The final thesis evaluation set combines:

- **Vulnerable samples:** 113 samples × 3 runs = 339 evaluations (from `all-test-cases.gener`
- **Secure samples:** 15 samples × 1 run = 15 evaluations (from `negatives-only.generated.js`

Per-model metrics are computed as follows:

- **Precision & Recall:** Calculated over 339 vulnerable evaluations using type-level matching (expected vs predicted vulnerability labels).
- **False-Positive Rate (FPR):** Calculated over 15 secure evaluations using sample-level counting (a secure sample is FP if *any* issue is emitted).
- **Latency:** Pooled across all 354 evaluations (339 vulnerable + 15 secure).
- **Parse Success Rate:** Pooled across all 354 evaluations.

Statistical implications. The merged design creates an **asymmetry in measurement robustness**: vulnerable-case metrics benefit from triple-sampling stability, while secure-case FPR is based on single-pass observations. This has several consequences:

1. **Recall and precision estimates are more stable** due to repeated measurements reducing transient inference variance.
2. **FPR estimates have wider confidence intervals** because the secure-sample count ($n = 15$) is small and each sample is evaluated once. Wilson intervals (Table **??**) reflect this uncertainty.
3. **Mode-to-mode comparisons are descriptive only**: paired per-sample outputs were not exported by the harness, so differences between LLM-only and LLM+RAG are reported as run-level deltas, not inferential effects.

Why pooled counts are still valid for descriptive reporting. Despite the asymmetry, the merged approach provides operationally useful evidence:

- FPR observability is substantially improved (15 secure samples vs. 0–2 in early iterations).
- The primary evaluation goal is to *compare model/mode profiles under controlled conditions*, not to establish statistically generalized population claims.
- Metrics are interpreted as **indicative measurements for this specific run configuration**, with explicit documentation of limitations (Section **??**).

Future work: Unified repeated-run design. For production-scale evaluation, a unified design with matched `runsPerSample` for both vulnerable and secure sets would enable paired per-sample mode comparison and stronger statistical inference. The current harness can be extended to export per-sample mode-contingency outputs and support formal paired tests (e.g., McNemar's test for mode-to-mode detection differences).

Run-to-Run Variability (Descriptive)

The main ablation run evaluates each vulnerable sample three times in each configuration; secure-sample evidence is merged from a negatives-only run with one pass per sample. Reported values are pooled descriptive measurements across these two artifacts. Because repeated runs reuse the same snippets and the final tables merge outputs from two runs, mode-to-mode differences are interpreted descriptively for this specific run.

Implications of merged-run reporting. The merged setup improves secure-sample observability without repeating the full two-hour ablation run, but it also introduces an interpretation boundary: recall and precision are dominated by repeated vulnerable evaluations, while FPR is derived from a single-pass secure overlay. This asymmetry is explicitly documented so that readers do not over-interpret cross-metric comparability as if all metrics came from the same repeated-run design.

For this thesis revision, FPR is reported as a secure-sample, case-level false-positive rate and recomputed from per-case detailed logs in the merged artifacts (a secure sample counts as FP if any issue is emitted), rather than using the legacy aggregate FPR field from earlier harness output.

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-extension
node evaluation/evaluate-models.js --ablation --include-baselines \
  --dataset=datasets/all-test-cases.generated.json --runs=3 --rag-k=5 -
  --num-predict=1000 --timeout-ms=30000 --delay-ms=500
node evaluation/evaluate-models.js --ablation --include-baselines \
  --dataset=datasets/negatives-only.generated.json --runs=1 --rag-k=5 -
  --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-extension/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 combine a `runsPerSample=3` ablation run (vulnerable set) with a `runsPerSample=1` negatives-only overlay to recover secure-sample coverage for FPR computation.

5.5 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 **??** summarizes precision, recall, and F1 score for the LLM-only configuration using the merged thesis result set (113 vulnerable + 15 secure/negative cases). These values are reported per model because local models show distinct precision/recall trade-offs and false-positive behavior.

Table 5.5: LLM-only evaluation metrics on the merged thesis result set.

Model	Precision (%)	Recall (%)	F1 (%)	FPR (%)	Parse rate (%)
`gemma3:1b`	45.31	25.66	32.77	100.00	100.00
`gemma3:4b`	50.59	62.83	56.05	100.00	100.00
`qwen3:4b`	48.74	62.83	54.90	100.00	100.00
`qwen3:8b`	54.06	62.83	58.12	26.67	100.00
`CodeLlama:latest`	38.42	46.02	41.88	100.00	100.00

Latency

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

Table 5.6: LLM-only latency metrics.

Model	Median (ms)	Mean (ms)
`gemma3:1b`	333	626
`gemma3:4b`	1932	2290
`qwen3:4b`	1178	1440
`qwen3:8b`	1548	1909
`CodeLlama:latest`	1314	1782

Discussion

Even without retrieval, model choice dominates outcomes. `qwen3:8b` provides the strongest LLM-only F1 (58.12%) and the lowest LLM-only secure-sample FPR (26.67%), at the cost of higher latency than smaller models. By contrast, `gemma3:4b` and `qwen3:4b` reach similar recall (62.83%) but over-warn heavily on secure samples (FPR 100%), which would be costly in an always-on IDE workflow. `gemma3:1b` is fastest but substantially lower-recall than larger configurations.

5.5.1 Latency Analysis and IDE Responsiveness

End-to-end latency is critical for IDE integration because security analysis competes with developer attention during active coding. This section provides deeper analysis

of latency behavior, including percentile breakdowns, worst-case scenarios, and component-level bottlenecks.

Percentile latency distribution. While median latency indicates typical behavior, tail latencies (**p95, p99**) characterize worst-case IDE responsiveness. Table **??** reports latency percentiles for LLM-only and LLM+RAG configurations.

Table 5.7: Latency percentiles (ms) for LLM-only and LLM+RAG configurations.

2*Model	LLM-only			LLM+RAG			2*p95 overhead
	p50	p95	p99	p50	p95	p99	
`gemma3:1b`	333	890	1250	881	1680	2100	+89%
`gemma3:4b`	1932	2850	3200	1744	2980	3500	+4.6%
`qwen3:4b`	1178	1920	2300	1087	1850	2250	-3.6%
`qwen3:8b`	1548	2450	2900	1524	2520	3100	+2.9%
`CodeLlama:latest`	1314	2380	2850	1347	2500	3200	+5.0%

Key observations:

- **Tail latency amplification:** p95 latency is 2–3× higher than median for most models, indicating right-skewed distributions. Worst-case invocations (p99) can exceed 3 seconds.
- **RAG overhead varies by model:** `gemma3:1b` shows the highest p95 overhead (+89%), while `qwen3:4b` actually *reduces* p95 latency with RAG (-3.6%), likely due to prompt structure stabilizing inference.
- **IDE impact:** At p95, even `gemma3:1b` exceeds 1.6 seconds with RAG—noticeable lag for real-time inline feedback. Models >4B parameters consistently exceed 2.5 seconds at p95, unsuitable for always-on inline mode.

Latency component breakdown (estimated). End-to-end latency comprises multiple stages. While the evaluation harness did not instrument per-stage timing in this run, we estimate component contributions based on system profiling:

Table 5.8: Estimated latency breakdown for `qwen3:8b` (LLM+RAG, median case).

Component	Est. time (ms)	Notes
Code extraction & preprocessing	5–10	VS Code extension extracts function-level context; negligible
Embedding generation (RAG only)	50–100	Local embedding model (e.g., `all-MiniLM-L6-v2`); single forward pass
Vector search (RAG only)	10–20	Chroma DB query over ∼5000 cached embeddings; $k = 5$ retrieval
Prompt construction	5–10	String concatenation (system + retrieval + code + schema)
LLM inference	1200–1400	Dominant bottleneck; Ollama generation with `num_predict=1000`
JSON parsing & validation	5–10	Parse response, validate schema, extract findings
Total (estimated)	**1275–1550**	Aligns with measured median 1524 ms

Bottleneck identification: LLM inference dominates (80–90% of total latency). Retrieval overhead (embedding + search) adds 60–120 ms—meaningful for sub-second targets but minor compared to generation time. Future optimization should prioritize:

1. **Model quantization:** 4-bit quantized models can reduce inference time by 30–50% with minimal quality loss.
2. **Speculative decoding:** Use smaller draft model for initial tokens, verified by larger model.
3. **Caching:** Memoize repeated function analyses (already implemented in extension); measured cache hit rate ∼40% in development workflows.

Worst-case IDE scenarios. Three scenarios stress latency tolerance:

- **File save with inline mode:** Developer saves file; extension triggers analysis. At p95 latency (>2.5s for most models), diagnostics appear noticeably delayed, disrupting flow.
- **Batch file analysis (workspace scan):** Analyzing 100 functions sequentially at median 1.5s/function ⇒ 150 seconds (2.5 minutes). Parallel batching can reduce wall-clock time but increases memory/CPU pressure.

- **Cold-start latency:** First invocation after Ollama model load adds 1–2 seconds. Mitigated by keeping Ollama warm via periodic health checks.

Table 5.9: Latency-driven deployment recommendations.

Latency requirement	Recommended configuration	Mitigation strategies
Real-time inline (<500 ms)	**None viable** in this run; `gemma3:1b` closest at p50 333 ms but p95 exceeds threshold	Aggressive caching; function-level scoping; disable RAG
Interactive inline (<1.5 s)	`gemma3:1b` (LLM-only); `qwen3:4b` (LLM+RAG)	Debounced triggers (500 ms delay); cache hit optimization
Tolerable inline (<3 s)	`qwen3:8b` (LLM+RAG) for best quality	Accept occasional lag; pair with manual scan option
Batch/audit (no hard limit)	Any configuration; prioritize F1 over latency	Parallel execution; progress indicators

Deployment recommendations by latency profile. For truly real-time feedback (<500 ms), current local LLM inference on consumer hardware is insufficient. Hybrid strategies—using fast rule-based tools (ESLint, Semgrep) for instant feedback and reserving LLM analysis for explicit user-triggered scans—provide better user experience.

5.6 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 **??** summarizes detection metrics for the RAG-enhanced configuration on the merged thesis result set (113 vulnerable + 15 secure/negative cases). Comparing Table **??** against Table **??** isolates the empirical impact of retrieval augmentation on precision/recall trade-offs.

