Does Retrieval-Augmented Vulnerability Repair Generalize Beyond Snippets? A Study on TypeScript Web Applications

Breno Vergílio
Faculty of Computing, UFMS
Campo Grande, MS, Brazil
breno.vergilio@ufms.br

Matheus Mota
Faculty of Computing, UFMS
Campo Grande, MS, Brazil
m.bernard@ufms.br

Awdren Fontão
Faculty of Computing, UFMS
Campo Grande, MS, Brazil
awdren.fontao@ufms.br

Abstract

Retrieval-Augmented Generation (RAG) has been proposed to improve the security of Large Language Model (LLM)-generated code, with prior work reporting fix rates above 70% on curated vulnerability snippets. Whether these gains transfer to repository-level auto-patching, particularly in resource-constrained deployments, remains unexplored. We evaluate five RAG-based techniques (SOSecure, two variants of our adaptation POSecure, a CVEfixes-derived variation, and a No RAG baseline) on the OWASP Juice Shop, a vulnerable TypeScript application, using both Llama 3.1 8B (locally) and Llama 3.3 70B (on cloud), assessed through static analysis, code quality, and operational cost. Across 351 analyzed functions, none of the techniques improved security in a meaningful way *in our study setting*: static findings either increased or remained effectively unchanged, while code quality consistently degraded (code smells up to +81%, technical debt up to +106%) and operational cost rose substantially. These effects were most severe in the local 8B setting, where repeated healing attempts amplified token usage without yielding stable repairs. Our results suggest that the gains reported in snippet-level evaluations may not transfer to repository-level auto-patching, and that the cost-benefit balance of RAG-based vulnerability repair deteriorates further under resource constraints. We call for further studies across other settings.

Keywords

Large Language Models, Retrieval-Augmented Generation, Vulnerability Repair, Empirical Study, Prompt Compression, Code Quality

1 Introduction

Security is a central concern in modern software engineering, with new vulnerabilities being constantly discovered and exploited [38]. Generative AI has emerged as a widely adopted aid for software engineering tasks such as code generation and testing [39], but concerns remain about the security of its outputs. Perry et al. [35] show that developers using AI assistants tend to produce more insecure code while expressing higher confidence in its correctness. Pearce et al. [33] report that up to 40% of the code suggested by *GitHub Copilot* contains vulnerabilities, and Negri-Ribalta et al. [28] conclude that AI-generated code frequently contains security flaws.

This problem is exacerbated by the dynamic nature of cybersecurity: thousands of new CVEs are disclosed every year [26], and approaches based solely on static training data may become outdated over time [33]. Since retraining LLMs is costly and often impractical, Retrieval-Augmented Generation (RAG) has been proposed to incorporate external, up-to-date information without retraining. SOSecure [26] leverages Stack Overflow (SO) discussions to generate patched versions of insecure code, while Vul-RAG [6] uses CVE databases to support security analysis without generating patches. Although these approaches demonstrate the potential of combining LLMs with external knowledge sources, it remains open whether results obtained on curated vulnerability snippets generalize to repository-level auto-patching, where compilation, dependency consistency, maintainability, and operational cost become first-class constraints.

In practice, these approaches are typically evaluated in controlled environments and do not account for realistic software engineering constraints, such as compilation requirements, integration with existing code, and limited computational resources. These factors are critical for adoption in resource-constrained environments, where limited budget and infrastructure make locally hosted, smaller models a realistic option, and where the trade-offs between model size, performance, and operational cost remain underexplored [45]. To make "resource-constrained" concrete and ground it in a real population, we adopt as a representative example the setting of small and medium-sized software enterprises (SSMEs) in emerging countries, where cost and infrastructure constraints are particularly significant [19, 36].

To illustrate, consider a function that unpacks uploaded ZIP archives. At the snippet level, a technique like SOSecure retrieves a Stack Overflow discussion and is judged on whether the isolated function looks more secure. In a real repository, the same patch must also compile against surrounding type definitions, preserve the function's contract, and integrate with the framework, while the retrieved discussion can add thousands of tokens to each request, inflating operational cost. A retrieved discussion may even contain a stronger check than the original code, yet the model can still preserve the original flawed logic and only add boilerplate, yielding a patch that compiles, and is therefore accepted, without fixing the vulnerability. Our Supplementary Material [46] shows one such case end-to-end.

To address this, we investigate the applicability of RAG-based approaches for automatic vulnerability repair in a realistic software repository. We evaluate SOSecure on the OWASP Juice Shop [18], a deliberately vulnerable TypeScript application, under two deployment scenarios: a resource-constrained setting with a local Llama 3.1 8B model, and a cloud-based setting with a Llama 3.3 70B model. Our preliminary findings indicate that SOSecure substantially increases token consumption and operational cost. Compilation errors and broken application functionality were also frequent, but occurred even without SOSecure, suggesting a broader challenge of applying LLM-based repair to integrated codebases rather than a property of SOSecure itself. From a software engineering perspective, our contribution is an empirical assessment of this

snippet-to-repository generalization gap, showing that repository-level vulnerability repair imposes constraints largely absent from snippet-level evaluations, and that these constraints materially change the observed cost-benefit profile of RAG-based repair in the setting we studied.

To mitigate these limitations, we propose **POSecure** (PackOverflow Secure), an adaptation of SOSecure that incorporates prompt compression to reduce unnecessary context and a 1-hop dependency graph to improve code consistency and compilability. To further investigate how the choice of retrieval source affects the outcome, we also evaluate a variation that replaces SO discussions with curated vulnerability fixes from the CVEfixes dataset [2]. We evaluate POSecure and the CVEfixes variation against SOSecure and a No RAG baseline. Our results show that none of the techniques improved the security of the patched repository, while all of them introduced regressions in code quality and raised operational cost.

2 Background

2.1 Large Language Models (LLMs) and Small Language Models (SLMs)

LLMs are pre-trained models with large parameter sizes that require substantial computational resources [4, 15]. As a result, they are typically deployed via cloud services, which introduces monetary cost, latency, and potential privacy concerns.

To mitigate these limitations, smaller models (SLMs) have been proposed as a more resource-efficient alternative. SLMs require fewer computational resources and can be deployed locally, making them more suitable for constrained environments such as SSMEs, while still achieving competitive performance in domain-specific tasks [47]. There is no universally accepted threshold for defining SLMs [47]; in this work, we consider models with up to 8B parameters as SLMs.

2.2 Retrieval-Augmented Generation (RAG) and k-hop Dependency Graphs

RAG is an approach used to provide external context to a language model based on a query without needing to retrain it, helping to prevent hallucinations and improving precision of responses [20]. However, traditional RAG can face limitations with queries that require deeper context understanding of the dataset. Knowledge Graphs have been proposed to mitigate these limitations, where nodes represent entities and edges represent relationships between these entities in the dataset [7]. A k-hop Dependency Graph means a graph with entities located within k hops away from the original node (e.g., type signatures and library contracts immediately invoked by the vulnerable function) [29].

2.3 Prompt Compression

When prompting a model with RAG or other techniques (e.g., chain-of-thought), the final prompt may get lengthy [16]. A lengthy prompt leads to more expensive cloud bills, more computational power needed and may exceed the model's context window, which remains a problem even in local deployments. One possible mitigation to this problem is **prompt compression**, which can be either **extractive**, **abstractive** or **hybrid** [8, 49]. Extractive compression selects the most important sentences in the input document(s) and concatenates them. Abstractive compression generates a summary synthesizing information from multiple retrieved documents with sentences that are different than the original sentences. Hybrid compression combines both.

In the case of SOSecure, the use of full Stack Overflow discussions and multiple model queries further amplifies this issue. These limitations motivated the design of POSecure, which reduces token consumption while preserving essential security context. Here, prompt compression acts not only to lower cost, but also to eliminate noisy retrieved content that may destabilize patch generation.

