POLITECNICO DI TORINO

MASTER's Degree in CYBERSECURITY

MASTER's Degree Thesis

A Hybrid Multi-Agent Architecture for Enhancing CodeQL Static Analysis with Large Language Models

Supervisors

Prof. ADAM ALAMI

Prof. DANILO GIORDANO

Prof. MATTEO BOFFA

Candidate

REBECCA DE ROSA

MARCH 2026

Abstract

Software vulnerabilities remain a critical challenge, with tens of thousands of Common Vulnerability and Exposure (CVEs) recorded annually. Static analysis tools like CodeQL offer scalable, deterministic detection through pre-determined rule-based queries, but they are limited in contextual reasoning and novel vulnerability patterns, while fully LLM-based approaches raise concerns regarding reproducibility, cost, and integration within established DevSecOps pipelines. This thesis proposes a hybrid three-agent architecture that uses LLMs to augment CodeQL rather than replace it. An Analyzer agent validates CodeQL results through autonomous reasoning on source code, tripling CodeQL's F1-score on a labeled Python dataset (0.774 vs 0.209). A Suggestor agent identifies coverage gaps by analyzing false negatives and generating structured improvement proposals, and a Creator agent synthesizes new CodeQL queries based on these proposals, successfully targeting missing sources, sinks, and taint-propagation steps for CWE-89, CWE-79, and CWE-78. Preliminary LLM-as-Judge evaluation confirmed high gap coverage (4-5/5), thought generated queries required manual refinement to compile due to syntactic issues (3.5/5 syntactic correctness). These results demonstrate that hybrid LLM-SAST pipeline can substantially augment static analysis through contextual reasoning, while preserving the determinism and integration properties essential for production use.

Acknowledgements

ACKNOWLEDGMENTS

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Context and Motivation

Software vulnerabilities represent a critical and escalating threat to modern digital infrastructure, due to the increasing sophistication of software systems and cyber-attacks [1]. As a result, the ability to identify and mitigate vulnerabilities in source code has become a fundamental requirement in modern software development. Statistics from recent years paint an alarming picture. The number of reported Common Vulnerabilities and Exposures (CVEs) has grown significantly, rising from only a few thousand annually in the early 2000s to more than 28,000 newly published in 2023, representing a 15% increase over the previous year [2]. The trend accelerated in 2024, with over 49,000 CVEs reported, marking a 38% increase compared to 2023 [3]. This growth is driven not only by the increasing volume of code produced globally, but also by the continuous emergence of new attack patterns that exploit the rapid evolution of frameworks, libraries, and programming paradigms.

In this landscape, traditional Static Analysis Security Testing (SAST) tools have assumed a central role in modern development pipelines. Tools such as CodeQL, SonarQube, and Snyk allow developers to analyze source code without running it, identifying potential vulnerabilities through pattern matching and data-flow analysis [4]. These tools integrate directly into Continuous Integration and Continuous Deployment (CI/CD) pipelines, enabling automated security checks at every stage of the software development lifecycle. CodeQL in particular, developed by GitHub and completely integrated into the platform [5], has achieved significant adoption due to its ability to express complex vulnerability patterns through a powerful declarative language and its scalability across codebases of millions of lines of code. Unlike rule-based tools that perform checks operating on syntactic patterns, CodeQL models program semantics by constructing a relational database

representation of the code, enabling queries that reason about data flow, control flow, and taint propagation across entire codebases. A large-scale empirical study conducted on 258 embedded open-source projects demonstrated that CodeQL was able to detect 709 real-world defects with a false positive rate of 34%, of which 75% are classified as potential security vulnerabilities [6].

Despite their widespread adoption, traditional SAST tools suffer from inherent limitations that undermine their effectiveness in modern development contexts. In particular, they rely on queries written manually by security experts, struggle to adapt to new frameworks and libraries without explicit rule updates, and produce false negatives when they encounter unknown vulnerability patterns [7]. The manual dependency of these rule sets leads to the consequence that new vulnerability classes are only detectable once a corresponding rule has been created, reviewed, and deployed. This creates a structural lag between the advent of novel attack patterns and the availability of detection coverage in deployed tools.

A recent survey examining the limitations of adopting SAST tools identified several usability issues encountered by developers. Some of these issues arise from the technology itself, while others reflect shortcomings that tool developers still have to address [8]. In particular, an empirical study of Charoenwet et al. [9] on real-world vulnerabilities in C/C++ projects shows that at least 76% of warnings produced in vulnerable functions are unrelated to actual vulnerabilities. Furthermore, 22% of vulnerability-fixing commits are not detected due to inherent limitations in SAST rule design.

These limitations have motivated the exploration of alternative approaches that can adapt to novel patterns without the need for explicit rule updates. In parallel, the emergence of Large Language Models (LLMs) such as GPT-4 and Claude has opened new possibilities for code understanding and generation. These models, trained on a large number of parameters, have a form of semantic structure that allows them to reason about code in ways that go beyond syntactic pattern matching [10]. Recent systematic surveys catalog the rapid progress on this domain: Sheng et al. [11] analyzed 58 preliminary studies on LLM-based vulnerability detection and repair, while Yao et al. [12] survey 83 articles on positive security applications of LLMs. These studies document impressive capabilities in zero-shot vulnerability identification and automated path generation, suggesting that LLMs could complement traditional static analysis workflows.

However, empirical evaluation reveals critical limitations for production deployment. Khare et al. [13] demonstrate that LLMs remain unreliable in detecting vulnerabilities in real-world code at method level, achieving precision and recall significantly below industry requirements. Furthermore, concerns regarding reproducibility, computational costs, and confidentiality of proprietary code sent to cloud APIs create barriers to enterprise adoption.

This landscape gives rise to a fundamental research question: how can the accuracy

and scalability of traditional SAST tools be combined with the contextual reasoning and adaptability capabilities of LLMs, eventually creating a hybrid system that overcomes the limitations of both approaches individually? Rather than consider these two paradigms as competitors, a promising direction could be their complementarity, where static analysis provides a deterministic and auditable foundation, while LLM-based reasoning fills the semantic gaps that formal rules cannot easily detect.

Recent pioneering works demonstrate the feasibility of hybrid approaches. Li et al. [14] introduce IRIS, a neuro-symbolic approach that combines LLMs with static analysis to perform vulnerability reasoning, even if their approach focuses on result validation rather than automated rule improvement. He et al. [15] conducted a systematic review of LLM-based multi-agent systems for software engineering, documenting how multi-agent architecture can improve robustness and scalability in addressing complex software engineering tasks, while also mitigating known limitations of standalone LLMs, such as hallucination, through cross-examination and iterative validation mechanisms.

These works establish technical feasibility but leave critical gaps, such as the fact that no existing systems systematically identify what SAST rules are missing, nor automates the generation of improved detection queries grounded in empirical evidence of false negatives.

This thesis is situated within this emerging line of research through the design, implementation, and evaluation of a multi-agent architecture that integrates LLM-based reasoning into the CodeQL static analysis workflow, aiming to demonstrate that hybrid approaches can improve vulnerability detection without sacrificing the determinism and reproducibility required for industrial adoption.

1.2 Research Problem

The previous section outlines that traditional SAST tools suffer from limitations such as rigidity of the rules, lack of contextual reasoning, and coverage gaps that are difficult to identify. At the same time, LLMs offer semantic reasoning capabilities, but lack production-readiness guarantees in terms of determinism, auditability, and computational efficiency, required for deployment in industrial software development pipelines.

The central research question of this thesis is:

> **How can an LLM-based multi-agent system improve the effectiveness of CodeQL vulnerability detection?**

Answering this central question requires decomposing it into three operational sub-questions, each addressing a distinct phase of the hybrid workflow:

RQ1 - Autonomous Validation: Can an LLM agent autonomously validate CodeQL results through contextual code reasoning, improving precision, by filtering false positives and recall, by identifying false negatives, compared to CodeQL analysis? This question aims to verify whether the semantic understanding of LLM can reliably differentiate between genuine vulnerabilities and spurious SAST alerts, without access to additional runtime information.

RQ2 - Systematic Gap Identification: Can a multi-agent system automatically identify which specific sources, sinks, and sanitizers are missing from existing standard CodeQL queries? This question tries to address the core coverage problem, which consists of verifying if the system can produce actionable and structured proposals that overcome the precise modeling gap responsible for the false negative, rather than simply flagging that a vulnerability was missed.