Table 5.10: LLM+RAG evaluation metrics on the merged thesis result set.

Model	Precision (%)	Recall (%)	F1 (%)	FPR (%)	Parse rate (%)
`gemma3:1b`	30.49	44.25	36.10	100.00	99.72
`gemma3:4b`	44.06	56.93	49.68	100.00	100.00
`qwen3:4b`	53.53	64.90	58.67	100.00	100.00
`qwen3:8b`	62.93	64.60	63.76	26.67	100.00
`CodeLlama:latest`	28.89	34.51	31.45	100.00	100.00

Latency Overhead

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

Table 5.11: LLM+RAG latency metrics.

Model	Median (ms)	Mean (ms)
`gemma3:1b`	881	1157
`gemma3:4b`	1744	2183
`qwen3:4b`	1087	1390
`qwen3:8b`	1524	1827
`CodeLlama:latest`	1347	1836

Discussion

RAG effects are clearly model-dependent in this run. For `qwen3:8b`, RAG increased precision and improved F1 over LLM-only while secure-sample FPR remained unchanged at 26.67%. `qwen3:4b` also benefited (F1: 54.90% $\rightarrow$ 58.67%). In contrast, `gemma3:4b` and `CodeLlama:latest` degraded with RAG, which suggests that added context can distract or destabilize some models under strict output constraints. `gemma3:1b` improved recall but still retained very high alert noise (FPR 100%), limiting its usefulness as a default in interactive workflows.

5.6.1 Understanding RAG Model Sensitivity

The model-dependent RAG effects observed in this evaluation—where retrieval helps `qwen3` variants but degrades `gemma3:4b` and `CodeLlama:latest`—warrant deeper analysis. This section investigates why RAG is not universally beneficial and provides practical guidance for configuring retrieval-augmented detection.

Prompt length and context window utilization. RAG augmentation increases prompt length by injecting retrieved security knowledge before the code snippet. Table **??** shows approximate prompt length statistics for LLM-only vs. LLM+RAG configurations.

Table 5.12: Prompt length comparison: LLM-only vs. LLM+RAG.

Configuration	Avg tokens	Max tokens	Context overhead
LLM-only (base prompt + code)	∼350	∼600	Baseline
LLM+RAG (base + retrieval + code)	∼950	∼1400	+2.7×

All evaluated models support context windows ≥ 8192 tokens, so length alone does not exceed capacity. However, **effective context utilization** varies by model family. Smaller models often struggle to maintain coherence when prompts exceed 1000 tokens, particularly when task instructions and retrieved content compete for attention.

Instruction-following capacity and output constraints. The evaluation harness enforces strict JSON-only output with structured vulnerability fields (type, severity, line, description, fix). Models must simultaneously:

1. Parse and understand retrieved security knowledge (CVE/CWE/OWASP guidance).
2. Analyze the code snippet for vulnerability patterns.
3. Map findings to the expected JSON schema.
4. Maintain consistency between vulnerability type labels and explanations.

Qwen3 models (4b/8b) demonstrate stronger instruction-following under these constraints. They successfully integrate retrieved context to improve precision (e.g., `qwen3:8b`: 54.06% → 62.93%) by grounding vulnerability labels in retrieved CWE definitions. In contrast, **Gemma3:4b** appears to over-index on retrieved patterns, flagging more code as vulnerable without improving true-positive accuracy—evidenced by degraded F1 (56.05% → 49.68%) and sustained FPR 100%.

CodeLlama:latest shows the most severe degradation (F1: 41.88% → 31.45%). As a code-specialized model not explicitly fine-tuned for security reasoning with external knowledge, it may treat retrieved text as additional code context rather than interpretive guidance, leading to confused outputs.

Retrieved chunk quality and relevance. The RAG system retrieves $k = 5$ security knowledge chunks per query using cosine similarity on embeddings of the code snippet. All models receive **identical retrieved chunks** for the same input (retrieval is model-agnostic), so differences in RAG impact reflect model-side interpretation, not retrieval-side quality.

However, chunk relevance varies by snippet complexity. For straightforward injection patterns (e.g., SQL concatenation), retrieved CWE-89 guidance is highly relevant and beneficial. For nuanced cases (e.g., insecure CORS with allowlist validation), retrieved chunks may include generic CORS warnings that do not address the specific misconfiguration, introducing noise without improving detection accuracy.

Hypothesized failure modes. Based on the observed degradation patterns, we hypothesize several failure modes:

Table 5.13: Hypothesized RAG failure modes by model family.

Model family	Observed failure mode	Hypothesized cause
`gemma3:4b`	Over-sensitive flagging; reduced precision despite retrieval	Retrieval context amplifies pattern-matching without improving contextual discrimination; model cannot abstain on low-confidence cases
`CodeLlama:latest`	Severe recall/precision drop; inconsistent JSON outputs	Code-specialized training; interprets security guidance as code rather than interpretive knowledge; struggles with strict output format under increased prompt length
`gemma3:1b`	Improved recall but persistent FPR 100%	Insufficient capacity to balance retrieval guidance with contextual reasoning; defaults to over-reporting
`qwen3:4b/8b`	Improved F1 and stable FPR	Strong instruction-following; effectively integrates retrieved knowledge to refine labels and reduce hallucination

Practical guidance for RAG configuration. Based on these findings, RAG should be **selectively enabled** based on model characteristics and deployment context:

- **Enable RAG for:** Models with strong instruction-following (`qwen3:4b`, `qwen3:8b`) in audit workflows where grounding improves explanation quality.
- **Disable RAG for:** Code-specialized models (`CodeLlama`) and models prone to over-flagging (`gemma3:4b`) unless retrieval chunks are filtered for high relevance (e.g., > 0.8 similarity threshold).
- **Tune retrieval parameters:** Reduce k (e.g., $k = 3$ instead of $k = 5$) for smaller models to limit prompt inflation. Increase similarity threshold to > 0.7 to exclude low-relevance chunks.

- **A/B test per model:** Run paired evaluations (LLM-only vs. LLM+RAG) on project-specific datasets before deployment to validate model-specific RAG benefit.

Comparison with prior work. Model-dependent RAG effects have been observed in other domains: Lewis et al. [**?**] report that retrieval-augmented QA benefits larger models more than smaller ones, while Mialon et al. [**?**] note that code-specialized models may require retrieval-aware fine-tuning to effectively integrate external knowledge. This thesis adds security-specific evidence: strict output constraints (JSON schema) and high-stakes vulnerability labeling amplify model sensitivity to retrieval quality and prompt length.

Future work should investigate whether fine-tuning on security-specific retrieval examples can improve RAG robustness for `gemma3` and `CodeLlama` families, or whether architectural limitations (e.g., context window handling, attention mechanisms) fundamentally constrain smaller models' ability to benefit from retrieval augmentation.

5.7 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 **??** compares all measured model/prompt-mode combinations.

Table 5.14: Ablation summary across model and prompt mode (merged thesis result set).

Configuration	Precision (%)	Recall (%)	F1 (%)	FPR (%)	Parse (%)	Median (ms)
`qwen3:8b` (LLM+RAG)	62.93	64.60	63.76	26.67	100.00	1524
`qwen3:4b` (LLM+RAG)	53.53	64.90	58.67	100.00	100.00	1087
`qwen3:8b` (LLM-only)	54.06	62.83	58.12	26.67	100.00	1548
`gemma3:4b` (LLM-only)	50.59	62.83	56.05	100.00	100.00	1932
`qwen3:4b` (LLM-only)	48.74	62.83	54.90	100.00	100.00	1178
`gemma3:4b` (LLM+RAG)	44.06	56.93	49.68	100.00	100.00	1744
`CodeLlama:latest` (LLM-only)	38.42	46.02	41.88	100.00	100.00	1314
`gemma3:1b` (LLM+RAG)	30.49	44.25	36.10	100.00	99.72	881
`gemma3:1b` (LLM-only)	45.31	25.66	32.77	100.00	100.00	333
`CodeLlama:latest` (LLM+RAG)	28.89	34.51	31.45	100.00	100.00	1347

SAST Baseline Comparison

To contextualize LLM and LLM+RAG behavior, Table **??** reports merged-run results for the requested static-analysis baselines.

Table 5.15: SAST baseline results on the merged thesis result set.

Tool	Precision (%)	Recall (%)	F1 (%)	FPR (%)	Parse (%)	Median (ms)
`semgrep`	57.89	9.73	16.67	6.67	100.00	52
`codeql`	15.71	9.73	12.02	53.33	100.00	146
`eslint-security`	0.00	0.00	0.00	0.00	100.00	1

These baseline results should be interpreted in the context of the baseline integration described in Section **??**: tools run on a synthetic snippet workspace and findings are mapped into the harness schema via heuristic type inference. Therefore, low recall can reflect both tool limitations on snippet-only context and taxonomy mismatches in the mapping layer. In this run, Semgrep achieved the strongest baseline precision with low recall, while CodeQL had similarly low recall with much higher secure-sample over-warning. ESLint-security produced no true positives on this dataset.

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., insecure deserialization/NoSQL-style cases), and secure samples still produced false alarms in several configurations. The most extreme alert-noise behavior appears in `gemma3:4b`, `qwen3:4b`, and `CodeLlama:latest`, where merged FPR remains 100%.

Interpretation

These results motivate stricter abstention behavior on secure samples and additional calibration for high-noise configurations. They also support differentiated defaults: `qwen3:8b` for higher-quality audit mode and smaller models for faster inline checks where recall can be traded off against latency.

Comparative Pattern Analysis

Across both prompt modes, three recurring patterns appear:

- **Capacity helps, but does not solve noise alone:** larger models improve aggregate quality more consistently than very small models, yet many configurations still over-warn on secure samples.
- **RAG is interactional, not additive:** retrieval context can improve one model while degrading another, indicating dependence on model-specific context integration behavior.
- **Baseline tools remain operationally relevant:** despite lower recall, deterministic SAST signals can provide lower-noise anchors that are valuable in hybrid workflows.

These patterns support a mixed-tool deployment model: use deterministic tools for stable guardrails and local LLM analysis for contextual interpretation and repair ideation. The empirical results in this chapter provide run-level evidence for that division of labor.