2.4 Static and Dynamic Application Security Testing (SAST and DAST)

SAST and DAST are complementary approaches to test application security. In SAST, the source code, bytecode or binaries are tested without running the application. SAST tools usually operate in the early stages of development lifecycle, allowing early detections. DAST consist of analyzing a running application to detect vulnerabilities, usually in a black-box manner, simulating attacks from an external user's perspective [40].

SAST tools usually operate in the early stages of development lifecycle, allowing early detections. As our approaches aim to fix source code vulnerabilities in a white-box scenario, we use CodeQL [1], a widely used query based SAST tool [10, 11, 21, 26], to analyze Juice Shop vulnerabilities before and after the application of each given technique applied in this paper.

2.5 CVEs and CWEs

According to CVE official website [5], Common Vulnerabilities and Exposures (CVE) Program's mission is to "identify, define, and catalog publicly disclosed cybersecurity vulnerabilities". Each CVE has a CVE ID, which uniquely identifies it. For example, CVE-2021-44228 identifies the Log4shell vulnerability, one of the most significant cybersecurity vulnerabilities in the modern age [9].

A Common Weakness Enumeration (CWE), according to their FAQ [25], is a "formal list or dictionary of common software and hardware weaknesses that can occur in architecture, design, code, or implementation that can lead to exploitable security vulnerabilities". CWEs also have a unique identifier. While a CVE relates to a specific vulnerability, a CWE can be used to classify a group of vulnerabilities with characteristics in common. For example, while CVE-2021-44228 relates only to Log4shell, CWE-20 (Improper Input Validation) and CWE-502 (Deserialization of Untrusted Data) are used to classify it [27].

To connect these identifiers to practical vulnerability analysis, datasets such as CVEfixes [2] are available. Specifically, CVEfixes compiles vulnerable functions (JavaScript and TypeScript in our study), their corresponding fixes, and the related CVE and CWE identifiers. In our approach, the function under analysis queries this repository to retrieve similar vulnerable functions. Both the retrieved code and its patch are then provided as context. It is worth noting that no compression is applied during this step, as

[1] https://codeql.github.com/

the retrieved content consists strictly of source code rather than natural language discussions.

3 Related Works

LLMs for vulnerability repair with security knowledge. Vul-RAG [6] builds a knowledge base from CVEs and patched code for vulnerability detection; LSAST [17] integrates LLMs with SAST tools to guide vulnerability identification; GRASP [32] structures Secure Coding Practices as a graph to guide secure generation; and SOSecure [26] retrieves Stack Overflow discussions as context to improve insecure code. We differ by adding local context awareness (dependency graphs and type interpretation) and prompt compression, and by evaluating the cost-benefit trade-off under constrained resources.

Repository-level code understanding and generation. Recent benchmarks show that LLMs struggle beyond isolated snippets: A.S.E [22] targets the security of AI-generated code at repository level, and SWE-QA [34] shows snippet-based setups miss cross-file reasoning and long-range dependencies. On the tooling side, RepoScope [23] retrieves structure-guided multi-view context for code generation, and EnAgent [43] tackles issue resolution via agent-based ensemble reasoning. We share this repository-level concern but target retrieval-augmented *vulnerability repair* and its cost-benefit trade-off.

Smaller models and deployment constraints. Hasan et al. [14] evaluate 20 open-source SLMs (0.4B–10B) across five benchmarks, finding that compact SLMs reach competitive accuracy at far lower resource demand, while larger ones need up to 4× more VRAM for marginal gains; TinyLLM [13] characterizes latency, memory, and accuracy trade-offs of sub-3B models on edge devices. Both use isolated benchmarks, leaving open how such efficiency considerations interact with retrieval pipelines and repository-level vulnerability repair.

4 Motivation

Security vulnerability fixes in Automatic Program Repair (APR) is an important but under-researched area [1]. For simplicity, we refer to this task as **auto-patch**. We aim to contribute to this domain by exploring lightweight methods that would be accessible to resource constrained environments, like SSMEs in emerging countries. We found SOSecure [26] a promising technique for generating secure code using LLMs and SO community knowledge. SOSecure filters SO discussions using security-specific and language-specific keywords to build a security-focused knowledge base, then queries it with a code snippet to retrieve related discussions and use them as RAG context for a model.

We had the following concerns about the technique: (1) It relies on the proprietary GPT-4o-mini, whose parameter count has not been disclosed. We aim to evaluate the approach using an open-weight model, more suitable for SSMEs in emerging countries [36]. (2) It was evaluated only for Python and C; we aim to assess its applicability to TypeScript, a more recent language for which the original authors note the approach may be less effective. (3) Using entire SO discussions may consume unnecessary tokens and bring irrelevant context. (4) Its impact on code quality remains unexplored, motivating us to analyze code smells and technical debt.

To scrutinize these points, we conducted an early exploratory implementation of SOSecure with JavaScript/TypeScript-related keywords, using Meta's open-weight **Llama 3.3 70B** on the cloud, applied to a vulnerable repository[2]. This implementation already exhibited substantial token and inference cost overhead relative to a No RAG baseline (see an end-to-end example in our Supplementary Material [46]), frequent compilation breaks (independently of whether SOSecure was used), and no clear improvement in the security of the patched repository. We treated these observations as design signals for adapting SOSecure to a repository-level setting and implemented two changes: (1) compression of SO discussions to reduce token usage, inference cost, and irrelevant context, using both an extractive compressor [31] and an abstractive approach (Subsection 5.4); (2) a 1-hop Dependency Graph that extracts type signatures of imported symbols and uses them as additional context to reduce compilation breaks. We named our adaptation **POSecure (PackOverflow Secure)**. To check the data source impact, we also derived a dataset of vulnerable code and its fixes using CVEfixes [2] as base.

The remainder of this paper presents the consolidated study comparing all five techniques (No RAG, SOSecure, both POSecure variants, and the CVEfixes variation) in terms of integrity, security, quality, and cost. Section 5.4 properly present each technique in detail. The effects observed in our early implementation are confirmed by these results: SOSecure on the 70B model required approximately 1.24M additional input tokens per run (+348%) and a 264% cost increase compared to No RAG, while increasing technical debt by 72% and not reducing static security vulnerabilities.

5 Research Method

In this section, we define our goal, Research Questions (RQs) and explain the Research Method adopted. We structure our main goal using the Goal-Question-Metric (GQM) framework [3]. Our goal is to evaluate **the viability of RAG-based techniques for repository-level vulnerability auto-patching** with the purpose of **analyzing which one is the most viable solution to repository vulnerability auto-patch** with respect to **integrity, security, quality and costs metrics** from the point of view of real world, resource constrained SE environments (such as **SSMEs in emerging countries**) in the context of **Automatic Program Repair for security**. Based on our main goal, we define the following RQs:

RQ1: How do the different techniques affect the structural integrity and compilability of the target application? Rationale: A vulnerability patch is only useful if the patched code still compiles and integrates into the project. RQ1 establishes whether each technique meets this precondition.

RQ2: Which technique achieves the highest efficacy in remediating static vulnerabilities? Rationale: The primary goal of RAG-based vulnerability repair is to reduce security flaws. RQ2 evaluates whether the techniques deliver on this promise in a realistic repository.

RQ3: To what extent do the generated security patches degrade the overall code quality and maintainability? Rationale: Patches that fix vulnerabilities at the cost of significant

[2] https://github.com/juice-shop/juice-shop

quality degradation are unlikely to be adopted. RQ3 checks whether the techniques preserve maintainability.

RQ4: What is the cost-benefit trade-off of each technique regarding execution overhead and financial budget? Rationale: Adoption is constrained by budget and infrastructure. RQ4 measures whether each technique is affordable in such contexts.

The following Subsections explain the procedures and decisions we adopted to address the RQs. Figure 1 depicts our Research Method, grouping it in three stages: (1) SO and CVEfixes dataset preparation for RAG context, (2) patch generation, where the vulnerable snippets are processed by the model together with external retrieved context and (3) analysis of the results through the perspective of security, quality and cost by their respective evaluation metrics (see Subsection 6.3).