RQ3 - Query Synthesis: Can an LLM agent synthesize a CodeQL query draft targeting identified gaps, reducing manual effort even when generated queries require human refinement before deployment? This question examines the extent to which rule manual writing can be automated, and what level of quality can be expected from an AI-generated QL code in terms of syntactic correctness, semantic accuracy, and structural alignment with CodeQL conventions.

1.3 Proposed Approach

To address the research questions mentioned above, this thesis proposes a three-agent architecture that integrates LLM-based reasoning into the CodeQL static analysis workflow. The system is designed around the principle of augmentation rather than replacement. Indeed, CodeQL continues to serve as the deterministic, scalable analysis tool, while LLM-based agents provide a higher level of reasoning capabilities that address the SAST tool's structural limitations. Each agent is specialized in a particular task:

- **AnalyzerAgent**: Validates CodeQL results by independently analyzing source files through contextual reasoning. It produces a comparative report identifying where CodeQL and the agent share the same results, where they do not, and which findings are false positives or false negatives when compared with ground truth annotations.
- **SuggestorAgent**: Systematically identifies the coverage gaps between the two approaches and generates structured improvement proposals specifying which sources, sinks, and sanitizers are missing from existing standard CodeQL queries, grounded in concrete evidence from files that CodeQL failed to detect.

- **CreatorAgent**: Synthesizes new or improved CodeQL queries based on the proposals of the previous agent. Iteratively generates QL code (CodeQL language), validates it through compilation checks and refinements based on compiler feedback, producing deployment-ready or near-deployment-ready detection rules.

Each agent operates following the ReAct paradigm (Reasoning and Acting), a framework for prompting large language models (LLMs) on tasks that require explicit reasoning and/or acting in an environment [16]. The general idea consists of integrating this system into the CodeQL workflow to improve queries when necessary.
In this way, this thesis contributes to demonstrating the potential of a multi-agent architecture based on Large Language Models (LLMs) to support and structure the security analysis process.
Key Design Principles:

- Augmentation, not replacement: CodeQL maintains its role as a deterministic, scalable, and verifiable detection mechanism. LLM-based agents assist human experts rather than replacing formal analysis. The goal is to make security engineers more effective, not to remove them from the loop.
- Evidence-based proposals: All query improvement suggestions are grounded in concrete false negative examples, ensuring that generated rules address real and documented detection failures rather than hypothetical scenarios.
- Human-in-the-loop: Generated queries need expert review before deployment, preserving the quality assurance standards of traditional SAST tool development. The system is explicitly positioned as a support tool that reduces manual effort, not as a fully autonomous rule generator.
- Modularity and evaluability: Each agent exposes well-defined inputs and outputs, making it possible to evaluate and improve individual components of the pipeline independently, and to replace or modify specific agents without redesigning the entire system.

1.4 Contributions

This thesis makes the following contributions to the field of automated vulnerability detection:

1. Novel multi-agent architecture: The thesis investigates the design and implementation of a system that integrates LLM-based agents into the CodeQL workflow specifically for automated query improvement, combining autonomous

validation, systematic gap identification, and assisted query generation in a unified pipeline.

2. Gap analysis methodology: A systematic approach for identifying missing coverage in CodeQL queries through CodeQL output analysis, LLMs reasoning, and ground truth annotations, producing structured and actionable improvement proposals.

3. Empirical evaluation: Assessment on a labeled Python vulnerability dataset demonstrating that the AnalyzerAgent achieves a F1-score of 0.774 compared to CodeQL's 0.209, demonstrating substantial improvements in contextual detection. The SuggestorAgent identifies coverage gaps with high accuracy as validated through LLM-as-Judge evaluation. The CreatorAgent generates semantically accurate query drafts, though it still requires human refinement for deployment.

4. Multi-level evaluation framework: An evaluation methodology combining automated metrics, qualitative assessment via LLM-as-a-Judge, and empirical validation through execution on ground truth data. This framework is designed to be applicable for eventual future hybrid SAST-LLM systems.

1.5 Outline of the Thesis

The remainder of this document is organized as follows.

- Chapter 2 provides a comprehensive Background, examining the state-of-the-art in Software Vulnerability Detection (SVD), the limitations of traditional SAST and LLM-based approaches, and the fundamental concepts of AI agent architectures, together with the Related Work to the thesis.

- Chapter 3 presents the proposed Multi-Agent Software Vulnerability Detection (SVD) Architecture Methodology, detailing the role of each agent, the interaction protocols between them, design principles, and the specific mechanism for generating and refining CodeQL queries.

- Chapter 4 describes the Experimental Setup, including the datasets used, the implementation details of the CodeQL-LLM integration, and the memory management in the architecture.

- Chapter 5 presents the Results, offering an analysis of the architecture's performance across all three research questions, including qualitative metrics, qualitative case studies, and a discussion of representative successes and failure modes observed during the evaluation.

- Chapter 6 presents the Conclusions, summarizing the key contributions of the research, discusses the theoretical and practical implications, and outlines the limitations and the suggested avenues for Future Works.

Chapter 2

Background and Related Work

This chapter provides an overview of the technical foundations and related literature relevant to this thesis. Section 2.1 introduces software vulnerability detection (SVD) and traditional approaches, from static analysis tools to machine learning and deep learning methods, highlighting their strengths and limitations. Section 2.2 reviews the recent and emerging field of Large Language Models (LLMs) for vulnerability detection, describing both their capabilities and critical gaps that prevent standalone LLM-based detection from reaching production readiness. Finally, Section 2.3 outlines the concepts of LLM-based agent architectures, with particular focus on multi-agent systems and their application to software engineering and cybersecurity.

2.1 Vulnerability Detection

In the context of software security, vulnerabilities are specific flaws in the software that allow attackers to do something malicious, such as expose or alter sensitive information, disrupt or destroy a system, or take control of a computer system or program [17]. The severity of this problem is reflected in the continuous growth of reported Common Vulnerabilities and Exposures (CVEs), with exactly 48,172 in 2025 (almost 10,000 more than 2024) and 7975 CVEs already reported in 2026 [18]. To systematically categorize and communicate these weaknesses, the security community relies on standardized frameworks. Among them, there are:

- **CVE**: The Common Vulnerabilities and Exposures (CVEs) are unique identifiers assigned to publicly known cybersecurity vulnerabilities. These identifiers help security professionals and organizations communicate about specific weaknesses, ensuring that everyone refers to the same vulnerability with a common

name. CVEs are essential for knowledge-sharing, enabling researchers and vendors to collaborate and develop appropriate patches or mitigation to protect systems from potential exploitations. Unfortunately, vulnerabilities can be complex, involving intricate technical details such as specific products and versions [19].

- **CWE**: Common Weakness Enumeration (CWE) [20] is a community-developed list of common software weaknesses and security flaws. Unlike CVEs, which identify specific vulnerabilities, CWEs categorize broader classes of weaknesses, embracing various instances of similar vulnerabilities. This classification helps to understand the root causes of vulnerabilities and facilitates more comprehensive security measures during software development and system deployment. The CWEs explain how (conditions and procedures) and why a vulnerability can be exploited (cause), and clarify the consequences (impact)[19].

In addition to classification, the Common Vulnerability Scoring System (CVSS) is used to categorize different aspects of vulnerabilities. The result of this categorization is a vector whose elements are a machine-readable representation of the vulnerability's properties. Based on the components of the CVSS vector, a numerical severity score is calculated, ranging from 0.0 (informational) to 10.0 (critical). This score enables organizations to prioritize remediation efforts in proportion to the risk posed by each vulnerability. The vulnerability assessment is usually performed by IT security experts based on the available Open Source Intelligence (OSINT) information [21].

During the past two decades, a variety of approaches have been developed to detect vulnerabilities before they can be exploited. These can be organized into three categories: static analysis tools, machine learning and deep learning methods, and, more recently, approaches based on Large Language Models. This section focuses on the first two categories, outlining the landscape in which LLM-based approaches emerged. The following subsections examine these approaches in detail, starting with static analysis tools.