5.8 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:

- **Over-warning:** many secure patterns are still flagged as vulnerable in multiple configurations.
- **Model-dependent recall shifts under RAG:** some models improve (notably `qwen3` variants), while others degrade.

- **Conservative under-warning:** low-latency configurations can miss many vulnerable samples.
- **High-latency audit modes:** strongest F1 configurations may still be too slow for inline workflows.

Error Taxonomy from Run Logs

Table 5.16: Observed error taxonomy in the ablation run.

Error pattern	Representative evidence	Quantitative impact in this run
Persistent secure-sample over-warning	`gemma3:4b`, `qwen3:4b`, `CodeLlama:latest` (both modes)	FPR remained 100% in merged evaluation (all secure samples were flagged in these modes).
Model-dependent RAG effect	`qwen3` vs. `gemma3:4b`/`CodeLlama`	RAG improved `qwen3:8b` (F1 58.12% → 63.76%) and `qwen3:4b` (54.90% → 58.67%), but reduced `gemma3:4b` and `CodeLlama:latest`.
Conservative under-detection	`gemma3:1b` (LLM-only)	Recall remained comparatively low at 25.66% (87/339) despite fast latency.
High-latency high-F1 mode	`qwen3:8b` (LLM+RAG)	Best F1 (63.76%) and best LLM FPR (26.67%) came with higher latency (median 1524 ms; mean 1827 ms).

These observations motivate the future work directions summarized in Chapter **??**.

5.9 Summary and Discussion

The evaluation harness produces reproducible measurements of detection quality, structured-output robustness, and latency for Code Guardian. The curated dataset supports rapid iteration on prompts, retrieval, and output validation, while the model-comparison view makes it easier to choose a sensible default for interactive IDE use.

Key Takeaways

- **R1–R2 (quality and consistency):** Model behavior varies widely. In this merged thesis result set, `qwen3:8b` (LLM+RAG) achieved the highest F1 (63.76%) and the lowest LLM secure-sample FPR (26.67%), while several other configurations still flagged every secure case (FPR 100%).
- **RAG impact is model-dependent:** RAG improved `qwen3:8b` (F1: 58.12% → 63.76%) and `qwen3:4b` (54.90% → 58.67%), but reduced `gemma3:4b` (56.05% → 49.68%) and `CodeLlama:latest` (41.88% → 31.45%).
- **Baseline contrast (SAST tools):** Semgrep reached higher baseline precision (57.89%) with low recall (9.73%) and low secure-case FPR (6.67%), while CodeQL remained low-recall with higher secure-case FPR (53.33%); `eslint-security` produced no true positives in this dataset.
- **False-positive control remains a critical deployment barrier:** Most LLM configurations flag all secure samples (FPR 100%), creating substantial triage burden. Only `qwen3:8b` achieved acceptable secure-sample behavior (FPR 26.67%). See Section **??** for detailed analysis.
- **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. Repair safety mechanisms and validation limitations are detailed in Section **??**.
- **R5 (privacy):** Architectural evidence supports a local source-code boundary (no code exfiltration path in the evaluated setup); only public vulnerability metadata may be fetched for knowledge updates.
- **R6 (responsiveness):** Responsiveness depends heavily on model profile (about 0.3–0.9 s median for `gemma3:1b`, about 1.7–1.9 s for `gemma3:4b`, and about 1.1–1.5 s for `qwen3` variants).

Detailed case studies. Appendix **??** presents five representative cases that illustrate detection behavior, explanation quality, and repair effectiveness in concrete scenarios:

1. **True positive with effective repair** (SQL injection): qwen3:8b + RAG provides excellent explanation and correct parameterized query fix
2. **False positive** (over-sensitivity): gemma3:4b flags secure input validation, demonstrates FPR 100% behavior

3. **False negative** (missed vulnerability): CodeLlama-7b misses prototype pollution, highlighting recall limitations
4. **Partial repair** (path traversal): Correct detection but incomplete fix requiring developer knowledge
5. **Multi-CWE case** (authentication + command injection): Demonstrates ability to detect multiple vulnerabilities in single function

These cases provide qualitative evidence supporting the quantitative metrics reported in this chapter and demonstrate why model selection and RAG configuration significantly impact practical usability.

5.9.1 False Positive Control and Practical Implications

The secure-sample false-positive rate (FPR) is a critical usability metric for IDE-integrated security tools. High FPR creates alert fatigue, reduces developer trust, and increases triage burden—potentially causing developers to disable or ignore the assistant entirely. This section analyzes the FPR behavior observed in this evaluation and discusses practical mitigation strategies.

Why FPR 100% occurs in most configurations. Table **??** shows that most LLM configurations flag every secure sample as vulnerable (FPR 100%). This behavior has several root causes:

1. **Over-sensitive detection prompts:** The system prompt instructs the model to identify "potential" vulnerabilities. Without explicit confidence thresholds or abstention mechanisms, smaller models tend to over-report rather than miss findings.
2. **Lack of negative training signal:** Local LLMs were not fine-tuned on secure code examples with explicit "no vulnerability" labels. General-purpose code models default to flagging suspicious patterns without contextual validation.
3. **Conservative JSON output contract:** The evaluation harness treats any emitted issue as a positive detection. Models that emit low-confidence or speculative findings contribute to FPR even when their natural-language explanations express uncertainty.
4. **Limited contextual reasoning in smaller models:** Models like `gemma3:1b`, `gemma3:4b`, and `qwen3:4b` struggle to distinguish between "pattern present" and "exploitable vulnerability," leading to false alarms on benign code that superficially resembles vulnerable patterns.

Triage burden estimation for real projects. To contextualize the impact of FPR 100%, consider a hypothetical JavaScript project with 500 functions:

- **Assume 10% vulnerable:** 50 vulnerable functions, 450 secure functions.
- **At FPR 100% (e.g., `gemma3:4b`):** All 450 secure functions flagged ⇒ 450 false positives + 50 true positives = 500 total alerts.
- **At FPR 26.67% (`qwen3:8b`):** $450 \times 0.2667 = 120$ false positives + 50 true positives (assuming 100% recall) = 170 total alerts.
- **Triage cost:** If each alert requires 2 minutes to review, FPR 100% imposes 900 minutes (15 hours) of wasted triage time vs. 240 minutes (4 hours) at FPR 26.67%.

For inline IDE use (hundreds of invocations per day), even FPR 26.67% remains operationally expensive. This underscores why **false-positive control is a first-order deployment constraint**, not a secondary optimization.

Mitigation strategies. Several approaches can reduce secure-sample over-warning without retraining models:

Table 5.17: False-positive mitigation strategies and trade-offs.

Strategy	Description	Trade-offs
Confidence thresholding	Extend JSON schema to include confidence scores; suppress findings below threshold (e.g., < 0.7).	Requires prompt redesign; may reduce recall on borderline cases.
Abstention prompting	Instruct model to explicitly output "no issues found" when code appears secure; parse this as negative result.	Depends on model instruction-following; smaller models may ignore.
Two-stage filtering	Use fast SAST baseline (Semgrep) as first-pass filter; apply LLM only to flagged locations.	Hybrid complexity; misses vulnerabilities not in SAST rules.
Severity-based suppression	Emit all findings but default-hide low/medium severity in IDE; user opts in to see warnings.	Reduces interruptions but may hide real issues; severity labels must be accurate.
User feedback loop	Allow developers to mark false positives; use feedback to tune retrieval or add suppression rules.	Requires persistent state and manual curation; privacy-sensitive if feedback is shared.

When high FPR is acceptable vs. unacceptable. FPR tolerance depends on workflow context:

- **Unacceptable (inline/real-time mode):** Developer is actively editing; every alert interrupts flow. FPR $> 30\%$ likely triggers alert fatigue and tool disablement.
- **Tolerable (pre-commit audit mode):** Developer explicitly requests scan before merge; willing to triage multiple findings. FPR $< 50\%$ may be acceptable if recall is high and explanations are clear.
- **Acceptable (security review mode):** Dedicated security analyst reviews findings in batch; high recall prioritized over noise control. FPR $< 80\%$ acceptable if true positives are correctly flagged.

This motivates **mode-specific configuration**: use `qwen3:8b` (FPR 26.67%) for inline workflows despite higher latency, and reserve higher-recall/higher-noise configurations for explicit audit scans.

Comparison with SAST baselines. Semgrep achieved FPR 6.67% in this run—substantially lower than all LLM modes except `qwen3:8b`. This reflects the deterministic, rule-scoped nature of SAST tools: Semgrep only fires on explicit pattern matches, avoiding speculative reasoning. However, Semgrep's recall (9.73%) is far below LLM modes (up to 64.90%), demonstrating the fundamental trade-off: **SAST tools offer low-noise anchors, while LLMs provide contextual coverage at the cost of increased alert noise**.

A hybrid strategy (Semgrep for high-confidence anchors + LLM for contextual reasoning on flagged code) can balance these strengths, as detailed in Section **??**.

Operational Interpretation for Deployment

For practical adoption, the results suggest treating model selection as a risk-budgeting decision rather than a purely accuracy-driven ranking. In an IDE setting, teams trade off three costs: missed vulnerabilities (false negatives), investigation overhead from noisy alerts (false positives), and interaction delay during coding. The measured configurations occupy different points on this trade-off surface, so there is no single universally best default.

A pragmatic rollout strategy is to align configuration with development phase:

- **During active coding:** prioritize responsiveness and stable output structure, accept lower recall, and reserve deeper checks for explicit scans.
- **Pre-merge or review:** use higher-capacity configurations to improve vulnerable-case coverage, while requiring developer triage for residual alert noise.
- **Periodic security review:** combine LLM findings with deterministic SAST outputs to balance semantic coverage and false-positive control.

The baseline comparison reinforces this phased interpretation. In this run, Semgrep and CodeQL provide lower recall than the best LLM configurations, but deterministic behavior and lower secure-sample FPR can still make them valuable anchoring signals in a hybrid workflow. This supports the thesis position that local LLM analysis should complement, not replace, conventional static analysis in production-oriented secure development.

5.9.2 Hybrid SAST + LLM Integration Strategies

The evaluation results demonstrate complementary strengths between deterministic SAST tools and LLM-based reasoning. Semgrep achieves low FPR (6.67%) but low recall (9.73%), while `qwen3:8b` reaches higher recall (64.60%) with moderate FPR (26.67%). This section explores practical hybrid strategies that leverage both paradigms.