5.1 Scenario Definition

We use SSMEs in emerging countries as an example of resource constrained SE environments throughout this work. We define two scenarios that SSMEs are likely to face, designed to reflect real-world challenges. **(1) Fully Local Environment**: an SSME has access only to local hardware (described in Subsection 6.1) and uses a locally-deployed model for patch generation. At submission time, the local hardware used can be bought for less than $1,500 after direct conversion from BRL (Brazilian currency, where $1 ≈ R$5.31), making it plausibly accessible for some SSMEs in emerging countries, particularly compared to the recurring cost of large-scale cloud usage. **(2) Local Environment with a Cloud LLM**: the SSME has access to the same local hardware but also relies on a cloud LLM for patch generation. Even so, cloud usage is not unlimited: SSMEs typically operate under tight budget constraints and usage quotas, which restrict request volume and make cost-efficiency a concern when selecting a technique. Our results show that the budget required for token consumption, in average, ranged from $0.36 to $1.31.

5.2 RAG Data Source Generation

To obtain security-related and JavaScript/TypeScript-related discussions in SO, we downloaded the Stack Exchange Data Dump from December 31, 2025 [41]. We followed the steps documented by the authors of SOSecure [26] with keyword modifications to the target languages (e.g., `prototype pollution`). To obtain the CVEfixes [2] dataset variation containing only vulnerable code, its fixes and CVE/CWE identifiers, we performed Structured Query Language (SQL) queries on the original dataset and saved the results.

To retrieve data, we followed Mukherjee and Hellendoorn [26] and Du et al. [6] and used BM25 [37], a bag-of-words model that considers term frequency and inverse document frequency. It is widely used in search engines due to its efficacy and efficiency [6], including SE retrieval tasks [26]. We retained BM25, rather than embedding-based or hybrid retrieval, to preserve fidelity with the original SOSecure implementation, which we evaluate as a baseline. Keeping the retrieval strategy fixed across all treatments also isolates the effects we set out to study, prompt compression (POSecure) and knowledge-base substitution (CVEfixes), from variations in retrieval.

5.3 1-hop Dependency Graph Extractor Implementation

One of the issues identified during exploratory testing was that the LLM frequently broke the code in multiple ways (e.g., supposing a function or variable exists in the context). To mitigate this, we implemented a Node.js[3] server that receives the function under analysis and its file path. With these, it retrieves all the function signatures and type declarations used inside that function to be used as context to the model. The server operates over the TypeScript Abstract Syntax Tree (AST) and its type/symbol information via the official TypeScript Compiler API. Unlike a purely syntactic parser such as Tree-Sitter [4] (used only for function extraction, Subsection 6.2), it leverages the compiler's semantic model, with fully resolved types rather than raw syntax. It identifies imported symbols, looks for their resolved types, and resolves aliases to their original definitions. Extraction stays single-hop because only directly imported symbols are reported, though their types are resolved against the full program context.

5.4 Treatments Definition

We used each technique mentioned in Section 4 as a treatment, along with a baseline without RAG. All treatments use the 1-hop Dependency Graph Extractor (Subsection 5.3) to give the model context about used functions and types. The treatments are summarized in Table 1.

We based our prompts on the SOSecure replication package, but adjusted them so that the model outputs only code, without explanations. This reduced token usage and simplified replacing the original function with the model's output. Implementation details are in Section 6. The treatments are designed to isolate the effect of retrieval source and context compression while keeping dependency-based compilation support consistent across configurations. Our treatments are summarized in Table 1.

5.5 Repository Auto-Patch Flow

We selected Juice Shop [18] as our target repository. Juice Shop is an open-source, intentionally vulnerable web application maintained by OWASP[5] contributors. It is written in JavaScript/TypeScript with Node.js, Express [6], and Angular [7], and covers all OWASP Top 10 vulnerabilities [30]. The application is organized as a set of challenges, each completed when the user exploits a corresponding vulnerability.

Juice Shop contains both the backend and frontend in the same repository. We narrowed our analysis to the backend (`models` and `routes` folders) because it contains most of the project's vulnerabilities. To save computational resources, we first visit each function to be analyzed and cache its retrieved context in a JSON file, using the filename and MD5 hash of the function as keys for `O(1)` access. For treatments that use context compression, only the compressed context is stored.

[3] https://nodejs.org/
[4] https://tree-sitter.github.io/tree-sitter/
[5] https://owasp.org/
[6] https://expressjs.com/
[7] https://angular.dev/

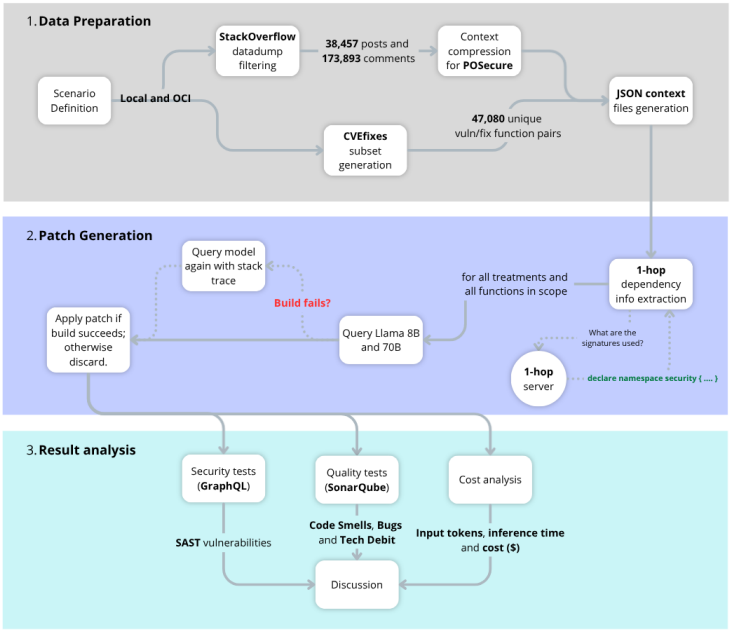

Figure 1: Overview of the research method.

Table 1: Summary of the evaluated treatments and their distinguishing characteristics.

Treatment	Retrieval Source	Context Processing	Additional Models	Purpose
No RAG	None	None	None	Baseline without retrieval augmentation.
SOSecure	Stack Overflow discussions	Raw retrieved discussions	None	Reproduces the original SOSecure strategy by injecting retrieved Stack Overflow content directly into the prompt.
Extractive POSecure	Stack Overflow discussions	LLMLingua-2 [31] extractive compression	None	Reduces prompt size by retaining only the most relevant portions of the retrieved discussions.
Abstractive POSecure	Stack Overflow discussions	LLM-generated abstractive summaries	Llama 3.2 3B (see Subsection 6.2)	Compresses retrieved discussions into concise summaries before patch generation.
CVEfixes	CVEfixes dataset	None	None	Provides vulnerability-fix examples from real-world commits instead of Stack Overflow discussions.

We adopt compilability as the minimum acceptance criterion for patch integration, recognizing that it does not guarantee runtime correctness or exploitability reduction. Each treatment is applied 3 times, and all reported metrics are the mean of these 3 executions. To mitigate compilation errors, we also used strict prompts (e.g., `STRICT TYPE PRESERVATION: Do not change the original nullability of variables or return types`). Details about the prompts used and an end-to-end example are available at our Supplementary Material [46].

We visit every file inside the `models` and `routes` folders that do not exclusively contain code related to Juice Shop challenges, since such files only manage internal challenge mechanics rather than

the realistic application logic. From each remaining file, we extract the individual functions to be patched, excluding those tied to the challenge-solving logic to reduce information leakage about the vulnerabilities. For every patch returned by the model, we check whether it breaks the project's compilation. If so, we prompt the model again with the previously generated patch and the compiler's stack trace. If the compilation is corrected, we accept the patch; otherwise, we discard it, record a failure, and proceed to the next function.