2.1.1 Static Analysis Security Testing (SAST)

Static Analysis Security Testing (SAST) tools detect vulnerabilities by analyzing the source code without executing it, operating as a white-box approach [22]. By examining the code directly, without the need to deploy or run the application, SAST tools can be integrated early in the development lifecycle. These tools can generate reports highlighting alerts and potential risks that could lead to security breaches [23, 24], and can identify potential errors and security vulnerabilities in the code [25]. Many of these tools are predefined with certain rules based on

CWEs or CVEs [26]. Other approaches, such as static taint analysis, use data flow to discover vulnerabilities by testing different inputs [26]. Taint analysis tracks the propagation of untrusted input through the program's data flow, from *sources* (points where external data enters the system) to *sinks* (points where that data is used in potentially dangerous operations), checking whether any *sanitizer* - a transformation that neutralizes the risk - intervenes along the path. SAST tools have evolved significantly over time, moving from small lexical analysis to a set of complex techniques that provide more sophisticated and comprehensive feedback [26]. SAST tools are used during the developers' workflow because threats arise during the development process, and it is easier to address them at this stage. Efficient integration of SAST is sought using Continuous Integration and Continuous Deployment (CI/CD), employing automation to prevent emerging threats. In a CI/CD environment, the application can be deployed in a testing environment, where vulnerabilities detected by SAST analysis are identified and corrected. However, it is important to consider that some SAST tools may not detect certain specific issues, which could be overlooked in the process [27]. A recent survey on barriers of SAST adoption [8] identifies several limitations. A primary concern is the high rate of false positives, which significantly reduces developers' trust and wastes time due to manual verification efforts, often overwhelming developers with large result files difficult to interpret. Additionally, many SAST tools require time-consuming setup and configuration, discouraging integration into the development environment. The lack of automated fix suggestions increases manual effort, while limited integration with IDEs and development pipelines disrupts workflow and reproducibility. Other reported challenges include insufficient customization, limited language support, scalability issues, and invalidated or inconsistent metrics literature tool versions. A particularly consequential limitation is the inability of rule-based tools to generalize beyond the vulnerability patterns explicitly modeled in their query libraries. When a new framework introduces a novel API for handling user input, or when a vulnerability arises from an unusual combination of library calls, rule-based SAST tools will miss it unless a human expert authors and deploys a new detection rule. Altogether, these technical limitations highlight that a successful adoption of SAST tools depends not only on detection capabilities but also on usability, workflow integration, and developer trust.

Among SAST tools, CodeQL, developed by GitHub, occupies a prominent position. It is a language and tool chain for code analysis designed to allow security researchers to scale their knowledge of a single vulnerability to identify variants of that vulnerability across a wide range of code-bases [28]. It is also designed to allow developers to automate security checks and integrate them into their development workflows. Querying code using CodeQL is the most efficient way to perform variant analysis, which is the process of using a known security vulnerability as a seed to find similar problems in the codebase. The standard CodeQL queries can be used

to identify seed vulnerabilities, or researchers can write custom CodeQL queries to find new variants that could be missed during traditional manual techniques.
The CodeQL analysis consists of three steps [28]:

1. **Preparing the code by creating a CodeQL database**: To create a database, CodeQL first extracts a single relational representation of each source file in the codebase. After extraction, all the data required for analysis - relational data, copied source files, and a language-specific database schema specifying the mutual relations in the data - is imported into a single directory known as a CodeQL database. This relational representation enables queries to express complex structural and semantic properties of the code, including data flow, control flow, and call graph relationships.

2. **Running CodeQL queries against the database**: Once the CodeQL database has been created, one or more queries are executed against it. CodeQL queries are written in QL, a specially-designed object-oriented query language. Queries can be run using the CodeQL for VS Code extension or the CodeQL CLI, and can be sourced from the official CodeQL repository or written as custom rules by security researchers.

3. **Interpreting the query results**: The final step converts results produced during query execution into a form that is more meaningful in the context of the source code. Queries contain metadata properties that indicate how the result should be interpreted. Following interpretation, results are output for code review. In CodeQL for Visual Studio Code interpreted query results are automatically displayed in the source code. Results generated by the CodeQL CLI can be output in a number of different formats for use with different tools - in this thesis, the output format used is the SARIF format, a standardized JSON-based schema for static analysis results that facilitates interoperability with downstream tooling.

A large-scale empirical study conducted on 258 open-source embedded projects demonstrated that CodeQL detected 709 real-world defects, of which 75% were classified as potential security vulnerabilities [6]. However, CodeQL shares the fundamental limitation of all rule-based SAST tools: its effectiveness is entirely dependent on the completeness of its query library. When sources, sinks, or sanitizers specific to a given framework or library are not explicitly modeled in the query, the corresponding vulnerabilities go undetected [9]. Furthermore, the manual maintenance of the query library is resource-intensive and structurally reactive: new vulnerability classes only become detectable after a corresponding rule has been authored, reviewed, tested, and deployed, creating an inherent lag between the emergence of novel attack patterns and the availability of detection rules. This limitation is central to the motivation of this thesis, as the proposed

multi-agent architecture specifically targets the identification and remediation of such coverage gaps through LLM-assisted query synthesis.

2.1.2 Machine Learning and Deep Learning Approaches

The limitations of rule-based static analysis - particularly their reliance on manually crafted rules (such as CodeQL queries), high false positive rates, and inability to adapt to novel vulnerability patterns - have motivated the exploration of data-driven approaches that can learn to identify vulnerabilities directly from source code. Over the past ten years, this field of research has evolved through three distinct phases: early applications of traditional machine learning algorithms on handcrafted code features, the adoption of deep learning architectures capable of learning representation automatically, and, most recently, the use of pre-trained language models [17].

Early data-driven approaches relied on traditional machine learning algorithms trained on handcrafted features extracted from source code. A survey by Hanif et al. [17], which analyzed 90 research papers published between 2011 and 2020, found out that supervised learning was the most widely adopted paradigm for vulnerability detection. Among the most common algorithms it cites Random Forest, Support Vector Machine (SVM), Naïve Bayes, Logistic Regression, and Decision Tree, applied to features such as software metrics, static code attributes, and source code tokens. These approaches achieved detection accuracy up to 95% in controlled settings [29]. Ensemble methods, especially Random Forest-based meta-classifiers, have been explored to combine the predictive power of multiple models, achieving accuracy scores of up to 84% with oversampling techniques for managing class imbalance [30]. However, these approaches suffer from significant limitations. First, their performance is highly dependent on the quality of manually created features, which often fail to capture the actual semantic and syntactic representations of source code [31]. Second, classical software metrics have been shown to be indirectly related to code vulnerabilities and are unable to specify particular levels of granularity [32]. These constraints motivated the adoption of deep learning approaches capable of automatically learning more expressive code representations.

Deep learning has rapidly emerged as the dominant paradigm for vulnerability detection, accounting for over 31% of the research papers surveyed by Hanif et al. [17]. Unlike traditional machine learning, deep learning architectures can automatically learn hierarchical feature representations from raw code inputs, substantially reducing the dependence on manual feature engineering. Among the several architectures adopted, recurrent models based on Long Short-Term Memory (LSTM) and its bidirectional variant (BiLSTM) have demonstrated strong results. Li et al. [33] proposed VulnDeePecker, a BiLSTM-based system that achieved precision

up to 94.6% and discovered four previously unreported vulnerabilities that had been patched by developers. Russell et al. [34] implemented Convolutional Neural Network (CNN) and Recurrent Neural Network (RNN) architectures, achieving 95.4% Area Under the ROC Curve (AUC) score on the SATE IV Juliet Suite Dataset, which means that the model is very accurate with a low false positive rate. Graph-based approaches have also gained significant interest: Zhou et al. [35] proposed Devign, a Graph Neural Network (GNN) that learns comprehensive program semantics from composite code representations, capturing multiple structural relationships within the code - including abstract syntax trees, control flow graphs, and data flow graphs - in a unified graph representation. The advent of pre-trained language models for code further shifted the paradigm from training task-specific architectures from scratch toward fine-tuning models that already encode broad programming knowledge, enabling effective transfer learning even in low data regimes.

Despite these promising results, several fundamental limitations prevent these approaches from being reliably deployed in production environments. A first issue concerns dataset quality: the lack of standardized and gold-standard datasets remains a major obstacle, as many studies rely on synthetic datasets such as Software Assurance Reference Dataset (SARD) [36], which do not reflect the complexity of real-world vulnerabilities. Different studies also adopt different private datasets, making reliable cross-study comparison difficult [17]. A second limitation concerns semantic understanding: although deep learning models are better than manually created features because they automatically learn representations, they still struggle to explain why a specific code fragment is vulnerable. Current interpretation methods provide only superficial explanations and lack a deeper contextual understanding [37], which is a significant barrier in a security context where explainability is a regulatory and operational requirement. A third challenge is generalization: most existing models focus predominantly on binary detection (vulnerable vs. non-vulnerable) for a limited set of vulnerability types - mainly buffer overflow, SQL Injection, and cross-site scripting - limiting their practical applicability to the diverse and evolving landscape of real-world software vulnerabilities [17].
These limitations - dataset scarce quality, lack of deep semantic reasoning, and poor generalization beyond controlled benchmarks - have motivated researchers to explore Large Language Models as the next paradigm for vulnerability detection, which, pre-trained on more parameters and with their capacity for contextual and multi-step reasoning, can overcome these barriers, as discussed in the following section.