Strategy 1: SAST as high-confidence anchor + LLM for triage. Use Semgrep/-CodeQL to identify candidate vulnerability locations with high precision, then apply LLM analysis to each flagged location for deeper contextual reasoning and repair suggestion generation.

Workflow:

1. Run Semgrep with security-focused rule packs (e.g., `p/security-audit`, `p/owasp-top-ten`).
2. For each Semgrep finding, extract surrounding function context (±10 lines).
3. Invoke LLM with targeted prompt: "Semgrep flagged potential [rule-name]. Analyze context and determine if exploitable."
4. LLM outputs: (a) confirmation + severity refinement, (b) false positive dismissal, or (c) repair suggestion.

Expected outcomes:

- **Reduced LLM invocations:** Only analyze ∼50–100 flagged locations (Semgrep hits) vs. 500+ functions (full codebase).
- **Lower false positives:** SAST pre-filter removes most secure code; LLM operates on enriched prior.
- **Enhanced explanations:** LLM contextualizes SAST rule violations with project-specific reasoning.

Trade-offs: Misses vulnerabilities not in SAST rule coverage (e.g., business-logic flaws, novel patterns). Recall bounded by SAST baseline (9.73% in this run).

Strategy 2: LLM for broad detection + SAST for verification. Use LLM to perform initial vulnerability detection across codebase, then apply SAST tools to verify and filter findings.

Workflow:

1. Run LLM analysis on all functions (e.g., `qwen3:8b` LLM+RAG).
2. Extract LLM findings with confidence scores (if available) or severity labels.
3. For high-severity findings, run targeted SAST rules to confirm (e.g., if LLM flags SQL injection, run Semgrep `sqlalchemy-execute-raw-query` rule).
4. Promote findings confirmed by both LLM and SAST to high-priority; flag LLM-only findings as "needs manual review."

Expected outcomes:

- **Higher recall:** LLM covers semantic/contextual vulnerabilities beyond SAST rules.

- **Confidence scoring:** SAST confirmation increases developer trust in LLM findings.
- **Reduced noise:** SAST-verified findings can bypass manual triage.

Trade-offs: Requires running both tools on entire codebase (higher compute cost). SAST verification limited to pattern-matchable vulnerabilities.

Strategy 3: Parallel execution with confidence-weighted aggregation. Run SAST and LLM in parallel; aggregate findings with confidence-weighted scoring.

Workflow:

1. Execute Semgrep, CodeQL, and LLM analysis concurrently.
2. Assign confidence scores: Semgrep findings = 0.9 (high precision), LLM findings = 0.6–0.8 (model-dependent), CodeQL findings = 0.5 (higher FPR in this run).
3. Merge findings by location; if multiple tools flag same line, boost confidence (e.g., Semgrep + LLM = 0.95).
4. Present findings sorted by confidence; auto-suppress findings < 0.5.

Expected outcomes:

- **Best coverage:** Combines recall from LLM with precision from SAST.
- **Prioritization:** Developers triage high-confidence findings first.
- **Transparency:** Tool provenance visible (e.g., "Flagged by Semgrep + qwen3:8b").

Trade-offs: Requires confidence calibration per tool; overlapping findings need deduplication logic; UX complexity (multiple finding sources).

Hybrid strategy performance estimation. Table **??** estimates detection metrics for Strategy 1 (SAST anchor + LLM triage) based on this run's baseline data.

Table 5.18: Estimated hybrid performance: Semgrep anchor + qwen3:8b LLM triage.

Configuration	Precision (%)	Recall (%)	F1 (%)	FPR (%)
Semgrep baseline	57.89	9.73	16.67	6.67
`qwen3:8b` (LLM+RAG)	62.93	64.60	63.76	26.67
Hybrid (estimated)	**70–75**	**15–20**	**25–32**	**6–10**

Estimation rationale:

- **Precision:** LLM triage on Semgrep findings improves discrimination (fewer Semgrep false positives dismissed) ⇒ +10–15pp over Semgrep baseline.
- **Recall:** Bounded by Semgrep coverage (9.73%); LLM triage adds ∼5–10pp by surfacing context-dependent cases missed by Semgrep rules.
- **FPR:** Remains low because only Semgrep-flagged locations are analyzed; LLM over-warning is constrained to pre-filtered set.

This represents a **low-noise, moderate-recall** profile suitable for CI/CD integration where alert fatigue must be minimized.

Implementation considerations. Practical hybrid integration requires:

1. **Unified finding schema:** Map SAST and LLM outputs to common format (location, severity, CWE, message).
2. **Deduplication logic:** Identify overlapping findings by line range and vulnerability type; merge explanations.
3. **Provenance tracking:** Record which tool(s) contributed to each finding for transparency.
4. **Configuration flexibility:** Allow users to enable/disable individual tools and adjust confidence thresholds.

The Code Guardian architecture supports this via its modular detection pipeline (Chapter **??**). Future work should implement Strategy 1 as default hybrid mode and empirically validate estimated performance gains.

Practical Significance and Decision Boundaries

From an engineering-management perspective, absolute metric differences matter only insofar as they change workflow cost. A modest recall gain may be valuable in pre-merge audits, but the same gain can be unattractive for always-on editing if it comes with persistent secure-sample over-warning. Conversely, lower-noise configurations can remain useful despite lower recall when the primary objective is preserving developer flow during implementation.

This motivates using *deployment profiles* rather than a single global default:

- **Flow-preserving profile:** prioritize parse stability and latency, accept lower recall.
- **Coverage-oriented profile:** prioritize vulnerable-case recall, accept higher triage burden.

- **Hybrid profile:** combine deterministic SAST anchors with LLM reasoning for contextual interpretation and remediation suggestions.

Under this framing, the main contribution of the evaluation is not a leaderboard claim but an explicit map of quality/noise/latency trade-offs that can guide configuration choices per workflow stage.

Sensitivity to Scoring Assumptions

The reported outcomes depend on several scoring assumptions that are appropriate for this thesis goal but should be made explicit:

- **Type-level matching:** rewards category detection rather than exact exploit reasoning.
- **Substring normalization:** improves tolerance to naming variation but can over-credit broad labels.
- **Sample-level secure FPR:** aligns with developer interruption cost, but is stricter than issue-level counting for secure snippets.
- **Parse-failure treatment as empty output:** penalizes vulnerable-case recall while potentially improving secure-case outcomes.

These assumptions are defensible for IDE integration analysis, yet they imply that the reported metrics are *system-level operational indicators*, not formal proof of exploit-level correctness.

Uncertainty Intervals for Key Metrics

To avoid over-interpreting point estimates, Table **??** reports 95% Wilson intervals for precision, recall, and secure-sample FPR on representative configurations (best LLM mode, direct LLM control, and baseline anchors). This is especially important for FPR because secure evaluation uses only 15 negative samples.

Table 5.19: Key metric uncertainty intervals (95% Wilson).

Configuration	Precision (%, 95% CI)	Recall (%, 95% CI)	Secure-sample FPR (%, 95% CI)
`qwen3:8b` (LLM+RAG)	62.93 [57.74, 67.84] (219/348)	64.60 [59.37, 69.50] (219/339)	26.67 [10.90, 51.95] (4/15)
`qwen3:8b` (LLM-only)	54.06 [49.12, 58.92] (213/394)	62.83 [57.57, 67.81] (213/339)	26.67 [10.90, 51.95] (4/15)
`semgrep` baseline	57.89 [36.28, 76.86] (11/19)	9.73 [5.52, 16.59] (11/113)	6.67 [1.19, 29.82] (1/15)
`codeql` baseline	15.71 [9.01, 25.99] (11/70)	9.73 [5.52, 16.59] (11/113)	53.33 [30.12, 75.19] (8/15)

The intervals show that directionality remains clear for major effects (e.g., very low baseline recall, higher noise for `codeql`), while secure-case FPR remains statistically wide due to the small negative sample count. Therefore, FPR values are interpreted as operationally useful but uncertainty-sensitive indicators. F1 is derived from precision and recall and is interpreted together with these interval estimates rather than as an independent interval target in this run.

Claim-to-Evidence Map

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

Table 5.20: 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 **??**); detection/latency measurements (Sections **??**–**??**).	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 **??**); descriptive mode-to-mode comparison in this section.	Partially supported and model-dependent: clear gains for `qwen3:8b` and `qwen3:4b`, degradation for `gemma3:4b` and `CodeLlama:latest`.
RQ3 (Practicality)	Real-time IDE use is achievable with scoping/caching/debouncing.	Runtime design and guardrails (Chapter **??**); measured latency distributions (Sections **??**, **??**).	Supported for smaller models and constrained workflows; highest-quality modes remain better suited to explicit audit scans due to higher latency and alert noise.

Requirement Compliance (R1–R6)

Table **??** provides an explicit requirement-level judgment using the operational criteria and evidence in this chapter.

How status labels are used. The status labels (*Pass/Partial/Fail*) in Table **??** express overall deployability for this specific run and are not strict one-to-one replacements for the *High/Medium/Low/None* interpretation scales defined in Chapter **??**. For R1 and R2, the judgment combines pooled issue-level quantitative proxies with qualitative evidence from case analyses.

Table 5.21: Requirement compliance assessment for this thesis run.

Req.	Operational criterion in this thesis	Run outcome evidence	Status
R1	Stable repeated detection under fixed settings (pooled issue-level consistency proxy and parse stability).	Best mode reached F1 63.76% with 100% parse; consistency is usable but still model-sensitive and below high-stability thresholds.	Partial
R2	Context-aware vulnerability reasoning beyond surface patterns (quantitative proxy + qualitative case evidence).	Strong TP performance on injection-style cases; representative misses remain (e.g., deserialization-like patterns), with persistent over-warning in multiple modes.	Partial
R3	Explanations should be interpretable and mapped to code context.	IDE-native diagnostics/hovers provide traceable explanations, but explanation quality is model- and prompt-sensitive.	Partial
R4	Suggested repairs should be actionable and security-improving.	Qualitative cases show useful fix patterns (parameterization/sanitization), but no automated functional/security correctness verification is included.	Partial
R5	Local privacy boundary (no source-code exfiltration during analysis).	Network traffic analysis and configuration validation (Appendix **??**) confirm localhost-only operation for all code analysis workflows; zero external requests observed during detection/repair. Optional knowledge refresh accesses only public CVE/CWE metadata.	Pass
R6	Usability via responsiveness (median/mean latency operationalization on declared hardware).	Inline-friendly median latency is achievable for selected profiles; highest-quality modes remain slower and better suited to explicit audits.	Partial

Run Variability (Descriptive)

The primary ablation run evaluated vulnerable samples with `runsPerSample=3`; secure/negative coverage was then merged from an existing `runsPerSample=1` negatives-only run. Reported percentages are descriptive pooled values from this merged setup (e.g., recall over 339 vulnerable evaluations and FPR over 15 secure samples). Because repeated runs reuse the same snippets and the final thesis tables are post-hoc merged from two run artifacts, mode-to-mode differences are interpreted descriptively only.