6 Study Setup

In this Section, we describe the hardware, models and tools to implement all the treatments and run the study. We also report the evaluation metrics collected.

6.1 Hardware Configuration and Cloud Infrastructure

All studies were conducted on a Windows 11 Pro 64 bit workstation equipped with an Intel Xeon E5-2686 v4 CPU (36 cores @2.30 GHz), 24 GB of system RAM, and an NVIDIA GeForce RTX 4060 GPU (8 GB dedicated VRAM, 12 GB shared). While on a Windows machine, Windows Subsystem for Linux (WSL) 2 with Ubuntu and Python3.11 was used to run all the studies.

The Cloud Infrastructure used was Oracle Cloud Infrastructure (OCI). OCI is a "globally available hyperscale cloud with all the services needed for enterprise workloads, including GenAI" [24]. We used OCI due to the availability of a managed endpoint compatible with our target model and to institutional access through cloud credits.

6.2 Models and Tools Used

We used **Llama 3.3 70B** (OCI) and **Llama 3.1 8B** (local) as patch-generation models, and **Llama 3.2 3B** (local) to compress SO discussions in Abstractive POSecure. We do not aim to identify the best model for vulnerability repair, but to observe how RAG-based techniques behave at different model capacities within the same family. We selected three Llama variants because they (1) share the same family, isolating model size from training methodology; (2) are open-weight, fitting the resource-constrained scenarios in Section 5; and (3) perform well on code and text tasks [12].

All models used `temperature`=0.0 for reproducibility. The 70B model used `max_tokens`=2048 and `top_p`=1.0; the local 8B[8] and 3B models used `num_ctx`=24576 to fit the function, retrieved context, and type signatures in one prompt, and `keep_alive`=-1 to keep the model in VRAM. Local models were served with Ollama.

For patch generation and analysis we combined several tools: LLMLingua-2 [31] for extractive prompt compression in Extractive POSecure; TreeSitter for AST-based function extraction; and Qdrant[9] with BM25 [37] for document storage and retrieval. Generated patches were evaluated through an automated pipeline: CodeQL (*javascript-security-extended.qls*) for SAST, and SonarQube[10] for code quality. Both were scoped to the target directories (`/routes`, `/models`), excluding test files, to isolate the LLM's impact on the application's core logic.

6.3 Evaluation Metrics

In this study, security effectiveness is operationalized through static findings only, given the inability to obtain stable runtime security measurements across treatments. We group metrics into four categories. *Integrity* captures whether patches are usable: *Valid Patches* are functions whose patch compiles (on the first pass or after a self-healing iteration with compiler feedback), while *Failed Healings* are functions whose patch still fails to compile after self-healing and are restored to their original state and excluded from downstream analysis. *Security* is measured by *SAST Vulnerabilities*, the static flaws detected by CodeQL in the backend logic (`/routes`, `/models`). *Quality* combines *Code Smells* and *Logical Bugs* (maintainability and reliability issues detected by SonarQube) and *Technical Debt* (estimated minutes to remediate them). *Budgetary* covers *Input Tokens* sent across all passes (measured with the Llama 3.1 8B tokenizer[11]), inference *Time* in seconds (excluding compilation, I/O, and retrieval), and estimated *Cost* based on Llama 3.3 70B OCI pricing[12] ($0.0018 per 10,000 chars, accessed 2026-05-03).

7 Results

This section presents the results obtained from the study evaluation. All reported values correspond to the mean across three independent executions for each treatment. Results are organized according to the defined research questions. A total of 351 functions from the Juice Shop repository were analyzed.

7.1 RQ1: Structural Integrity and Compilability

We analyzed valid patches and failed healings per technique (Table 2). We stress that a "valid patch" (Section 6) is one accepted by our pipeline, i.e., one that compiles on the first pass or after self-healing; it does not imply the patch is correct, secure, or semantically equivalent.

The two model scales behave very differently. On the 8B model, retrieval sharply reduces compilability: **No RAG** yields the most valid patches (153.00), while **SOSecure** collapses to 19.67 (−87.1%) with the highest failed-healing count (+70.5% over the baseline). The **POSecure** variants stay close to the baseline (−14 to −16%), and **CVEfixes** falls in between (−49.7%). On the 70B model, this effect largely disappears: all five techniques cluster within roughly ±2.5% of the baseline in valid patches and within 10% in failed healings. In short, the cost of retrieved context to compilability is severe at 8B but negligible at 70B, a scale dependence we return to in the Discussion.

7.2 RQ2: Security Effectiveness

Security effectiveness was evaluated using the SAST vulnerabilities reported by CodeQL. For the 8B model, the **Original repo** contained 46.00 SAST vulnerabilities. The **No RAG** approach increased this number to 52.33 (+13.8%), representing the worst result among all techniques. **Extractive** and **Abstractive POSecure** also

[8] GGUF Q5_K_M served via Ollama: https://huggingface.co/bartowski/Meta-Llama-3.1-8B-Instruct-GGUF.
[9] https://qdrant.tech/
[10] https://www.sonarsource.com/products/sonarqube/
[11] https://huggingface.co/unsloth/Meta-Llama-3.1-8B-Instruct
[12] https://www.oracle.com/artificial-intelligence/enterprise-ai/pricing/

Table 2: Summary of All Metrics for Llama 3.1 8B and Llama 3.3 70B

Model	Technique	Val. Pat.	Fail. Heal.	SAST	Bugs	Smells	Debt (min)	Inp. Tokens	Time (s)	Cost ($)	Cost Δ (%)
8B	Original repo	—	—	46.00	0.00	40.00	380.00	—	—	—	—
	No RAG	**153.00**	**190.67**	52.33	2.00	65.33	598.00	**375,329.00**	**5,347.94**	—	—
	SOSecure	19.67	325.00	46.00	**1.00**	**53.33**	**475.67**	2,725,239.67	12,038.67	—	—
	Ext. POSecure	129.00	216.67	50.00	1.67	62.00	591.33	650,096.00	5,847.44	—	—
	Abs. POSecure	131.00	216.00	48.67	5.67	65.33	570.33	468,325.33	5,898.32	—	—
	CVEfixes	77.00	255.33	**45.67**	2.00	55.00	489.67	1,632,678.00	9,926.02	—	—
70B	Original repo	—	—	46.00	0.00	40.00	380.00	—	—	—	—
	No RAG	193.00	**141.00**	50.00	2.00	55.67	656.00	**354,616.00**	**1,964.96**	**0.36**	ref.
	SOSecure	191.33	143.33	51.67	3.33	72.33	**653.67**	1,589,986.33	2,267.72	1.31	+259.4
	Ext. POSecure	194.00	155.00	**48.00**	13.00	52.33	784.67	574,904.33	2,093.39	0.59	+61.4
	Abs. POSecure	190.67	151.33	49.67	7.00	53.67	705.67	429,180.33	2,119.54	0.44	+20.8
	CVEfixes	**197.67**	144.33	51.67	**1.00**	**49.67**	655.00	1,132,510.33	2,087.16	0.98	+169.8

Cost Δ is relative to **No RAG** of the same model. **Original repo** has no cost data. Best value per metric within each model is highlighted in bold (excluding **Original repo**, which is reported as a reference baseline). Cost ($) is reported only for the 70B model, as it was used via OCI; the 8B model ran locally, incurring no cloud costs.

increased the count, reaching 50.00 (+8.7%) and 48.67 (+5.8%), respectively. **SOSecure** matched the baseline (46.00, +0.0%), and **CVEfixes** was the only technique to slightly reduce it (45.67, −0.7%). However, the lower vulnerability counts observed for techniques with high failed-healing rates must be interpreted with caution. They reflect a backend in which most functions remained unmodified.