2.2 LLM-based Vulnerability Detection

The limitations outlined in the previous section, particularly the rigidity of static analysis rules and the restricted generalization capability of traditional ML/DL models, have motivated the exploration of a different paradigm: leveraging Large Language Models (LLMs) for vulnerability detection. Unlike prior approaches that rely on rules manually created or task-specific architectures trained on limited labeled datasets, LLMs are pre-trained on large amounts of code and natural language, enabling them to capture deep semantic patterns and reason about code in ways that go beyond syntactic pattern matching [10]. Crucially, LLMs can contextualize a code snippet within its broader environment, incorporating and seeing surrounding function logic, API documentation conventions, variable naming semantics, and developer comments. This shift has been accelerated by the integration of LLMs into developer tools such as GitHub Copilot and Cursor, which offer real-time vulnerability analysis together with code generation capabilities [38].
The early applications of LLMs to vulnerability detection adopted a straightforward approach: provide the model with a target code snippet and directly query whether it containes vulnerabilities [39]. While fine-tuning techniques can improve detection accuracy [40], they require the modification of model parameters and retraining, making them impractical for many applications. As a consequence, research has increasingly focused on two enhancement strategies that do not alter the core model weights.
The first is *Retrieval Augmented Generation (RAG)*, which improves the accuracy of LLM responses by retrieving relevant knowledge from external sources and integrating input prompts with this information. This approach addresses the limitations of LLMs in domain-specific knowledge gaps while mitigating hallucinations by basing responses on retrieved evidence rather than relying solely on parametric memory. In vulnerability detection, RAG typically retrieves code snippets with known vulnerabilities or patches that are semantically similar to the target code [41, 42]. However, RAG relies only on semantic similarity to determine which external knowledge to incorporate, without guaranteeing that the target code shares the same vulnerabilities as the retrieved snippets [38]. This can introduce noise or even mislead the model when the retrieved examples, though apparently similar, differ in the specific conditions that make them exploitable.
The second strategy is *Reasoning-Enhanced Vulnerability Detection*. Recent studies show that reasoning significantly improves both accuracy and interpretability in LLM-based vulnerability detection systems [**un2024llm4vuln**, 43, 44]. Reasoning exploits the advanced logical capabilities of LLMs by breaking down complex problems into smaller, verifiable reasoning steps [43]. Furthermore, during the reasoning process, LLMs demonstrate self-reflection [45] and adaptive decision-making skills, which allow them to refine their analysis and correct errors. This

iterative refinement mechanism effectively improves reasoning about vulnerabilities and reduce the incidence of overconfident incorrect conclusions[46].
Despite these evolutions, the effectiveness of LLM-based vulnerability detection remains a subject of active debate. The security community has formed a series of influential (but potentially erroneous) consensus beliefs based on various assessments [38]: that LLMs are *unreliable* - since their vulnerability detection capabilities are comparable to random guesses; *insensitive* to code patches - unable to distinguish vulnerable code from its patched counterpart; and that performance has *reached a plateau* across model scales. However, Li et al. [38] challenge these conclusions by demonstrating that they are artifacts of *context-deprived evaluations*. Their CORRECT (Context-Rich Reasoning Evaluation of Code with Trust) evaluation framework, which systematically incorporates contextual information into LLM-based vulnerability detection, shows that state-of-the-art models achieved up to 67% accuracy and precision approaching 0.8 when provided with sufficient context. Moreover, their analysis shows that most false positives derive not from an inability to recognize patches, but from reasoning errors in which the model incorrectly considered a patch insufficient to eliminate the vulnerability. These findings suggest that the primary bottleneck for LLM-based detection lies not in binary classification capability, but in the quality of contextual reasoning about the root causes of vulnerabilities.
Nevertheless, important limitations persist. LLMs demonstrate weaker performance on rare or out-of-distribution vulnerabilities compared to common patterns such as out-of-bounds access or integer overflow [38]. Recall remains a critical challenge, with some state-of-the-art models that achieve recall around 0.5, indicating that a substantial portion of real vulnerabilities remain undetected. Additionally, reasoning models exhibit an *overthinking* tendency, where extended deliberation can lead the model to continuously change the initially correct conclusions in favor of a more complex but incorrect one.
These results point to a fundamental insight: rather than treating programs as simple text, future LLM-based detection systems should integrate LLMs with program analysis tools — such as static analyzers and property graphs — to build context-specific vulnerabilities that support more accurate reasoning [38]. Given these findings, Li et al. [38] suggest that future LLM-based detection systems should move beyond treating programs as plain text and instead integrate LLMs with program analysis tools to construct vulnerability-specific context that supports more accurate reasoning.

2.3 LLMs and LLM-based Agents

With the rise of Large Language Models (LLMs), such as OpenAI GPT-4, DeepSeek-R1, and PaLM 2, LLM-based agents are unlocking new possibilities for autonomous reasoning and action[47]. Serving as the *brain* of AI agents, LLMs empower them with advanced capabilities in human-machine interaction, few/zero-shot planning, contextual understanding, knowledge retention, and general-purpose task solving across physical, virtual, or mixed-reality environments [48, 49]. The main distinction between LLM-based agents and earlier AI systems lies in the combination of flexible natural language understanding with the ability to take actions in an environment to pursue a goal, enabling agents not only to produce text, but also to plan, use tools, observe the effects of their actions, and revise their behavior accordingly.
LLM agents are generally divided into two broad categories. First, *Virtual LM agents*, such as AutoGPT [50] and AutoGen [51], that can autonomously interpret human instructions and leverage external tools to gather information and complete complex, multi-step tasks [52]. Second, *Embodied LM agents*, such as FigureAI's and Tesla's Optimus, that directly communicate with the physical world, perceiving their surroundings through sensors and acting upon them through robotic actuators [53]. In both cases, the key distinguishing feature compared to earlier AI systems is the shift from narrow, task-specific competence toward flexible, goal-directed autonomy [47]. This thesis is concerned exclusively with virtual agents operating in software development and security analysis environments.

2.3.1 Architecture of LLM-based Agents

Despite the variety of existing agent designs, the literature has converged on a common modular architecture consisting of five interconnected components: a planning module, a memory module, an action module, an interaction module, and a security module [49, 54, 48]. Together, these modules enable the agent to perceive its environment, reason about goals, retain relevant knowledge, execute actions, and operate safely and responsibly.
The *planning module* serves as the core of the agent [55, 49], applying advanced reasoning techniques to devise effective solutions for complex problems. Several prompting paradigms have been proposed to enable structured reasoning LLM-based agents. Chain-of-Thought (CoT) [56] decomposes tasks into a linear sequence of intermediate reasoning steps, improving the reliability of the model's conclusions on multi-step problems. Tree-of-Thought (ToT) [57] and Graph-of-Thought (GoT) [58] extend this idea to non-linear reasoning, enabling the agent to explore multiple reasoning paths simultaneously before committing the final answer. Feedback -enhanced methods such as ReAct [16] and Reflexion [59] go further, allowing agents to iteratively revise their plans based on environment's feedback. The ReAct