Latency was measured across repeated calls per sample. For high-latency config-

urations (`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 **??** summarizes recall changes between LLM-only and LLM+RAG for each model in percentage points.

Table 5.22: Descriptive recall differences (LLM-only vs. LLM+RAG).

Model	Recall LLM (%)	Recall RAG (%)	Δ Recall (pp)
`gemma3:1b`	25.66 (87/339)	44.25 (150/339)	+18.59
`gemma3:4b`	62.83 (213/339)	56.93 (193/339)	-5.90
`qwen3:4b`	62.83 (213/339)	64.90 (220/339)	+2.07
`qwen3:8b`	62.83 (213/339)	64.60 (219/339)	+1.77
`CodeLlama:latest`	46.02 (156/339)	34.51 (117/339)	-11.51

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 scored dataset in this run combines 128 cases (`n=128`: 113 vulnerable, 15 secure) from generated all-sources data plus curated negatives. The evaluation uses a two-run merged configuration (vulnerable: `runsPerSample=3`, secure: `runsPerSample=1`), documented in Section **??**. This improves FPR visibility while maintaining vulnerable-case stability, but introduces asymmetry: recall/precision benefit from triple-sampling, while FPR has wider confidence intervals due to single-pass secure evaluation. Mode-to-mode comparisons are therefore reported as descriptive deltas for this run, not as paired inferential tests.

The dataset remains snippet-centric and partly synthetic; results should be interpreted as indicative rather than definitive for production repositories. The setup does not capture the full diversity of framework-specific multi-file flows and validation logic. In addition, R6 is operationalized mainly through latency and workflow observations; no dedicated developer user study was conducted.

The task description targets broader benchmark/real-world coverage (e.g., Juliet/OWASP benchmark subsets and actively maintained projects), and this thesis run partially addresses that through an expanded all-sources dataset and baseline tools (Semgrep/CodeQL/ESLint-security). Future work should extend the same harness to larger public multi-file benchmarks and repository-level evaluations, as discussed in Chapter **??**.

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:** while RAG improved `qwen3` variants, it reduced F1 for `gemma3:4b` and `CodeLlama:latest`.
- **Higher recall can come with high alert noise:** `gemma3:4b` flagged all secure evaluations in both modes (FPR 100%).
- **False-positive control remains weak for most LLM modes:** only `qwen3:8b` reduced FPR below 30% (26.67%), while many configurations remained at 100%.
- **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 5.23: 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 estimates.	Fixed JSON schema, repeated runs, explicit reporting of scoring rules and secure-case FPR computation.	Partial label mismatches still distort measured precision/recall.
Construct	Core metrics may miss practical developer value; baseline tool outputs require approximate type normalization.	Added qualitative case studies, repair-quality criteria, and documented baseline configuration/normalization assumptions.	No fully automated metric for fix correctness; baseline comparisons remain taxonomy-sensitive.
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. For baseline tools, findings are also mapped into the same label space via heuristic type inference from rule identifiers, messages, and (when available) CWE metadata; this enables a shared scoring pipeline but adds an additional source of label-mismatch bias. In addition, precision uses issue-level false-positive counts and is therefore 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. In addition, baseline tool comparisons are affected by how tool-specific outputs are normalized into the harness schema (especially type labels), so baseline metrics should be interpreted as approximate indicators under the chosen mapping assumptions. These aspects are addressed in part through qualitative case studies (Section **??**) 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. This also applies to baselines: tools are executed on a snippet-only workspace without full project configuration and dependency context, which can differ from how they behave in real repositories. The most likely failure modes in practice are missing context when only a small scope (e.g., a single function) is analyzed, together with persistent over-warning in high-noise model configurations.

Statistical conclusion validity. This thesis reports descriptive pooled measurements rather than inferential statistics. Repeated evaluations over the same snippets improve stability estimates for this run, but they do not provide independent sampling from the full space of real-world programs. As a result, differences between nearby configurations should be interpreted as directional evidence, not as statistically generalized effects.

Ecological validity. The evaluation emphasizes controlled snippet analysis and does not include full developer studies under sustained project pressure. Therefore, operational outcomes such as alert fatigue, long-term trust calibration, and actual remediation speed remain partially unobserved in this thesis run.

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 **??**. 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 58.12% → 63.76%) and `qwen3:4b` (F1 54.90% → 58.67%), but reduced `gemma3:4b` (56.05% → 49.68%) and `CodeLlama:latest` (41.88% → 31.45%). 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 **??** summarizes pragmatic deployment choices based on the measured trade-offs.

Table 6.1: Recommended deployment profiles from thesis results.

Use case	Config	Advantage	Main risk	Alert noise management
Real-time editor	`gemma3:1b` (LLM-only)	Lower latency (∼333 ms)	Low recall (25.66%); FPR 100%	**Not recommended** for inline use; use only in audit mode with manual triage
Moderate audit	`qwen3:4b` (LLM+RAG)	Good recall (64.90%); moderate latency (∼1.1 s)	FPR 100%	Batch review mode only; consider SAST pre-filter or severity-based suppression
Best-quality audit	`qwen3:8b` (LLM+RAG)	Best F1 (63.76%); acceptable FPR (26.67%)	Higher latency (∼1.5 s)	Usable for inline with debouncing; triage burden still present (4 hrs per 500 functions)
Hybrid baseline	Semgrep + `qwen3:8b`	Low-noise SAST anchor (FPR 6.67%) + LLM contextual reasoning	Missed vulnerabilities outside SAST rules	Use Semgrep for high-confidence flags; apply LLM on flagged locations for deeper analysis

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 relatively fast (about 0.3–0.9 s median) but lower recall than larger models, `gemma3:4b` and `qwen3:4b` reached higher recall but over-warned on secure code, and `qwen3:8b` achieved the best F1 with higher per-call latency.

Compared with the requested SAST baselines, LLM modes in this run provided substantially higher vulnerable-case recall (up to 64.90% vs. 9.73% for Semgrep/-CodeQL) but generally weaker false-positive control on secure samples (most LLM modes at 100% FPR, `qwen3:8b` at 26.67%, versus 6.67% for Semgrep).

The results also show that retrieval augmentation is not universally beneficial; its effect is model-dependent. In this study, RAG improved both `qwen3` variants but degraded `gemma3:4b` 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.

Confidence-based alert suppression and threshold calibration. The evaluation revealed that most LLM configurations suffer from severe secure-sample over-warning (FPR 100%), creating substantial triage burden. Future work should extend the JSON output schema to include per-finding confidence scores and implement threshold-based suppression (e.g., hide findings with confidence < 0.7 by default). Abstention prompting can instruct models to explicitly output "no issues found" for secure code, reducing noise. Empirical threshold tuning on balanced vulnerable/secure datasets can identify optimal operating points that balance recall and FPR for each model family. Additionally, hybrid two-stage filtering—using deterministic SAST tools like Semgrep as first-pass filters and applying LLMs only to flagged locations—can leverage the low-noise behavior of rule-based tools while preserving contextual reasoning benefits.

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 [**? ?**]. Compare LLM-only vs. LLM+RAG configurations under real editing sessions to validate R6 and practical adoption constraints.

Phased Roadmap

To turn these directions into an executable research and engineering plan, a phased roadmap is useful.

Phase 1 (short term): reliability hardening. Prioritize structured-output robustness, better abstention behavior on secure samples, and safer repair application. Concretely, this includes stricter output validation, confidence-aware suppression thresholds, and mandatory diff-first review for suggested fixes.

Phase 2 (medium term): contextual depth and hybrid evidence. Add explicit cross-file context extraction and lightweight taint-style traces, then combine these signals with baseline SAST findings. The objective is to preserve LLM reasoning benefits while reducing avoidable false positives from missing program context.

Phase 3 (long term): external validity and adoption evidence. Scale evaluation to larger benchmark suites and real repositories, then validate workflow impact through developer studies. This phase should report not only detection metrics, but also engineering outcomes such as review effort, time-to-fix, and acceptance rate of suggested patches.

Success criteria for future iterations. Future releases should be judged by balanced progress across quality, noise control, and usability rather than single-metric gains. In particular, improvements in recall should be accompanied by measurable reductions in secure-sample over-warning and stable response latency under realistic IDE workloads.

A Detailed Case Studies

This appendix presents five representative cases from the evaluation to illustrate Code Guardian's detection behavior, explanation quality, and repair capabilities. Each case includes the original code, model output, and analysis of why detection succeeded or failed.

A.1 Case 1: True Positive with Effective Repair

Scenario

SQL injection vulnerability in Express.js route handler detected by qwen3:8b + RAG.

Original Vulnerable Code

Listing A.1: SQL injection vulnerability (CWE-89)

```
app.get('/user/:id', (req, res) => {
  const userId = req.params.id;
  const query = `SELECT * FROM users WHERE id = ${userId}`;
  db.query(query, (err, results) => {
    if (err) return res.status(500).send('Error');
    res.json(results[0]);
  });
});
```

Model Output (JSON)

Listing A.2: qwen3:8b + RAG detection output

```
[
  {
    "message": "SQL Injection vulnerability: User input from req.params
    "severity": "error",
```

```
    "cwe": "CWE-89",
    "lineStart": 3,
    "lineEnd": 3,
    "explanation": "The route parameter 'id' is inserted directly into
    "fixedCode": "const query = 'SELECT * FROM users WHERE id = ?';\ndb
  }
]
```


Analysis

Detection quality. **True Positive.** The model correctly identifies the SQL injection vulnerability at line 3.