For the 70B model, all techniques increased the number of detected vulnerabilities relative to the **Original repo**. **No RAG** increased this to 50.00 (+8.7%). **SOSecure** and **CVEfixes** reached 51.67 (+12.3%), representing the worst results. **POSecure** variants slightly mitigated this increase, with **Extractive POSecure** achieving the lowest 70B count at 48.00 (+4.3%) and **Abstractive POSecure** 49.67 (+8.0%). This trend is particularly important because the 70B model produced 190–198 valid patches per run across all techniques (over 54% of the 351 analyzed functions), meaning that the increase in vulnerabilities cannot be attributed to a lack of applied patches. On the contrary, when the model successfully integrates retrieved context into compilable code, the result is consistently more vulnerable rather than more secure. Crucially, no technique on either model reduced SAST findings below the original codebase by a meaningful margin.

7.3 RQ3: Code Quality Impact

Code quality was assessed through SonarQube code smells, logical bugs, and technical debt (Table 2). Every technique degraded quality relative to the original repository (40.00 smells, 380.00 minutes of debt), on both model scales. On the 8B model, smells rose by +33% to +63% and debt by +25% to +57%, with **SOSecure** showing the smallest increase and **No RAG** and **Abstractive POSecure** the largest. On the 70B model the regression was more severe: debt rose by +72% to +107% across all techniques, peaking at **Extractive POSecure** (784.67 minutes, +106.5%). The 70B **POSecure** variants also introduced the most logical bugs (13.00 and 7.00 for Extractive and Abstractive), indicating reduced reliability. This matters because an auto-patch technique that preserves or slightly improves security but substantially worsens maintainability is unlikely to be acceptable in practice.

Beyond the aggregate metrics, we also conduct a small qualitative analysis of the issues introduced by POSecure and the CVEfixes variation, examining the per-rule breakdown of findings relative to the original repository. Three quality patterns recur across all treatments. The first is *dead code*: unnecessary imports and unused assignments are the most frequently added smells. The second is *increased function complexity*: Cognitive Complexity rises in every treatment. The third is *unreachable code*, the dominant logical bug, often accompanied by conditional branches sharing identical implementations. Regarding security, the most consistently introduced CodeQL findings are *potential server crashes* and *incomplete sanitization*, suggesting that defensive code added by the model is often partial or unsafe at runtime. Overall, the patches tend to add redundant, more complex, and partially unreachable code rather than concise, well-integrated fixes. We examine the implications of these patterns, and why prompt-level constraints fail to prevent them, in Subsection 8.4.

7.4 RQ4: Cost-Benefit Trade-off

We evaluated cost through token consumption, execution time, and financial cost (Table 2); no financial cost is reported for the local 8B model. On both scales, retrieval imposes large overhead over **No RAG**, and the ordering of techniques is consistent. **SOSecure** is the most expensive by a wide margin (8B: +626% input tokens, +125% time; 70B: +348% tokens, +259.4% cost), followed by **CVEfixes** (+335% tokens at 8B; +219% tokens, +169.8% cost at 70B). The **POSecure** variants substantially reduce this overhead: Abstractive POSecure is the cheapest RAG-based technique at both scales (+24.8% tokens at 8B; +21.0% tokens, +20.8% cost at 70B), and Extractive POSecure follows (+61.4% cost at 70B).

A scale-dependent effect is visible in the **Input Tokens** metric: although the initial prompts are identical across both models, the 8B run consumes substantially more input tokens overall. Each failed first-pass patch triggers an additional healing prompt that

re-sends the generated code together with the compiler errors, so the lower compilability of the 8B model (RQ1) compounds directly into higher cost.

8 Discussion

The results presented in Section 7 reveal a consistent pattern across all five techniques and both model scales: retrieval-augmented generation, regardless of the type of context provided, fails to improve the security of the patched codebase, and frequently degrades it. We discuss the implications of these findings and the trade-offs that emerge when LLM-based vulnerability repair is applied to a realistic TypeScript backend rather than isolated code snippets. Our findings do not simply show that one technique underperformed another; more importantly, they indicate that the evaluation setting itself matters: when vulnerability repair is moved from isolated snippets to repository-level patching, the relevant notion of success changes.

Taken together, the results expose trade-offs across the four dimensions we measured. Context-based approaches consistently introduce substantial operational overhead and degrade code quality, while their effect on security is at best neutral and frequently negative. These effects are amplified in resource-constrained settings, where smaller models not only fail to leverage retrieved context effectively but also compound its cost through repeated healing attempts. We examine each of these dimensions, and why they arise, in the following subsections.

8.1 Why Does RAG Fail to Improve Security in a Realistic Codebase?

Our results stand in apparent contrast with prior work that reports substantial fix rates for retrieval-augmented vulnerability repair. Mukherjee and Hellendoorn [26] report fix rates between 71% and 97% for SOSecure across the SALLM, LLMSecEval, and LMSys datasets. In our evaluation, no technique reduces the number of SAST findings reported by CodeQL below the **Original repo**. Two methodological differences help explain this divergence and suggest that the gains reported in prior work may not generalize to realistic codebases.

First, prior evaluations operate on isolated snippets curated to contain specific CWE patterns. In such settings, the LLM is not required to reconcile the patch with surrounding type definitions, framework conventions, or cross-file dependencies. In our evaluation, each function is patched in place within a Juice Shop backend that includes Express routes and Sequelize [13] models, and the patch must compile against the project's existing type signatures. The retrieved context, while it may discuss the relevant security pattern, may not align with the constraints of the host codebase.

Second, analyses such as those performed by CodeQL operate over data flow graphs that span multiple files. A defensive transformation applied to one function may expose previously unreachable sinks elsewhere or alter the taint propagation path the analyzer follows. This is consistent with our observation that the 70B model, despite producing close to 200 valid patches per run, never reduces the number of CodeQL findings; in fact, all techniques increase them by 4–12% relative to the original codebase.

[13] https://sequelize.org/

These findings align with the broader observation by Toran et al. [44] that LLMs, when prompted with security-related Stack Overflow content, often produce *more* vulnerabilities than the human-authored answers used as reference. Our results extend this observation to the project level: even when the LLM successfully integrates retrieved security guidance into a syntactically valid patch, the project-level security posture does not improve.

8.2 Model Capacity Dominates Retrieval Strategy

A second consistent pattern is that the choice of retrieval strategy matters less than the capacity of the underlying model. For the 70B model, all four RAG variants and the No RAG baseline cluster within a narrow band of 190–198 valid patches per run, and SAST findings vary by less than four vulnerabilities across the entire treatment space. For the 8B model, the same techniques span a much wider range, from 19.67 valid patches in SOSecure to 153.00 in No RAG.

This echoes the broader finding by Sultana et al. [42] that prompt-based strategies are sensitive to model choice and configuration, while methods such as fine-tuning yield comparatively more uniform results. In our setting, model capacity plays an analogous stabilizing role: the 70B model absorbs the additional complexity of long retrieved contexts and emits patches that, while not necessarily more secure, are at least syntactically and semantically valid most of the time. The 8B model, by contrast, treats the additional context as a destabilizing input, leading to first-pass patches entangled enough to overwhelm the self-healing mechanism.

8.3 The Hidden Cost of Healing in Smaller Models

A finding that emerges in the cost dimension is that the operational overhead of RAG techniques scales non-linearly with model capacity. On the 70B model, SOSecure consumes 1.59M input tokens per run; on the 8B model, the same technique consumes 2.73M tokens, a 71% increase for an identical input prompt set. This increase is explained not by the initial RAG prompts (which are identical across both models), but by the `Failed_Healings` metric: the 8B model produces first-pass patches that fail to compile more often (325.00 failed healings for SOSecure vs. 143.33 for the 70B), and each failed first-pass triggers a second LLM call that re-sends both the previously generated code and the compiler errors as input.

The cost-effectiveness of RAG-based vulnerability repair, when measured on larger models, does not transfer proportionally to smaller ones. Practitioners considering smaller models for resource-constrained deployments should be aware that apparent savings can be offset by the amplification of input cost through the healing loop, with no compensating gain in patch quality. More broadly, smaller-model deployments do not automatically translate into cheaper workflows: the absence of per-token cloud charges may be offset by hardware acquisition, energy consumption, operational maintenance, and substantially longer execution times. In our setting, for example, the 8B model required between 2.7× and 5.3× more inference time than the 70B model under equivalent treatments, and these numbers exclude additional overheads such as

compilation checks and context retrieval that contribute to the end-to-end workflow duration.