paradigm is of particular relevance to this thesis, as it is the one adopted for all three agents in the proposed architecture.
The memory module governs how the agent stores and retrieves information to learn and adapt over time [54]. Short-term memory manages contextual information about the current situation. It is temporary and limited, typically managed through a contextual window that limits the amount of information that the agent can learn in a certain moment [60]. Long-term memory stores the historical behaviors and thoughts of the agent through an external vector storage, which allows rapid retrieval of important information [61]. Hybrid memory that synergies both types has been shown to improve reasoning over extended time horizons [62]. Retrieval-Augmented Generation (RAG) [63] further enriches agent memory by dynamically fetching relevant information from external knowledge sources at inference time, reducing the risk of outdated or hallucinated responses.
The action module provides the agent with the capability to translate thoughts and plans into concrete operations, whether in physical environments through embodied actions [53] or in digital environments through tool use - such as invoking APIs, browsing the web, or executing code [54]. The availability of the tool set can significantly expand the scope of tasks an agent can autonomously address, making tool design a critical aspect.
The *interaction module* governs communication between the agent and its environment, human users, and other agents [64]. Natural language serves as the primary interface for human-agent interaction, while agent-agent communication enables coordination, knowledge exchange, and collaborative problem-solving in a multi-agent setting. The incorporation of theory-of-mind capabilities [65] - the ability to model the beliefs, intentions, and goals of other agents - has been shown to further enhance the quality of such collaborative interactions.
Finally, the *security module* is integrated across all other components to ensure that the agent's action remains safe, ethical, and privacy-preserving [66], monitoring agent decisions and outputs to prevent harm and maintain compliance with applicable legal and ethical standards. In the context of security analysis, this module also ensures that the agent does not generate harmful artifacts, such as exploitable code, as a product of its analysis.
These individual modules, when instantiated across multiple coordinated agents, give rise to a more powerful architectural paradigm known as *multi-agent systems*, in which the collective capabilities of specialized agents can be leveraged to address problems that exceed the scope of any single agent operating in isolation. Although a single LLM-based agent can address a wide range of tasks, many real-world problems are too complex to be solved in isolation, motivating a growing interest in this type of system [64]. In such systems, specialized agents collaborate by dividing work with parallel *horizontal collaboration* or sequentially *vertical collaboration*, with hybrid architectures combining both paradigms [67, 68]. Some examples

include ChatDev [69], MetaGPT [68], and AutoGen [51], which have demonstrated strong performance across domains such as software engineering and scientific reasoning.
Cross-agent verification - where one agent reviews the conclusions of another - has also been identified as a particularly effective mitigations for hallucination [15].
In the context of software vulnerability detection, multi-agent architectures can be particularly interesting, as the inherent complexity of security analysis naturally decomposes into distinct sub-tasks - result validation, coverage gap identification, and rule synthesis - each assigned to specialized agents, enabling verification between agents to reduce hallucinated results. This decomposition principle is the architectural foundation of the multi-agent system proposed in Chapter 3.

Chapter 3

Methodology

3.1 Proposed architecture and Workflow

This chapter presents the architecture designed and implemented in this thesis, which consists of three LLM-based agents that collaborate autonomously to investigate the three research questions introduced in Section 1.2. Each agent is in charge of a different phase of the pipeline and addresses a different goal. The first agent, the **AnalyzerAgent**, investigates how to address question **RQ1** by autonomously validating the results produced by CodeQL through contextual code reasoning, filtering false positives, and identifying false negatives missed by the static analysis tool. The second agent, the **SuggestorAgent**, addresses question **RQ2** by systematically analyzing the validation results to identify structural gaps in existing CodeQL queries. The third agent, the **CreatorAgent**, addresses question **RQ3** by generating new CodeQL queries to overcome the identified gaps, with the goal of reducing manual effort required to extend the existing query suite.
Rather than relying on a standalone LLM model, the system adopts the multi-agent paradigm discussed in Section 2.3.1, decomposing the overall problem into sequential phases, each assigned to a specialized agent. This decomposition is not only organizational; rather, it represents a deliberate design choice aimed at reducing the cognitive load of the overall pipeline, and enabling the independent evaluation of each reasoning stage. The overall structure of the system is illustrated in Figure 3.1, and the details of single components, their roles, and their interactions are described in the following sections.

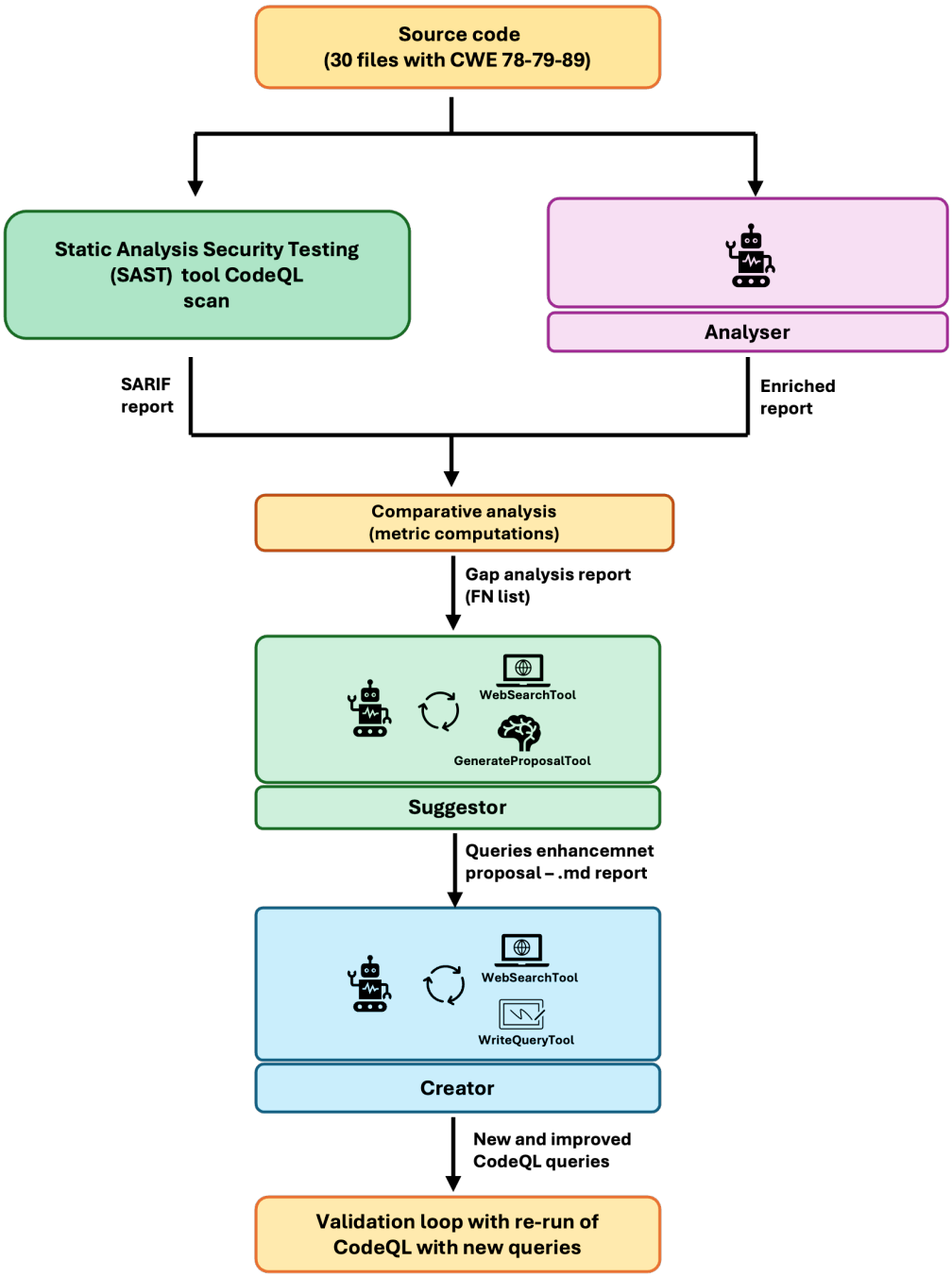

Figure 3.1: Overview of the overall multi-agent architecture.

The first of the three agents is the **Analyzer agent**, implemented using the

ReAct framework[1]. The agent's workflow begins with a call to the CodeQL tool, which scans the provided dataset and reports detected vulnerabilities in a standardized SARIF format. In parallel, the AnalyzerAgent performs its own independent analysis on the same dataset, generating a separate report with its own vulnerability assessments and CWE classifications. The two reports are then individually evaluated and compared against the ground truth to compute detection metrics - true positives, false positives, false negatives, recall, precision, and F1-score[2] - allowing a direct comparison of performance and coverage gaps between the SAST tool and the LLM-based agent.
The second agent, **SuggestorAgent**, also implemented as a ReAct agent, receives the AnalyzerAgent's output and identifies structural gaps in the existing CodeQL queries - specifically missing sources, sinks, and sanitizers - producing a detailed vulnerability report including concrete suggestions for new and improved queries. Finally, the **CreatorAgent**, also a ReAct agent, receives the SuggestorAgent's report and, for each targeted CWE requiring coverage, generates a corresponding `.ql` file through several reasoning steps and tool invocations. Crucially, CreatorAgent executes an iterative *build-and-fix* cycle, where, after producing an initial query draft, it compiles and validates the query, reasons over any errors or warnings returned by the CodeQL compiler, and corrects the query accordingly before proceeding to the next CWE. The final output of the pipeline is a directory of new CodeQL queries, ready to be tested and, after human expert review, integrated into the existing query suite.