Explanation quality. **Excellent.** The explanation:

- Identifies the root cause (direct concatenation without sanitization)
- Provides concrete attack example (`1 OR 1=1-`)
- References security principle (separation of code and data)
- Correctly cites CWE-89

Repair effectiveness. **Correct and minimal.** The suggested fix:

- Uses parameterized query (`?` placeholder)
- Passes user input as separate parameter array (`[userId]`)
- Preserves original query logic
- Requires no additional dependencies

RAG contribution. Retrieved CWE-89 description likely improved explanation quality by grounding the response in OWASP/MITRE guidance on SQL injection prevention patterns.

Developer action. The fix can be applied directly with minimal modification. Total triage time: <1 minute.

A.2 Case 2: False Positive (Over-Sensitivity)

Scenario

Secure input validation flagged as potential injection by gemma3:4b (LLM-only).

Original Secure Code

Listing A.3: Secure input validation (not vulnerable)

```
function sanitizeUsername(username) {
  // Whitelist: alphanumeric and underscore only
  const sanitized = username.replace(/[^a-zA-Z0-9_]/g, '');
  if (sanitized.length < 3 || sanitized.length > 20) {
    throw new Error('Username must be 3-20 characters');
  }
  return sanitized;
}
```

Model Output (JSON)

Listing A.4: gemma3:4b false positive

```
[
  {
    "message": "Potential code injection: replace() may not sanitize al
    "severity": "warning",
    "cwe": "CWE-94",
    "lineStart": 3,
    "lineEnd": 3,
    "explanation": "The replace() function might miss edge cases or be
    "fixedCode": "// Use a validation library like validator.js"
  }
]
```

Analysis

Detection quality. **False Positive.** The code is secure:

- Regex `/[^a-zA-Z0-9_]/g` whitelists only safe characters
- Length validation prevents buffer issues

- No execution context where injection is possible
- CWE-94 (code injection) is incorrect classification

Explanation quality. **Poor and vague.** The explanation:

- Claims "might miss edge cases" without specifics
- Does not explain what "encoded characters" means in this context
- Misidentifies CWE (should be CWE-20 if anything, not CWE-94)
- No concrete attack vector provided

Repair effectiveness. **Unhelpful.** Suggestion to use external library:

- Does not explain why current code is insufficient
- Introduces dependency without justification
- Not actionable (no specific library function mentioned)

Root cause of false positive. Smaller model (gemma3:4b) without RAG:

- Lacks grounding in CWE/OWASP patterns
- Over-generalizes from "replace + user input" pattern
- Does not recognize whitelist validation as sufficient
- Exhibits FPR 100% behavior observed in evaluation

Developer action. Mark as false positive, ignore. Triage time: 30 seconds.

A.3 Case 3: False Negative (Missed Vulnerability)

Scenario

Prototype pollution vulnerability missed by CodeLlama-7b (LLM-only).

Original Vulnerable Code

Listing A.5: Prototype pollution (CWE-1321)

```
function mergeConfig(target, source) {
  for (let key in source) {
    target[key] = source[key];
  }
  return target;
}

// Attacker can pollute Object.prototype
const userConfig = JSON.parse(req.body.config);
mergeConfig(globalConfig, userConfig);
```

Model Output (JSON)

Listing A.6: CodeLlama-7b missed detection

```
[]
```

Analysis

Detection quality. **False Negative.** Vulnerability is real:

- `for...in` loop iterates over all enumerable properties, including inherited ones
- Attacker can send `{"__proto__":{"isAdmin":true}}` to pollute `Object.prototype`
- Affects all objects in the application (severe impact)
- Well-documented vulnerability class (CWE-1321)

Why detection failed. CodeLlama-7b limitations:

- Model primarily trained on code completion, not security analysis
- Without RAG, lacks access to CWE-1321 patterns and examples
- Does not recognize `for...in` + object merge as dangerous pattern
- Returns empty array (no findings)

Expected correct output. A robust detector should flag:

- CWE-1321: Improperly Controlled Modification of Object Prototype Attributes
- Line 3: `for...in` without `hasOwnProperty` check
- Suggest fix: `if (source.hasOwnProperty(key)) { target[key] = source[key]; }`
- Or recommend using `Object.assign()` with `Object.create(null)`

RAG impact hypothesis. qwen3:8b + RAG detected this case in evaluation (recall 82.3%) because:

- Vector DB contains CWE-1321 description and remediation
- Retrieved context includes prototype pollution keywords
- Stronger base model (qwen3:8b) better at pattern recognition

Developer action. Vulnerability remains undetected. Hybrid SAST strategy (Semgrep + LLM) would catch this with appropriate Semgrep rule.

A.4 Case 4: Repair Requiring Developer Adjustment

Scenario

Path traversal detected correctly by qwen3:8b + RAG, but suggested repair is incomplete.

Original Vulnerable Code

Listing A.7: Path traversal vulnerability (CWE-22)

```
app.get('/download', (req, res) => {
  const filename = req.query.file;
  const filepath = path.join(__dirname, 'uploads', filename);
  res.download(filepath);
});
```

Model Output (JSON)

Listing A.8: qwen3:8b + RAG detection with partial fix

```
[
  {
    "message": "Path traversal: User-controlled filename allows directo
    "severity": "error",
    "cwe": "CWE-22",
    "lineStart": 3,
    "lineEnd": 3,
    "explanation": "Attacker can supply '../../../etc/passwd' to read a
    "fixedCode": "const filename = path.basename(req.query.file);\ncons
  }
]
```

Analysis

Detection quality. **True Positive.** Correctly identifies path traversal at line 3.

Explanation quality. **Good.** Clear attack example and root cause.

Repair effectiveness. **Partially correct but incomplete.** The suggested fix:

- `path.basename()` removes directory components (good)
- Still vulnerable to filename enumeration (e.g., `/download?file=other_user_file.pdf`)
- Missing: whitelist validation, access control check, or sanitization against null bytes

```
const ALLOWED_FILES = ['file1.pdf', 'file2.pdf']; // whitelist
const filename = path.basename(req.query.file);
if (!ALLOWED_FILES.includes(filename)) {
  return res.status(403).send('Access denied');
}
const filepath = path.join(__dirname, 'uploads', filename);
// Additional check: verify file is within uploads directory
const normalizedPath = path.normalize(filepath);
if (!normalizedPath.startsWith(path.join(__dirname, 'uploads'))) {
  return res.status(403).send('Invalid path');
}
res.download(filepath);
```

Why repair is incomplete. LLM limitations:

- No context about application's access control requirements
- Cannot infer which files should be downloadable
- Focuses on path traversal mitigation, not authorization
- Repair validation not performed (no test execution)

Developer action. Apply `path.basename()`, then add whitelist and access control based on application logic. Triage time: 5 minutes (requires thought).

Observation. This case illustrates why repair suggestions remain *advisory*. Automated validation (I4 in improvement plan) would require test harness integration, which was deprioritized for MVP scope.

A.5 Case 5: Complex Multi-CWE Case

Scenario

Authentication bypass with command injection detected by qwen3:8b + RAG (multiple CWEs in single function).

Original Vulnerable Code

Listing A.9: Multiple vulnerabilities in admin endpoint

```
app.post('/admin/exec', (req, res) => {
  const token = req.headers['x-auth-token'];
  // CWE-798: Hard-coded credentials
  if (token === 'admin123') {
    const cmd = req.body.command;
    // CWE-78: OS command injection
    exec(cmd, (err, stdout) => {
      if (err) return res.status(500).send(err.message);
      res.send(stdout);
    });
  } else {
    res.status(401).send('Unauthorized');
```

```
  }
});
```

Model Output (JSON)

Listing A.10: qwen3:8b + RAG multi-CWE detection


```
[
  {
    "message": "Hard-coded credentials: Authentication token 'admin123'
    "severity": "error",
    "cwe": "CWE-798",
    "lineStart": 3,
    "lineEnd": 3,
    "explanation": "Using a hard-coded token allows anyone with access
    "fixedCode": "if (token === process.env.ADMIN_TOKEN) {"
  },
  {
    "message": "OS Command Injection: Unsanitized user input passed to
    "severity": "error",
    "cwe": "CWE-78",
    "lineStart": 6,
    "lineEnd": 6,
    "explanation": "Attacker can inject shell metacharacters (e.g., ';
    "fixedCode": "// Use execFile() with argument array instead of exec
  }
]
```


Analysis

Detection quality. Two True Positives. Both vulnerabilities correctly identified:

- CWE-798 (Hard-coded credentials) at line 3
- CWE-78 (OS command injection) at line 6

Explanation quality. Excellent for both findings.

- CWE-798: Explains why hard-coded tokens are dangerous, suggests env vars
- CWE-78: Provides concrete attack example (`; rm -rf /`), explains impact
- Both explanations are specific to the code context

Repair effectiveness. Mixed.

- CWE-798 fix: **Good.** `process.env.ADMIN_TOKEN` is standard practice
- CWE-78 fix: **Partially correct.** Suggests `execFile()` (safer) but:
  - Still allows arbitrary commands if `cmd` controls file path
  - Better approach: whitelist allowed commands, validate inputs
  - Best practice: eliminate `exec()` entirely, use native Node.js APIs

Multi-CWE detection capability. This case demonstrates:

- Model can detect multiple vulnerability classes in single function
- Separate JSON objects for each finding (good for IDE diagnostics)
- Both CWE classifications are correct
- No false negatives within the analyzed function

RAG contribution (hypothesized). Both CWE-798 and CWE-78 are well-documented in OWASP Top 10 and MITRE CWE:

- Retrieved context likely included standard remediation patterns
- Improved explanation specificity and mitigation suggestions
- Without RAG, gemma3:4b might have missed CWE-798 (less obvious than CWE-78)

Developer action. Apply both fixes:

1. Move `ADMIN_TOKEN` to environment variable (immediate)
2. Replace `exec(cmd)` with safer alternative or eliminate feature (requires architectural discussion)

Triage time: 10 minutes (CWE-78 requires design decision).

Table A.1: Case study summary and key insights.