8.4 Code Quality Regression as a Systemic Side Effect

Across all techniques and models, the application of LLM-generated patches introduces a measurable regression in code quality, as captured by the SonarQube metrics for code smells and technical debt. The original codebase contains 40 code smells and 380 minutes of estimated technical debt; after patching, every technique increases both metrics, with Extractive POSecure on the 70B model reaching 784.67 minutes of debt (a 106% increase), for example.

This regression is not an incidental artifact of a few problematic patches, but a systemic effect that emerges across all five techniques and both model scales. The increases in smells and debt persist regardless of whether retrieval is used, which type of context is provided, or whether prompt compression is applied, suggesting that the source of the degradation is the act of generating patches itself rather than any specific retrieval strategy. As reported in RQ3, the recurring patterns behind these metrics are consistent across treatments: dead code (unnecessary imports and unused assignments), increased cognitive complexity, and unreachable code, alongside security findings such as potential server crashes and incomplete sanitization.

A notable aspect of these patterns is that our prompts already include explicit guardrails against several of them. The model is instructed to modify the code minimally, to add no new imports, and to preserve the original function signature and intent. The fact that unnecessary imports, dead code, and added complexity still dominate indicates that prompt-level instructions alone are insufficient to enforce these constraints, reinforcing our broader finding that the limitations are structural rather than a matter of prompt wording. This points to enforcement mechanisms beyond prompting. Since most of these patterns are reliably detectable by static analyzers, one direction is to integrate static-analysis feedback into the self-healing loop, rather than relying on compiler errors alone, so that a patch which compiles but adds dead code, raises complexity, or introduces runtime risks can be automatically rejected or regenerated. Complementarily, a deterministic post-processing pass (e.g., dead-code elimination and import pruning) could remove a substantial fraction of the introduced smells without depending on model compliance.

For the practical adoption scenarios that motivate our study, this finding has direct implications. SSMEs adopting auto-patching as a recurring practice would, over time, accumulate technical debt proportionally to the volume of patches applied. Such accumulation may erode the long-term value of the very code being patched, partially offsetting whatever security benefit the patches were intended to deliver. In contexts where engineering capacity is already constrained, paying down this accumulated debt may itself become a non-trivial cost, raising the question of whether security-only metrics are sufficient to evaluate the viability of LLM-based vulnerability repair in production codebases.

8.5 The Effect of Patch Acceptance Rate on Project-Level Metrics

The central finding of this study is that no technique meaningfully improved security on either model. However, the magnitude of the SAST and code quality numbers in Section 7 must be interpreted in light of how many patches were actually applied. SOSecure on the 8B model produced only 19.67 valid patches per run, against 191.33 on the 70B model: approximately 94% of targeted functions on the 8B run were restored to their original state after the self-healing mechanism could not recover compilable code. The artifact analyzed by CodeQL and SonarQube under that treatment is therefore predominantly the unmodified Juice Shop backend, with only a small fraction of patched functions contributing to the reported metrics.

This effect explains, for example, why SOSecure on the 8B model reports 46.00 SAST findings (matching the original repository) while SOSecure on the 70B reports 51.67: the 8B run had little patched code to introduce additional findings in the first place. A similar pattern is visible in CVEfixes on the 8B model, where 255.33 failed healings out of 351 analyzed functions leave most of the codebase untouched.

This observation does not invalidate the central conclusion of this study. On both models, no technique reduced SAST findings below the original codebase by a meaningful margin, and code quality consistently regressed wherever patches were applied. It does, however, mean that direct numerical comparisons between treatments and between models on project-level metrics should weight the proportion of patches actually applied: when this proportion diverges across treatments, the project-level metrics become skewed, reflecting the patched fraction of the codebase rather than the model's intrinsic behavior.

9 Threats to Validity

We discuss threats to validity following Wohlin et al. [48], organized into construct, internal, external, and conclusion validity.

Construct validity. Each dimension was assessed with a single tool (CodeQL for static analysis, SonarQube for code quality), each with its own detection rules and false positive/negative profile; triangulation across multiple tools would strengthen our findings. Our security analysis also aggregates SAST findings into a single unweighted count per treatment, without severity or exploitability weighting. Finally, the Cost ($) metric covers only the 70B cloud deployment and omits the full cost of local 8B execution (hardware, electricity, maintenance, and longer run times); a workflow-level cost comparison is left as future work.

Internal validity. The 8B-vs-70B comparison is confounded by the differential rate at which patches are applied; we mitigate this by reporting `Valid_Patches` and `Failed_Healings` alongside project-level measurements and discussing the effect in Section 8, but it cannot be fully eliminated without a stricter acceptance policy. Our design originally included a runtime evaluation with OWASP ZAP[14], but it could not be run consistently: patched builds failed to start due to runtime issues absent during generation (e.g., missing modules from inconsistencies between the symlinked dependency

[14] https://www.zaproxy.org/

tree and the build artifacts), preventing runtime exploitability metrics. A lesson is that passing TypeScript compilation does not guarantee runtime executability in projects with native dependencies, and future work should add runtime smoke tests to the acceptance criteria. Finally, our leakage-reduction filtering is not exhaustive (some challenge-related calls such as `challengeUtils.solveIf` remained visible); since such cues could only ease the task and no technique improved security regardless, this does not weaken our negative findings.

External validity. Our evaluation used a single codebase, the OWASP Juice Shop backend (`routes` and `models`). Although it offers the structural complexity needed for repository-level repair, it is a deliberately vulnerable training application rather than a production system with organic evolution, and its Angular frontend, whose vulnerabilities may interact differently with retrieval, was not evaluated. Both patch-generation models also belong to the same family, so our findings may reflect Llama-specific characteristics. Replication across additional repositories, technology stacks, and model families (e.g., Qwen, DeepSeek, Mistral, including code-specialized variants) is needed to assess generalizability.

Conclusion validity. Each treatment was run three times to account for non-determinism in LLM inference. Three samples provide limited statistical power and only an early indicator of stability, so we report means rather than statistical tests and avoid strong claims resting on small numerical differences. Replications with more iterations would enable statistical comparison and confidence-interval reporting.

10 Conclusion and Future Work

In this work, we proposed POSecure, an adaptation of SOSecure that incorporates prompt compression to reduce unnecessary context. We evaluated POSecure (extractive and abstractive variants) against the original SOSecure, a CVEfixes-derived variation, and a No RAG baseline, on the OWASP Juice Shop backend, a realistic TypeScript project. Each technique was applied with both Llama 3.1 8B locally and Llama 3.3 70B on OCI, assessed through static security analysis, code quality, and operational cost, totaling 30 runs over 351 functions.

Our central finding is that none of the evaluated techniques meaningfully improved the security of the patched repository. On the 70B model, all techniques increased CodeQL findings relative to the original codebase by 4% to 12%. On the 8B model, the only technique to slightly reduce findings was CVEfixes (−0.7%), but this occurred within a context where roughly three quarters of patches were rejected by the self-healing mechanism, indicating that the apparent reduction reflects unmodified code rather than genuine security improvements. In parallel, every technique degraded code quality, increasing code smells by up to 81% and technical debt by up to 106%, while substantially raising token consumption and inference cost. These overheads were especially pronounced in resource-constrained settings, where repeated self-healing attempts amplify the operational cost of RAG-based approaches.

These results suggest that the security gains reported by prior work on curated vulnerability snippets do not transfer directly to the realistic repository we studied, where patches must reconcile with surrounding type definitions, framework conventions, and cross-file dependencies. Even with mechanisms designed to mitigate these challenges (e.g., the 1-hop dependency graph extractor), the gap between snippet-level evaluation and repository-level application remains substantial in our setting. We therefore call for further studies across other repositories, languages, and models to establish how broadly these findings generalize.