3.2 Agents and their roles

3.2.1 Analyzer Agent

The Analyzer is the specialized component that operates independently from the CodeQL SAST tool to perform its own security analysis of the source code. While CodeQL produces a separate vulnerability report through static analysis, the Analyzer generates its own assessment using LLM-based contextual reasoning. The two reports are produced independently and their performance is then compared against the ground truth, providing a quantitative evaluation of the detection gaps between the SAST tool and the LLM-based agent.

---

[1]By definition, a ReAct agent is an AI agent that uses the "reasoning and acting" (ReAct) framework to combine chain of thought (CoT) reasoning with external tool use. The ReAct framework enhances the ability of a large language model (LLM) to handle complex tasks and decision-making in agentic workflows [16].

[2]Recall $= \frac{TP}{TP+FN}$, Precision $= \frac{TP}{TP+FP}$, $F_1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$

To achieve this, the agent performs a series of reasoning steps. It operates according to the ReAct (Reasoning and Acting) paradigm, which enables it to formulate an explicit thought before executing an action, emulating a deliberate human reasoning process. The agent is initialized with a system prompt that defines its role as an `"Expert security auditor"`.
This iterative reasoning process is derived from the agent-based architecture proposed in *CyberSleuth: Autonomous Blue-Team LLM Agent for Web Attack Forensics* [**cybersleuth**] and is managed by three specialized procedures:

- **Summary procedure**: Summarizes the current state and previous observations from other agents, ensuring the AnalyzerAgent is constantly informed of the evolving context.
- **Thought procedure**: Enables the agent to generate logical reasoning based on the summary, the information stored in the Shared Memory, and the last action performed.
- **Action procedure**: Determines the subsequent action based on the thought, the summary, and the context contained in the Shared memory.

The three procedures are implemented as sequential calls inside the agent's reasoning loop. First, the `scratchpad` is populated with the context received from the Shared Memory and the `instructions` are loaded from the agent's prompt template. The summary procedure then takes these two inputs and produces a summarized review of the current state. Next, the thought procedure receives the summary, the scratchpad, and the last step performed, generating a logical chain of reasoning about what to do next. Finally, the action procedure takes the summary, the scratchpad, the last step, the thought, and the list of available tools, and determines which action to execute. The result of each action is concatenated into the scratchpad - the agent's short-term memory - allowing the agent to augment the information at its disposal progressively. Other agents can retrieve this information from the Shared Memory whenever needed. The output of each procedure is the input of the next, creating a coherent chain of decisions.
The AnalyzerAgent's workflow, implemented in the `run_analysis` method, is organized as follows. First, the agent invokes the *CodeQLSASTTool*, which executes static analysis in the source code files. The tool returns a SARIF (Static Analysis Results Interchange Format) report, which is then parsed by the `ParseSarifTool` and converted into a unified format (`UnifiedReportModel`) containing, for each file, a list of detected vulnerabilities with their CWE classification, description, and line number that indicates the crash location (i.e., where the vulnerability is executed). This output is saved as `codeql_report.json`.
Independently from the CodeQL scan, the AnalyzerAgent performs its own scan analysis on every file in the dataset through the `run_agent` method. For each file, the agent resets its reasoning context to avoid cross-file contamination and

constructs an analysis prompt containing the full source code with instructions to perform a complete security audit. The agent then starts its ReAct reasoning loop, up to a maximum of 20 steps, during which it examines data flows from sources to sinks. The loop terminates when the agent calls `FinishTool` with a structured JSON report listing all identified vulnerabilities. The agent's results are stored in the same unified format as CodeQL's output and saved as `agent_report.json`.
Once both reports are produced, they are evaluated against the ground truth to compute detection metrics. The `compute_metrics` method calculates true positives, false positives, false negatives, precision, recall, and F1-score, while the `compute_metrics_per_cwe` method computes these metrics per individual CWE category. The `compare_results` method performs a file-by-file comparison, classifying each file as `exact_match`, `partial_match`, `missed`, or `wrong` for both the agent and CodeQL, and computing the agreement rate between the two approaches. Finally, the `build_gap_analysis` method identifies files where CodeQL failed to detect vulnerabilities that the agent successfully identified, grouping them by CWE category. This gap analysis will constitute the primary input for the SuggestorAgent.
All these instructions and tasks are encapsulated in a system prompt, which serves as the essential operational specification that defines the agent's behavior. The prompt defines the agent's role, its reasoning workflow, and the mandatory output format. Consequently, the design of the prompt is a critical phase of implementation, since even minimal modifications can alter the agent's output. An effective prompt must function as a rigorous operational protocol: it must assign a clear and specific role, provide the complete context required for the task, and impose a structured workflow with well-defined steps in order to reliably transform the reasoning of the agent into the expected results. In the AnalyzerAgent's prompt, the most important rules and constraints are explicitly repeated at multiple points to ensure they are consistently followed across all reasoning steps. Furthermore, the prompt is not treated as a static component, but it is iteratively refined based on observations of the agent's output and the evolution of the system requirements, progressively aligning the prompt with the structural and semantic characteristics of the desired output.
At the end of its execution, the AnalyzerAgent has produced all necessary outputs: the CodeQL report, the agent's independent analysis report, the detection metrics, the file-by-file comparison, and the gap analysis. The gap analysis is passed to the SuggestorAgent to drive the structured production of improved CodeQL query proposals. The presence of detection gaps - particularly false negatives that CodeQL missed but the agent identified - provides the concrete motivation for designing more accurate and coverage-complete detection queries, a task handled by the SuggestorAgent in the following phase.

3.2.2 Suggestor Agent

Adopting a multi-agent architecture, rather than a single agent, is essential to ensure separation of responsibilities, reduce the model's cognitive load per invocation, and improve the accuracy of individual tasks. By dividing the work, each agent focuses on a specific goal, avoiding the risk of overloading a single model with too many different instructions that could introduce confusion and reduce the accuracy of the output. While the AnalyzerAgent focuses on scanning the code and evaluating detection performance, the SuggestorAgent uses those results to produce targeted and actionable improvements to CodeQL's detection capabilities.

The SuggestorAgent is also a ReAct agent, and it represents the proactive component of the architecture. Its main goal is to transform the AnalyzerAgent's results into operational intelligence for the development of enhanced, predictive CodeQL queries - going beyond mere vulnerability evaluation toward a structured strategy for closing detection gaps.
To fully understand the overall flow of this thesis, it is important to have a basic understanding of how standard CodeQL queries are constructed. The standard approach to building CodeQL path queries consists of modeling the entire propagation path that potentially dangerous data may follow through the program, by identifying three key components:

- **Source**: The entry point where external, potentially untrusted data enters the application. Accurately defining sources is the first step in instructing CodeQL on which data flows to track. Sources typically correspond to user-controlled inputs such as HTTP request parameters, file contents, or environment variables.
- **Sink**: The execution point where data is used to perform sensitive operations, such as a SQL query, a shell command, or a file path construction. Correctly identifying sinks is critical, as they represent the final destinations that the scanner must monitor to detect a completed exploit path.
- **Sanitizer**: A security filter - typically a validation or encoding function -that neutralizes the risk associated with tainted data. Identifying sanitizers enables more precise queries that avoid false positives by discarding data flow paths where the risk has already been neutralized before reaching the sink [28].

The agent operates via a modular architecture, delegating specific tasks to dedicated tools. It is equipped with three tools, which it can autonomously invoke whenever necessary:

- `WebSearchTool`: Allows the agent to interact with the external environment by accessing real-time information via the DuckDuckGo search engine [70].

Search results are automatically summarized by the agent to extract only relevant CodeQL class names, Python library methods, AST elements, and TaintTracking configuration patterns, discarding irrelevant or redundant content.

- `GenerateProposalTool`: Generates a structured improvement proposal (`QueryProposal`) for a specific CWE. When invoked, the tool receives the existing CodeQL query, the source code examples that CodeQL missed, the AnalyzerAgent's reasoning for those files, and any web search findings, and passes them to an LLM call that produces a proposal with defined sections: existing coverage, identified gap, missing sources, missing sinks, missing sanitizers, proposed additions, and estimated impact.