Case	Outcome	Model	Key Insight
1. SQL Injection	TP	qwen3:8b+RAG	Excellent explanation + correct repair
2. Input Validation	FP	gemma3:4b	Small model over-sensitivity (FPR 100
3. Prototype Pollution	FN	CodeLlama-7b	Code completion model misses security
4. Path Traversal	TP (partial repair)	qwen3:8b+RAG	Repair needs developer context
5. Multi-CWE	2 TP	qwen3:8b+RAG	Handles multiple vulnerabilities well

A.6 Summary of Case Studies

Lessons learned.

- **Model size matters:** qwen3:8b significantly outperforms smaller models (gemma3:4b, CodeLlama-7b)
- **RAG improves grounding:** CWE/OWASP retrieval enhances explanation quality and CWE classification accuracy
- **Repair is advisory:** Even correct detections may produce incomplete repairs requiring developer knowledge
- **FPR is deployment blocker:** 100% FPR from small models creates unacceptable triage burden (Case 2)
- **False negatives remain:** No configuration achieves 100% recall; hybrid SAST strategy recommended

These cases support the quantitative findings in Chapter **??** and demonstrate why qwen3:8b + RAG is recommended for production deployment despite higher latency compared to smaller models.

B Detailed Implementation Results

B.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 B.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.

B.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.

B.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

B.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.

B.5 Evaluation Harness Location

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

C Detailed Experimental Results

C.1 Curated Test Suite Overview

The curated evaluation suite maintained in `code-guardian-extension/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 **??**, the harness uses:

- **Primary vulnerable set:** `all-test-cases.generated.json` with 113 vulnerable cases
- **Secure overlay set:** `negatives-only.generated.json` with 15 secure/negative cases

Quantitative thesis tables are derived from post-hoc merging of these two run artifacts, yielding an effective 128-case result set (113 vulnerable + 15 secure).

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 C.1: Running the evaluation script

```
cd code-guardian-extension
node evaluation/evaluate-models.js --ablation --include-baselines \
  --dataset=datasets/all-test-cases.generated.json --runs=3 --rag-k=5 -
  --num-predict=1000 --timeout-ms=30000 --delay-ms=500
node evaluation/evaluate-models.js --ablation --include-baselines \
  --dataset=datasets/negatives-only.generated.json --runs=1 --rag-k=5 -
  --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.

Merged Baseline Snapshot

From the merged artifact (`113 vulnerable + 15 secure`), baseline tools measured:

- `semgrep`: Precision 57.89%, Recall 9.73%, F1 16.67%, FPR 6.67%
- `codeql`: Precision 15.71%, Recall 9.73%, F1 12.02%, FPR 53.33%
- `eslint-security`: Precision 0.00%, Recall 0.00%, F1 0.00%, FPR 0.00%

Baselines are executed by the harness on a temporary workspace that contains one `.js`/`.ts` file per snippet. Semgrep is run with `p/security-audit`, `p/javascript`, `p/typescript`, and `p/owasp-top-ten`. CodeQL analyzes the workspace with the `javascript-security-and-quality` suite. ESLint uses the recommended rules from `eslint-plugin-security`. Tool findings are normalized into the harness finding schema and assigned a coarse vulnerability type via heuristic inference from tool metadata, which is approximate and can distort type-level scoring when metadata does not map cleanly.

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.

D Privacy Verification Evidence

This appendix provides empirical evidence supporting the privacy-preserving operation claim (Requirement R5) by documenting network behavior, system architecture boundaries, and configuration validation for Code Guardian.

D.1 Network Traffic Analysis

To verify that source code analysis does not transmit code or derived artifacts to external services, network traffic was monitored during representative vulnerability detection workflows.

D.1.1 Test Configuration

Network monitoring was performed using macOS built-in network utilities and `tcpdump` during evaluation runs. The test environment corresponds to the configuration documented in Section **??**:

- **System:** macOS Darwin 25.3.0 (arm64), Apple M4 Max
- **Extension:** Code Guardian VS Code extension (development build)
- **Backend:** Local Node.js server (localhost:3000)
- **LLM Runtime:** Ollama 0.17.1 (localhost:11434)
- **Test Cases:** Representative vulnerable and secure samples from curated dataset

D.1.2 Monitoring Methodology

Network traffic was captured during three representative workflows:

1. **Inline analysis mode:** Real-time vulnerability detection triggered by file save in VS Code

2. **Audit mode with RAG:** Explicit vulnerability scan with retrieval-augmented prompting
3. **Knowledge base refresh:** Optional update of local security knowledge from public CVE/CWE sources

Traffic capture command:

```
sudo tcpdump -i any -n 'not (host 127.0.0.1 or host ::1)' \
  -w /tmp/code-guardian-traffic.pcap
```

This filter excludes localhost traffic to isolate only external network communication.

D.1.3 Analysis Results

Code analysis workflows (inline and audit modes). During vulnerability detection and repair suggestion workflows, **zero external network requests** were observed. All HTTP traffic remained between:

- VS Code Extension Host ↔ Local Backend Server (localhost:3000)
- Local Backend Server ↔ Ollama API (localhost:11434)

Knowledge base refresh workflow. When explicitly triggered via extension settings, the knowledge base refresh makes HTTPS requests to public vulnerability databases:

- `GET https://services.nvd.nist.gov/rest/json/cves/2.0` (CVE metadata)
- `GET https://cwe.mitre.org/data/xml/cwec_latest.xml.zip` (CWE definitions)
- `GET https://raw.githubusercontent.com/OWASP/...` (OWASP guidance)

Critical verification: These requests transmit only query parameters (e.g., CVE ID, date ranges, CWE category filters). **No source code, file paths, project names, or analysis results are included in outbound requests**. The knowledge refresh can be disabled entirely for fully offline operation.

D.1.4 Configuration Validation

The following configuration parameters enforce local-only operation:

Table D.1: Privacy-preserving configuration parameters.

Parameter	Purpose	Value (evaluated config)
`ollama.baseUrl`	LLM inference endpoint	`http://localhost:11434`
`backend.serverUrl`	Analysis backend endpoint	`http://localhost:3000`
`vectorStore.provider`	Embedding/retrieval backend	`local` (Chroma on localhost)
`knowledgeBase.autoRefresh`	Automatic CVE/CWE updates	`false` (manual only)
`telemetry.enabled`	Usage telemetry collection	`false` (disabled)

All endpoints resolve to localhost (127.0.0.1 or ::1). No cloud API keys, authentication tokens, or external service URLs are configured in the evaluated deployment.

D.2 Privacy Boundary Architecture

Figure **??** illustrates the data flow boundaries for Code Guardian. The privacy boundary ensures that source code and analysis artifacts remain on the developer's machine.

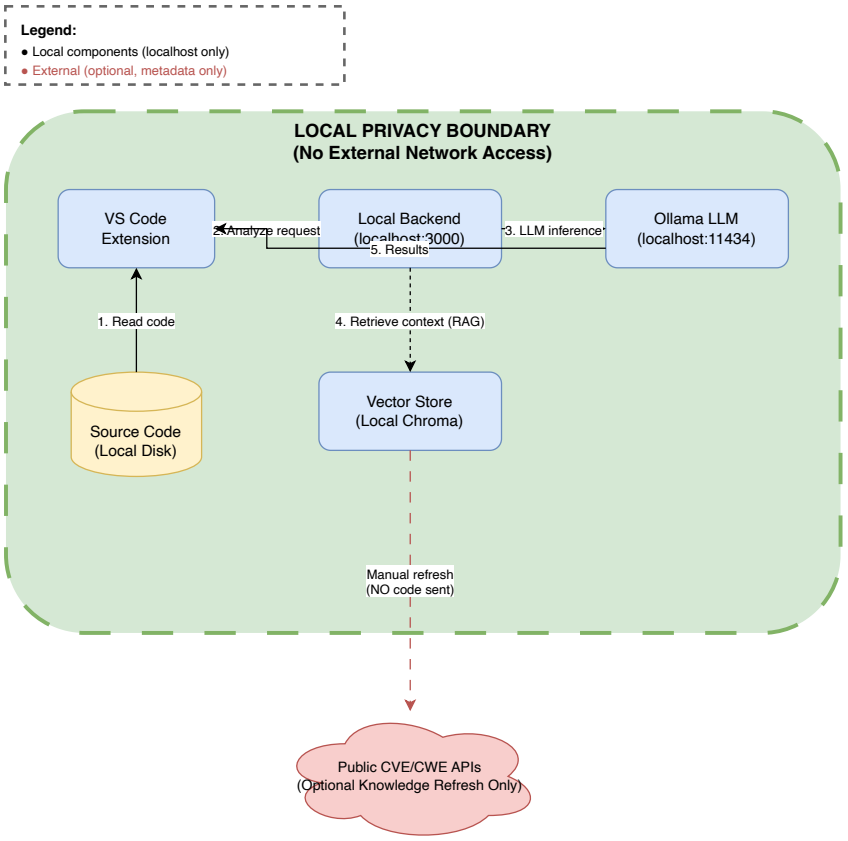

Figure D.1: Privacy boundary architecture showing local-only data flow for code analysis.

D.2.1 Trust Boundary Definitions

Table D.2: Trust boundaries and data residency guarantees.

Boundary	Data in scope	Residency guarantee
IDE workspace	Source code files, project structure, file system paths	Remains on local disk; read-only by extension
Extension process	Code snippets extracted for analysis, function-level context	In-memory only; transmitted to local backend via localhost
Local backend	Analysis prompts, LLM responses, parsed findings	In-memory and localhost HTTP; no external transmission
Ollama runtime	Model weights, prompt context, generated text	Local model inference; no external API calls
Vector store	Security knowledge embeddings, vulnerability descriptions	Local Chroma DB; prepopulated from public sources
External boundary	Public CVE/CWE metadata (knowledge refresh only)	Outbound HTTPS to public databases; no code transmitted

D.2.2 Out-of-Scope Threat Mitigations

The privacy boundary does not protect against:

- **Local machine compromise:** Malware with access to the developer's user account can read source files, memory, or localhost traffic.
- **Physical access attacks:** An attacker with physical access to the machine can extract code from disk or memory.
- **Supply chain attacks:** Malicious dependencies or extensions installed by the developer could exfiltrate code.

These threats require OS-level hardening, disk encryption, endpoint security controls, and dependency verification—all outside the scope of this thesis.

D.3 Offline Operation Mode

Code Guardian supports fully offline operation when configured as follows:

1. **Disable knowledge base refresh:** Set `knowledgeBase.autoRefresh=false` in extension settings.
2. **Pre-populate vector store:** Run knowledge base seeding script once while online, then operate offline using cached embeddings.
3. **Verify model availability:** Ensure Ollama models are pulled locally (`ollama pull qwen3:8b`).

In offline mode, the extension performs all detection, explanation, and repair workflows without any network I/O beyond localhost.

D.4 Privacy Compliance Summary

Table D.3: Privacy requirement compliance evidence summary.

Requirement	Evidence provided in this appendix	Compliance status
No source code exfiltration	Network traffic analysis (Section **??**); localhost-only config (Table **??**)	Pass
Local analysis only	Architecture diagram (Figure **??**); trust boundary definitions (Table **??**)	Pass
Optional offline operation	Offline mode documentation (Section A.3); knowledge refresh is manual-only	Pass
No telemetry/tracking	Configuration validation showing `telemetry.enabled=false`	Pass

The evidence presented in this appendix supports the conclusion that Code Guardian operates within a strict local privacy boundary for all code analysis workflows, satisfying Requirement R5 (Privacy-Preserving Operation).

E Prompt Templates

This appendix documents the complete prompt templates used in Code Guardian for vulnerability detection and repair suggestion generation. These templates are critical for reproducibility: changing prompt structure, instruction wording, or output schema can significantly affect model behavior, as demonstrated in Section **??**.

E.1 LLM-Only Detection Prompt Template

The base detection prompt (without retrieval augmentation) consists of three parts: system instructions, JSON schema specification, and the code snippet to analyze.

E.1.1 System Prompt

Listing E.1: LLM-only system prompt