Future work should also focus on extending current techniques toward repository-level applicability. We see three directions as particularly promising. First, retrieval and prompting strategies that incorporate richer cross-file context, beyond only the type signatures, may reduce both the rate of compilation failures and the introduction of code quality regressions. Second, more robust self-healing mechanisms, possibly informed by static analyzers or test execution feedback rather than compiler errors alone, may close the gap observed in smaller models, where the healing loop currently amplifies cost without producing usable patches. Third, alternative retrieval strategies, such as embedding-based or hybrid retrieval, could surface more relevant security context than the lexical matching used by BM25.

These directions connect to the practical motivation of our study. In resource-constrained environments, where limited budget and infrastructure make local, smaller-model deployments realistic (as is often the case for SSMEs in emerging countries [36]), RAG-based vulnerability repair must remain effective on whole repositories using open-weight models within a constrained resource envelope. Beyond the Llama family considered here, broader investigations across model families (e.g., Qwen, DeepSeek, Mistral) and code-specialized versus general-purpose models are needed to characterize how robustly these techniques generalize for local deployment.

Finally, future work should also strengthen how security effectiveness is measured. Our evaluation relied on static analysis, as we were unable to obtain stable runtime measurements in our setting. Complementing static findings with dynamic evidence, through DAST, professional penetration testing, or LLM-based penetration testing, would provide stronger evidence of whether patches actually reduce exploitability rather than only altering static findings.

Artifact Availability

All scripts and instructions necessary to run the study are available in https://github.com/brenovergilio/so-vuln-repair. Supplementary Material [46] is also available.

Acknowledgments

In accordance with the SBES policy, we disclose the use of generative AI tools (OpenAI's ChatGPT, Anthropic's Claude, Google's Gemini, and NotebookLM) to assist with pipeline code generation and debugging, analysis of results, manuscript writing, and methodological discussions. All AI-generated content was reviewed by the authors, who discarded inaccuracies and take full responsibility for the final content, including claims, methodology, and conclusions.

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), Brasil, Finance Code 001, by the Fundação de Apoio ao Desenvolvimento do Ensino, Ciência e Tecnologia do Estado de Mato Grosso do Sul (FUNDECT, Call No. 42/2024), and by the Federal University of Mato Grosso do Sul (UFMS).

References

[1] Guru Bhandari, Nikola Gavric, and Andrii Shalaginov. 2025. Generating vulnerability security fixes with Code Language Models. *Information and Software Technology* 185 (2025), 107786. doi:10.1016/j.infsof.2025.107786

[2] Guru Bhandari, Amara Naseer, and Leon Moonen. 2021. CVEfixes: automated collection of vulnerabilities and their fixes from open-source software. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering* (Athens, Greece) *(PROMISE 2021)*. Association for Computing Machinery, New York, NY, USA, 30–39. doi:10.1145/3475960.3475985

[3] Victor R Basili1 Gianluigi Caldiera and H Dieter Rombach. 1994. The goal question metric approach. *Encyclopedia of software engineering* (1994), 528–532.

[4] Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, et al. 2024. A survey on evaluation of large language models. *ACM transactions on intelligent systems and technology* 15, 3 (2024), 1–45.

[5] CVE official website. 2026. About the CVE Program. https://www.cve.org/About/Overview.

[6] Xueying Du, Geng Zheng, Kaixin Wang, Yi Zou, Yujia Wang, Wentai Deng, Jiayi Feng, Mingwei Liu, Bihuan Chen, Xin Peng, Tao Ma, and Yiling Lou. 2026. Vul-RAG: Enhancing LLM-based Vulnerability Detection via Knowledge-level RAG. *ACM Trans. Softw. Eng. Methodol.* (Feb. 2026). doi:10.1145/3797277 Just Accepted.

[7] Darren Edge, Ha Trinh, Newman Cheng, Joshua Bradley, Alex Chao, Apurva Mody, Steven Truitt, Dasha Metropolitansky, Robert Osazuwa Ness, and Jonathan Larson. 2024. From local to global: A graph rag approach to query-focused summarization. *arXiv preprint arXiv:2404.16130* (2024).

[8] Wafaa S El-Kassas, Cherif R Salama, Ahmed A Rafea, and Hoda K Mohamed. 2021. Automatic text summarization: A comprehensive survey. *Expert systems with applications* 165 (2021), 113679.

[9] Douglas Everson, Long Cheng, and Zhenkai Zhang. 2022. Log4shell: Redefining the web attack surface. In *Proc. Workshop Meas., Attacks, Defenses Web (MADWeb)*. 1–8.

[10] Ehsan Firouzi and Mohammad Ghafari. 2026. Persistent Human Feedback, LLMs, and Static Analyzers for Secure Code Generation and Vulnerability Detection. In *2026 IEEE International Conference on Software Analysis, Evolution and Reengineering - Companion (SANER-C)*. 265–272. doi:10.1109/SANER-C67878.2026.00042

[11] Damian Gnieciak and Tomasz Szandala. 2025. Large Language Models Versus Static Code Analysis Tools: A Systematic Benchmark for Vulnerability Detection. *IEEE Access* 13 (2025), 198410–198422. doi:10.1109/ACCESS.2025.3635168

[12] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024).

[13] Mohd Ariful Haque, Fahad Rahman, Kishor Datta Gupta, Khalil Shujaee, and Roy George. 2025. TinyLLM: Evaluation and Optimization of Small Language Models for Agentic Tasks on Edge Devices. *arXiv preprint arXiv:2511.22138* (2025).

[14] Md Mahade Hasan, Muhammad Waseem, Kai-Kristian Kemell, Jussi Rasku, Juha Ala-Rantala, and Pekka Abrahamsson. 2025. Assessing small language models for code generation: An empirical study with benchmarks. *arXiv preprint arXiv:2507.03160* (2025).

[15] Chandra Irugalbandara, Ashish Mahendra, Roland Daynauth, Tharuka Kasthuri Arachchige, Jayanaka Dantanarayana, Krisztian Flautner, Lingjia Tang, Yiping Kang, and Jason Mars. 2024. Scaling down to scale up: A cost-benefit analysis of replacing OpenAI's LLM with open source SLMs in production. In *2024 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 280–291.

[16] Huiqiang Jiang, Qianhui Wu, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. 2023. Llmlingua: Compressing prompts for accelerated inference of large language models. In *Proceedings of the 2023 conference on empirical methods in natural language processing*. 13358–13376.

[17] Mete Keltek, Rong Hu, Mohammadreza Fani Sani, and Ziyue Li. 2025. LSAST: Enhancing Cybersecurity Through LLM-Supported Static Application Security Testing. In *ICT Systems Security and Privacy Protection*, Lili Nemec Zlatolas, Kai Rannenberg, Tatjana Welzer, and Joaquin Garcia-Alfaro (Eds.). Springer Nature Switzerland, Cham, 166–179.

[18] Björn Kimminich and OWASP Contributors. 2026. OWASP Juice Shop. https://owasp.org/www-project-juice-shop/.