- `FinishToolSuggestor`: Terminates the agent's execution. The agent is prevented from calling this tool until all CWEs have been processed, ensuring completeness of coverage.

Specifically, the SuggestorAgent receives as input the gap analysis produced by the AnalyzerAgent, which contains:

- The CWEs for which CodeQL produced false negatives confirmed by the AnalyzerAgent.

- The file names and source code of the examples that CodeQL failed to detect.

- The AnalyzerAgent's reasoning explaining why those files are vulnerable.

- The existing CodeQL queries for each CWE, which serves as the baseline for improvement.

After initialization, the agent parses the gap analysis and identifies the set of CWEs to process. For each CWE, the agent follows a repetitive cycle: it may optionally invoke the `WebSearchTool` to gather additional information about specific APIs, libraries, or CodeQL modeling patterns (with a maximum of three searches per CWE to avoid excessive token usage), and then calls the `GenerateProposalTool` to produce the structured proposal. The `GenerateProposalTool` constructs a detailed prompt containing the existing query, the missed source code examples, the AnalyzerAgent's audits, and any web search findings, and invokes the LLM to produce a `QueryProposal` with the following fields:

- `existing_queries_summary`: What the current query already covers.

- `gap_description`: The specific pattern or library construct that is missing from the existing query.

- `missing_sources`: AST source nodes to add (e.g., `FastAPI: Request.query_params.get()`.
- `missing_sinks`: AST sink nodes to add (e.g., `SQLAlchemy: session.execute` with f-string argument).
- `missing_sanitizers`: Patterns that prevent exploitation (e.g., `SQLAlchemy: bindparams()` parameterized query).
- `proposed_additions`: Python-mapped AST pseudo-code describing what to modify and how in the existing query.
- `estimated_impact`: A qualitative estimation of false negative reduction expected from the proposed changes.

The agent tracks which CWEs have been processed and prevents the `FinishTool-Suggestor` from being while any CWE remains pending. Once all CWEs are processed, the agent compiles the proposals into a structured Markdown report and passes the final `SuggestorOutput` to the CreatorAgent.
As with the AnalyzerAgent, the SuggestorAgent's system prompt is a critical design artifact. The prompt defines the agent as a `CodeQL Security Expert specializing in query improvement` and provides a structured workflow: for each CWE, use the existing query as the baseline, optionally search for additional information via `WebSearchTool`, and invoke `GenerateProposalTool` exactly once per CWE. The prompt is iteratively refined based on the observed output quality, with particular attention to ensuring that the proposals contain sufficiently specific and actionable technical detail.
The output of the SuggestorAgent is a detailed Markdown report structured by CWE, where each section contains the existing coverage summary, the identified gap, the missing sources/sinks/sanitizers, the proposed additions, and the estimated impact. This report serves as the primary input of the CreatorAgent, providing concrete and actionable technical specifications for the development of the enhanced CodeQL queries.
In summary, the SuggestorAgent functions not as a simple query corrector but rather as an autonomous security architect within the multi-agent pipeline. It synthesizes the AnalyzerAgent's findings into structured improvement proposals that move beyond enumerating defects toward a targeted strategy for extending CodeQL's detection capabilities. By transforming qualitative feedback into precise technical constraints (missing sinks/sources/sanitizers) the agent bridges the analytical and generative phase of the pipeline, providing the CreatorAgent with the rigorous technical guidelines needed to produce syntactically and logically correct CodeQL queries.

3.2.3 CreatorAgent

Like the AnalyzerAgent and the SuggestorAgent, the CreatorAgent is a ReAct agent. Its main task is to analyze the proposals provided by the SuggestorAgent, identify all the targeted CWEs that require CodeQL query enhancement, and iteratively produce one `.ql` file per CWE through a structured sequence of reasoning steps and tool invocations. As with the previous agents, the CreatorAgent is initialized with a system prompt that defines it as an `"Autonomous CodeQL Engineer"`, and its reasoning process follows the same Summary-Thought-Action cycle adopted by the other agents in the pipeline.

A key design principle of the CreatorAgent is that it does not generate queries from scratch. Instead, it receives the existing CodeQL queries as structural templates and extends them based on the gap analysis produced by the SuggestorAgent. Specifically, the agent preserves the import statements, module structure, and class hierarchy from the existing query, and appends new predicates and class extensions to model the missing sources, sinks, and sanitizers identified in the proposals. This approach ensures that the generated queries maintain structural compatibility with the CodeQL standard libraries while targeting the specific detection gaps identified by the upstream agents. It also reduces the risk of suntactic hallucinations that would arise if the agent were asked to construct a valid QL module from first principles.
The CreatorAgent has three tools at its disposal:

- `WebSearchTool`: The same tool used by the SuggestorAgent. It is invoked when the agent needs to explore and deepen its knowledge about the latest CodeQL libraries or terminologies, or during the validation phase to correct any errors.

- `WriteQueryTool`: The core execution tool of the CreatorAgent. When invoked with a CWE identifier and the complete `.ql` query code, it writes the query file to disk, installs the CodeQL pack dependencies, and automatically compiles the query using `codeql query compile -check only` command. If compilation succeeds, the tool returns a success message; if it fails, it returns the compiler error with instructions for the agent to analyze the error, optionally search for the correct API usage, and retry with a corrected version.

- `FinishToolCreator`: Terminates the agent's execution. As with the SuggestorAgent, the agent is prevented from invoking this tool until all targeted CWEs have been processed.

The CreatorAgent receives as input the report of the SuggestorAgent, and loads from the Shared Memory the list of CWEs to process and the existing CodeQL

queries that serve as templates. For each CWE, the agent reads the existing query and the corresponding section of the proposal report. It then writes the improved query by extending the existing structure with new predicates derived from the proposed missing patterns, and invokes the `WriteQueryTool` to save and compile the result.
If compilation fails, the agent enters an error recovery cycle, where it analyzes the compiler error message, compares the generated code with the existing query template to identify the structural discrepancy, optionally invokes the `WebSearchTool` to verify the correct CodeQL Python API signature, and retries with `WriteQueryTool`. A maximum of three compilation attempts is allowed per CWE. If all three attempts fail, the query is saved with a `failed_` suffix to enable manual inspection, and the agent proceeds with the next CWE without blocking the pipeline. Once all CWEs are processed, the agent calls `FinishToolCreator` to finalize execution.
The CreatorAgent's system prompt defines it as an `Autonomous CodeQL Engineer for Python vulnerability detection` and provides the full context required for each generation step: the list of CWEs to process, the SuggestorAgent's proposal report, and the existing CodeQL queries as starting templates. The prompt also specifies a strict output format for generated `.ql` files, requiring mandatory metadata annotations (`@kind`, `@id`, `@name`, `@problem.severity`) and prohibiting Markdown code fence markers in the output, which would cause compilation failures.
At the end of the CreatorAgent's execution, the `generated_queries` directory is populated with the new `.ql` files containing the enhanced detection rules. Queries that compile successfully are saved directly; queries that failed compilation after three attempts are saved with the `_failed.ql` suffix for subsequent manual review by a security engineer. The quality and correctness of the generated queries are analyzed in the Results chapter (Chapter 5).

3.3 Tools

All the tools used within the project are listed and described below, organized by function.

- `CodeQLSastTool` and `ParseSarifTool`: Used for static analysis and report parsing. The `CodeQLSastTool` automates the full CodeQL database lifecycle - creation, analysis, and SARIF generation - abstracting the underlying CLI commands into a single callable interface. The `ParseSarifTool` transforms the SARIF report into structured JSON, converting it into the unified format (`UnifiedReportModel`) used by both the CodeQL and agent reports to enable direct comparison.

- `GenerateProposalTool`: Used by the SuggestorAgent to generate a structured improvement proposal (`QueryProposal`) for a specific CWE. When invoked, the tool constructs a detailed prompt containing the existing CodeQL query, the source code examples that CodeQL missed, the AnalyzerAgent's reasoning, and any web search findings, and passes it to the LLM to produce a proposal with defined fields: existing coverage, gap identified, missing sources, missing sinks, missing sanitizers, proposed additions, and estimated impact.

- `WriteQueryTool`: Used by the CreatorAgent to write, save, and compile CodeQL query code in the QL language based on the SuggetsorAgent's proposals. The tool receives an improvement proposal and an existing query as a structural baseline, invokes the LLM to produce the enhanced query code, writes it to disk and validates it through a compilation step using the CodeQL CLI.