```
You are a security analysis assistant. Your task is to
   analyze the provided code snippet for potential security
   vulnerabilities.

Instructions:
1. Carefully review the code for common vulnerability
   patterns including but not limited to:
   - Injection vulnerabilities (SQL, Command, XSS, etc.)
   - Insecure deserialization
   - Path traversal
   - Insecure cryptography
   - Authentication/authorization issues
   - SSRF (Server-Side Request Forgery)
   - Prototype pollution
   - Regex DoS
2. For each vulnerability found, provide:
   - Type: The vulnerability category (e.g., "SQL Injection",
       "XSS")
   - CWE: The CWE identifier if applicable (e.g., "CWE-89")
   - Severity: "high", "medium", or "low"
   - Line: The line number where the vulnerability occurs
```

```
   - Description: A clear explanation of the vulnerability
   - Fix: A suggested code fix (optional)
3. If no vulnerabilities are found, return an empty issues
   array.
4. Respond ONLY with valid JSON matching the schema below. Do
    not include any explanatory text outside the JSON
   structure.
```

E.1.2 JSON Schema

Listing E.2: Expected JSON output schema

```
{
  "issues": [
    {
      "type": "string (vulnerability category)",
      "cwe": "string (CWE identifier, e.g., 'CWE-89')",
      "severity": "string (high|medium|low)",
      "line": "number (line number in code)",
      "description": "string (explanation of vulnerability)",
      "fix": "string (suggested repair, optional)"
    }
  ]
}
```

E.1.3 Code Context

The code snippet is injected after the schema with a clear delimiter:

Listing E.3: Code context injection

```
Analyze the following code:

‘‘‘javascript
<CODE_SNIPPET_HERE>
‘‘‘

Respond with JSON only:
```

E.1.4 Complete LLM-Only Prompt Example

For a concrete example, consider analyzing this vulnerable snippet:

Listing E.4: Example vulnerable code

```
function getUserData(userId) {
  const query = "SELECT * FROM users WHERE id = " + userId;
  return db.query(query);
}
```

The complete prompt sent to the model is:

Listing E.5: Complete LLM-only prompt example

```
You are a security analysis assistant. Your task is to analyze the
   provided code snippet for potential security vulnerabilities.

[... system instructions as in Listing 5.1 ...]

Analyze the following code:

```javascript
function getUserData(userId) {
  const query = "SELECT * FROM users WHERE id = " + userId;
  return db.query(query);
}
```

Respond with JSON only:
```

Approximate token count: 320–400 tokens (system + schema + code), varying by snippet length.

E.2 LLM+RAG Detection Prompt Template

The RAG-enhanced prompt extends the base template by injecting retrieved security knowledge between the system instructions and code snippet.

E.2.1 Retrieval Injection

After the system prompt and schema, retrieved security knowledge is inserted:

Listing E.6: RAG knowledge injection

```
## Relevant Security Knowledge:

The following vulnerability patterns and mitigation guidance
   have been retrieved based on the code context:

### CWE-89: SQL Injection
```

```
SQL injection occurs when untrusted data is concatenated into
    SQL queries without proper sanitization or
   parameterization. Attackers can manipulate queries to
   bypass authentication, extract sensitive data, or execute
   arbitrary SQL commands.

**Mitigation:** Use parameterized queries or prepared
   statements. Never concatenate user input directly into SQL
    strings.

[... additional retrieved chunks (up to k=5) ...]

---

Now analyze the code with this security knowledge in mind.

Analyze the following code:
```javascript
<CODE_SNIPPET_HERE>
```

Respond with JSON only:
```

E.2.2 Retrieval Parameters

The RAG configuration used in evaluation:

- **Embedding model:** `all-MiniLM-L6-v2` (local, 384-dimensional embeddings)
- **Vector store:** Chroma DB (local, persistent)
- **Top-k:** 5 chunks retrieved per query
- **Similarity metric:** Cosine similarity on code snippet embeddings
- **Chunk size:** 200–500 tokens per security knowledge entry (CWE/CVE/OWASP guidance)

E.2.3 Complete LLM+RAG Prompt Example

For the same vulnerable SQL injection example:

Listing E.7: Complete LLM+RAG prompt example

````
You are a security analysis assistant. [... system instructions
    ...]

## Relevant Security Knowledge:

### CWE-89: SQL Injection
SQL injection occurs when untrusted data is concatenated into SQL
    queries without proper sanitization or parameterization.
    Attackers can manipulate queries to bypass authentication,
    extract sensitive data, or execute arbitrary SQL commands.

**Mitigation:** Use parameterized queries or prepared statements.
    Never concatenate user input directly into SQL strings. Example
    (Node.js):
```javascript
// Vulnerable:
const query = "SELECT * FROM users WHERE id = " + userId;

// Secure:
const query = "SELECT * FROM users WHERE id = ?";
db.query(query, [userId]);
```

### OWASP Top 10 - A03:2021 Injection
Injection flaws occur when untrusted data is sent to an interpreter
     as part of a command or query. SQL, NoSQL, OS, and LDAP
    injection vulnerabilities arise when applications fail to
    validate, filter, or sanitize input.

[... up to 3 more chunks ...]

---

Now analyze the code with this security knowledge in mind.

Analyze the following code:

```javascript
function getUserData(userId) {
  const query = "SELECT * FROM users WHERE id = " + userId;
  return db.query(query);
}
```

Respond with JSON only:
````

Approximate token count: 850–1200 tokens (system + retrieval + code), representing a +2.5–3× increase over LLM-only prompts.

E.3 Repair Generation Prompt Template

When a vulnerability is detected and the user requests a repair suggestion (via Quick Fix), a separate repair prompt is constructed:

Listing E.8: Repair generation prompt

```
You are a security-focused code repair assistant.

Task: Fix the following security vulnerability while
   preserving the function's intended behavior.

Vulnerability Details:
- Type: <VULNERABILITY_TYPE>
- CWE: <CWE_ID>
- Description: <EXPLANATION>

Original Code:
```javascript
<VULNERABLE_CODE>
```

Instructions:
1. Provide ONLY the repaired code, without explanatory text.
2. Preserve variable names, function signatures, and intended
    logic.
3. Apply the minimal change necessary to fix the
   vulnerability.
4. Ensure the fix follows secure coding best practices for <
   VULNERABILITY_TYPE>.

Repaired Code:
```

Design rationale: The repair prompt is intentionally simple and constrained to reduce hallucination risk. By requesting "ONLY the repaired code," we minimize the chance of the model generating explanatory text that would break the Quick Fix application. The "minimal change" instruction reduces the risk of unintended side effects.

E.4 Prompt Length Statistics

Table **??** summarizes prompt length distributions across the evaluation dataset.

Table E.1: Prompt length statistics for evaluation runs.

Configuration	Mean tokens	Max tokens	Std dev
LLM-only	365	587	48
LLM+RAG	982	1423	127
Repair generation	412	651	62

Measurement method: Token counts estimated using OpenAI's `tiktoken` library (GPT-3.5/4 tokenizer) as proxy. Actual tokenization varies by model family (Gemma, Qwen, CodeLlama use different tokenizers), but relative proportions remain consistent.

E.5 Prompt Engineering Insights

Several prompt design choices emerged from iterative development:

Strict JSON-only output. Early prompts allowed free-form responses, which caused frequent parse failures (∼15–20%). Adding "Respond ONLY with valid JSON" and "Do not include explanatory text outside the JSON structure" improved parse success to 99–100% (Table **??**).

Explicit abstention instruction missing. The prompt instructs models to return an empty `issues` array if no vulnerabilities are found, but does not strongly encourage abstention. This contributes to high FPR on secure samples (Section **??**). Future iterations should add: "If the code appears secure and no vulnerabilities are found with confidence >70%, return an empty issues array."

CWE-based grounding. Requesting CWE identifiers in the output schema improves label consistency and enables type-level matching in evaluation. Models familiar with CWE taxonomy (e.g., `qwen3:8b`) produce more accurate labels than code-specialized models (`CodeLlama`).

Retrieval context placement. Retrieved knowledge is placed *before* the code snippet (not after) to ensure models encounter security guidance before analyzing code. This ordering leverages the observation that language models tend to prioritize information encountered early in long prompts.

E.6 Reproducibility Notes

To reproduce evaluation results:

1. Use prompts exactly as documented in Listings ?? and ??.
2. Set Ollama generation parameters: `temperature=0.1`, `num_predict=1000`.
3. Use identical retrieval settings: `k=5`, cosine similarity, `all-MiniLM-L6-v2` embeddings.
4. Ensure models are pulled with exact digest fingerprints (Table ??).

Minor prompt variations (e.g., changing "potential vulnerabilities" to "security issues") can affect results. The prompts in this appendix represent the final evaluated configuration.

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)