[19] Hannes Kunstmann, Joseph Ollier, Joel Persson, and Florian von Wangenheim. 2024. EventChat: Implementation and user-centric evaluation of a large language model-driven conversational recommender system for exploring leisure events in an SME context. arXiv:2407.04472 [cs.IR] https://arxiv.org/abs/2407.04472

[20] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 9459–9474. https://proceedings.neurips.cc/paper_files/paper/2020/file/6b493230205f780e1bc26945df7481e5-Paper.pdf

[21] Zongjie Li, Zhibo Liu, Wai Kin Wong, Pingchuan Ma, and Shuai Wang. 2024. Evaluating C/C++ Vulnerability Detectability of Query-Based Static Application Security Testing Tools. *IEEE Transactions on Dependable and Secure Computing* 21, 5 (2024), 4600–4618. doi:10.1109/TDSC.2024.3354789

[22] Keke Lian, Bin Wang, Lei Zhang, Libo Chen, Junjie Wang, Ziming Zhao, Yujiu Yang, Miaoqian Lin, Haotong Duan, Haoran Zhao, Shuang Liao, Mingda Guo, Jiazheng Quan, Yilu Zhong, Chenhao He, Zichuan Chen, Jie Wu, Haoling Li, Zhaoxuan Li, Jiongchi Yu, Hui Li, and Dong Zhang. 2025. A.S.E: A Repository-Level Benchmark for Evaluating Security in AI-Generated Code. arXiv:2508.18106 [cs.SE] https://arxiv.org/abs/2508.18106

[23] Yang Liu, Li Zhang, Fang Liu, Zhuohang Wang, Donglin Wei, Zhishuo Yang, Kechi Zhang, Jia Li, and Lin Shi. 2026. RepoScope: Leveraging Call Chain-Aware Multi-View Context for Repository-Level Code Generation. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26)*. ACM, Rio de Janeiro, Brazil. doi:10.1145/3744916.3773211

[24] Michael Chen. 2024. What Is Cloud Infrastructure? https://www.oracle.com/cloud/cloud-infrastructure/.

[25] MITRE Corporation. 2026. CWE - Frequently Asked Questions. https://cwe.mitre.org/about/faq.html.

[26] Manisha Mukherjee and Vincent J Hellendoorn. 2025. Sosecure: Safer code generation with rag and stackoverflow discussions. *arXiv preprint arXiv:2503.13654* (2025).

[27] National Institute of Standards and Technology (NIST). 2021. CVE-2021-44228 Detail. https://nvd.nist.gov/vuln/detail/cve-2021-44228. Acesso em: 5 abr. 2026.

[28] Claudia Sofia Negri-Ribalta, Rémi Géraud-Stewart, Anastasia Sergeeva, and Gabriele Lenzini. 2024. A systematic literature review on the impact of AI models on the security of code generation. *Frontiers in Big Data* 7 (2024), 1386720. doi:10.3389/fdata.2024.1386720

[29] Giannis Nikolentzos, George Dasoulas, and Michalis Vazirgiannis. 2020. K-hop graph neural networks. *Neural Networks* 130 (2020), 195–205.

[30] OWASP Foundation. 2025. OWASP Top Ten Web Application Security Risks. https://owasp.org/www-project-top-ten/.

[31] Zhuoshi Pan, Qianhui Wu, Huiqiang Jiang, Menglin Xia, Xufang Luo, Jue Zhang, Qingwei Lin, Victor Rühle, Yuqing Yang, Chin-Yew Lin, et al. 2024. Llmlingua-2: Data distillation for efficient and faithful task-agnostic prompt compression. In *Findings of the Association for Computational Linguistics: ACL 2024*. 963–981.

[32] Rupam Patir, Keyan Guo, Haipeng Cai, and Hongxin Hu. 2025. Fortifying LLM-Based Code Generation with Graph-Based Reasoning on Secure Coding Practices. *arXiv preprint arXiv:2510.09682* (2025).

[33] Hammond Pearce, Baleegh Ahmad, Brendan Dolan-Gavitt, Ramesh Karri, and Ben Tan. 2025. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. *Commun. ACM* 68, 2 (2025), 96–105.

[34] Weihan Peng, Yuling Shi, Yuhang Wang, Xinyun Zhang, Beijun Shen, and Xiaodong Gu. 2025. Swe-qa: Can language models answer repository-level code questions? *arXiv preprint arXiv:2509.14635* (2025).

[35] Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. 2023. Do Users Write More Insecure Code with AI Assistants?. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23)*. Association for Computing Machinery, New York, NY, USA, 2785–2799. doi:10.1145/3576915.3623157

[36] Chaiyong Ragkhitwetsagul, Jens Krinke, Morakot Choetkiertikul, Thanwadee Sunetnanta, and Federica Sarro. 2024. Adoption of automated software engineering tools and techniques in Thailand. *Empirical Software Engineering* 29, 4 (2024), 90.

[37] Stephen Robertson and Hugo Zaragoza. 2009. The Probabilistic Relevance Framework: BM25 and Beyond. *Found. Trends Inf. Retr.* 3, 4 (April 2009), 333–389. doi:10.1561/1500000019

[38] Ita Ryan, Utz Roedig, and Klaas-Jan Stol. 2026. Software Security Is Your Highest Priority. Do Your Developers Know That? *IEEE Security Privacy* (2026), 2–10. doi:10.1109/MSEC.2026.3670641

[39] Patricia de Oliveira Santos, Allan Chamon Figueiredo, Pedro Nuno Moura, Bruna Diirr, Adriana C. F. Alvim, and Rodrigo Pereira Dos Santos. 2024. Impacts of the Usage of Generative Artificial Intelligence on Software Development Process. In *Proceedings of the 20th Brazilian Symposium on Information Systems (SBSI '24)*. Association for Computing Machinery, New York, NY, USA, Article 65, 9 pages. doi:10.1145/3658271.3658337

[40] Baljeet Singh. 2025. Automating Security Testing in CI/CD Pipelines using DevSecOps Tools a Comprehensive Study. *CD Pipelines using DevSecOps Tools a Comprehensive Study (May* 23 (2025), 2025.

[41] Stack Exchange, Inc. 2025. Stack Exchange Data Dump 2025-12-31. https://archive.org/details/stackexchange_20251231. Contributor: Stack Exchange Community. Item Size: 91.6G. Acesso em: 2 abr. 2026.

[42] Shaznin Sultana, Sadia Afreen, and Nasir U. Eisty. 2026. LLMs in Code Vulnerability Analysis: A Proof of Concept. In *Proceedings of the 4th International Workshop on Software Vulnerability Management (SVM '26)*. ACM, Rio de Janeiro, Brazil, 17–24. doi:10.1145/3786165.3788439

[43] Zhao Tian, Pengfei Gao, Junjie Chen, and Chao Peng. 2026. Agent-Based Ensemble Reasoning for Repository-Level Issue Resolution. In *2026 IEEE/ACM 48th*

International Conference on Software Engineering (ICSE '26). ACM, Rio de Janeiro, Brazil. doi:10.1145/3744916.3787761

[44] Markus Toran, Bettina Ballin, Marc Miltenberger, and Steven Arzt. 2026. Q&AEval: Benchmarking Secure Coding Ability of LLMs on Real-World Tasks. In *Proceedings of the 4th International Workshop on Software Vulnerability Management (SVM '26)*. ACM, Rio de Janeiro, Brazil, 1–8. doi:10.1145/3786165.3788437

[45] Venkata Kiran Vemulapalli. 2025. Choosing the Right Model for Enterprise AI Applications: A Comprehensive Analysis of Small Language Models versus Large Language Models in Enterprise Architecture. *European Modern Studies Journal* 9 (09 2025), 275–284. doi:10.59573/emsj.9(5).2025.25

[46] Breno Vergílio, Matheus Mota, and Awdren Fontão. 2026. Supplementary Material. https://github.com/brenovergilio/so-vuln-repair/blob/main/supplementary-material.pdf.

[47] Fali Wang, Zhiwei Zhang, Xianren Zhang, Zongyu Wu, Tzuhao Mo, Qiuhao Lu, Wanjing Wang, Rui Li, Junjie Xu, Xianfeng Tang, et al. 2025. A comprehensive survey of small language models in the era of large language models: Techniques, enhancements, applications, collaboration with llms, and trustworthiness. *ACM Transactions on Intelligent Systems and Technology* 16, 6 (2025), 1–87.

[48] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in Software Engineering*. Springer Berlin Heidelberg. doi:10.1007/978-3-642-29044-2

[49] Fangyuan Xu, Weijia Shi, and Eunsol Choi. 2023. Recomp: Improving retrieval-augmented lms with compression and selective augmentation. *arXiv preprint arXiv:2310.04408* (2023).