- `FinishTool`, `FinishToolSuggestor`, `FinishToolCreator`: Termination tools used respectively by the AnalyzerAgent, SuggestorAgent, and CreatorAgent to signal the completion of their tasks. Each tool enforces a completeness condition, checking that all required outputs have been produced, before allowing the agent to finalize its execution.

- `WebSearchTool`: Used by both the SuggestorAgent and the CreatorAgent to interact with the external environment and retrieve updated technical information. The tool is built upon the `duckduckgo-search 8.1.1` [70] Python library, which enables integration with the DuckDuckGo search engine without requiring an official API key. For each query, DuckDuckGo returns the first three results. A `summarize` function is applied to each result to reduce its size and avoid overwhelming the model's context window, retaining only information relevant to CodeQL syntax, Python library APIs, or vulnerability patterns.

3.3.1 SAST Tools and CodeQL

Static Application Security Testing (SAST) tools analyze source code without executing it, operating as a white-box approach to vulnerability detection [22]. Many of these tools are predefined with certain rules based on CWE (Common Weakness Enumeration) or CVE (Common Vulnerabilities and Exposures) taxonomies, which describe known vulnerability classes and provide a framework for automated detection. SAST tools are also designed to integrate into CI/CD pipelines, enabling security checks to be applied early and consistently during the development lifecycle.
The SAST tool used in this thesis is **CodeQL**, developed by GitHub. CodeQL is

designed to allow security researchers to scale their knowledge of a single vulnerability to identify structural variants of that vulnerability across large and diverse codebases, and also to allow developers to automate security checks as part of their standard development workflows [28].

How CodeQL works. Unlike rule-based SAST tools that operate on lexical patterns or simple AST traversal, CodeQL models program semantics by extracting the entire codebase into a relational database. This database encodes rich structural information about the program - including the abstract syntax tree, control flow graph, data flow graph, and call graph - using a language-specific schema. Queries are then written in QL, a declarative, object-oriented query language specifically designed for program analysis, and executed against this database using standard relational reasoning. This architecture enables CodeQL queries to express complex properties of program - such as the flow of untrusted data through multiple function calls across module boundaries - that would be infeasible to detect with simpler pattern-matching approaches.

The analysis pipeline. CodeQL's analysis consists of three steps: first, a language-specific extractor builds a CodeQL database from the source code; second, one or more `.ql` queries are executed against the database; third, the results are interpreted and exported in SARIF format - a standardized interchange format for static analysis results that facilitates integration with downstream tooling such as GitHub Advanced Security and third-party dashboards. In this thesis, the database is created with the command `codeql database create` targeting Python source files, and queries are executed via `codeql database analyze`, producing a SARIF report that is subsequently parsed by the `ParseSarifTool` into the unified format used across the pipeline.

Query structure. CodeQL queries follow a structured format:

```
/**
 *
 * Query metadata
 *
 */

import /* ... CodeQL libraries or modules ... */
```

```
/* ... Optional, define CodeQL classes and predicates ... */

from /* ... variable declarations ... */
where /* ... logical formula ... */
select /* ... expressions ... */
```

The `from` clause declares typed variables over the code entities of interest (e.g., function calls, expressions, parameters). The `where` clause defines logical conditions over those variables using predicates and data-flow libraries, such as the `TaintTracking` module. The `select` clause specifies the results to report, typically identifying the source, the sink, and the data-flow path connecting them.

Path queries and taint tracking. Of particular relevance for this thesis are *path queries*, which track the propagation of potentially tainted data from a source to sink, optionally interrupted by sanitizers. A path query requires the analyst to define three components as QL classes or predicates: (1) a `source` predicate identifying the program locations where external data enters the system; (2) a `sink` predicate identifying the locations where external data is used in a security-sensitive operation; and (3) a `sanitizer` predicate identifying transformations that neutralize the taint. When the `TaintTracking` configuration reports a path from a source to a sink with no intervening sanitizer, the query flags a potential vulnerability. The accuracy of this detection is therefore determined by the completeness and precision of the source, sink, and sanitizer definitions - which is precisely the coverage gap that the SuggestorAgent is designed to identify and that the CreatorAgent is tasked with extending.

Chapter 4

Experimental Setup

4.1 Dataset

The dataset used in this thesis consists of twenty-six real-world Python files extracted from **CVEfixes v1.0.8** [71], a dataset hosted on Hugging Face that collects vulnerability-fixing commits from public repositories indexed in the National Vulnerability Database (NVD). Each file in the dataset contains known vulnerabilities of a specific CWE category, and the ground truth is derived directly from the CVE metadata associated with each commit. The twenty-six files chosen for this thesis contain instances of three different vulnerabilities, chosen based on their prevalence and real-world Python applications, as reported by the National Institute of Standards and Technology (NIST) [72]:

CWE	**Name**	**Description**
CWE-89	SQL Injection	Externally-influenced input is used to construct an SQL command without proper neutralization, allowing an attacker to modify the intended query [73].
CWE-79	Cross-Site Scripting	User-controllable input is not properly neutralized before being included in web page output served to other users [74].
CWE-78	OS Command Injection	The product builds an OS command using external input without properly sanitizing it, allowing attackers to inject and execute unintended system commands. [75].

Table 4.1: Vulnerability categories in the dataset.

CVEfixes contains real source code extracted from open-source projects, making the evaluation more valid and closer to the conditions the system would face in a

real-world adoption scenario.

4.2 Models

The LLM is the cognitive core of each agent in the pipeline. It is responsible for all reasoning, planning, and decision-making that occur within the Summary-Thought-Action cycle: it interprets tool outputs, formulates the next action to take, generates structured proposals, and synthesizes executable code. Therefor, the quality, reliability, and computational characteristics of the chosen model directly determine the performance of the entire system, making the model selection one of the most consequential design decisions in the experimental setup.
This thesis evaluates two OpenAI models as the reasoning engine for all three agents: **GPT-4o-mini** and **GPT-5.2**. Both models are evaluated under identical conditions - same dataset, same prompts, same tools, and same agent logic - so that any difference in output quality can be attributed only to the model's intrinsic capabilities, The results of this comparison are discussed in Chapter 5.

4.2.1 GPT-5.2

GPT-5.2 is the most capable model evaluated in this thesis, offering stronger reasoning, code generation, and instruction-following capabilities than GPT-4o-mini [76]. It is designed for tasks requiring analytical reasoning, precise adherence to complex instructions, and high-quality code synthesis - capabilities that are directly relevant to the demands placed on the agents in this pipeline. In particular, the SuggestorAgent must reason about CodeQL modeling patterns, while the CreatorAgent must generate syntactically valid and semantically coherent QL code.
GPT-5.2 also provides a significantly larger context window than GPT-4o-mini [76]. Although the experimental setup of this thesis does not require extremely long inputs, a larger context capacity provides additional flexibility when handling the cumulative context produced by multi-stage agent reasoning chains. This is particularly relevant in scenarios where the SuggestorAgent report and the existing CodeQL query templates must be processed together.
The main trade-off compared to GPT-4o-mini is a higher inference latency and a higher cost per token, which increases the processing time for full pipeline executions and makes large-scale iterative evaluations more computationally expensive. However, the expected improvements in reasoning capability and code generation reliability justify this choice for the purposes of this comparative evaluation. In particular, the hypothesis guiding this comparison is that the quality of SuggestorAgent proposals and the compilation success rate of CreatorAgent queries may improve when using a more capable model, since these tasks require sustained multi-step reasoning and accurate code synthesis [38].

4.2.2 GPT-4o-mini

GPT-4o-mini is a lightweight, cost-efficient model in the GPT-4o family, designed to enable a broad range of tasks that require low latency and high throughput [77]. It is particularly well suited for applications that chain or parallelize multiple model calls (e.g., calling multiple APIs), pass a large volume of context to the model (e.g., full code base or conversation history), or require fast, real-time responses (e.g., customer support chatbots) [77] - all characteristics relevant to the multi-agent pipeline implemented in this thesis, where three agents execute sequential reasoning loops, each potentially invoking the model many times per run.
Despite its reduced parameter count relative to larger variants of the GPT-4o family, the model demonstrates strong performance on code-related tasks. It achieves a score of 87.2% in the HumanEval benchmark [77], a widely-used evaluation suite for code generation that measures a model's ability to produce syntactically correct and functionally accurate code from natural language descriptions. This syntactic accuracy is relevant for the CreatorAgent, which must translate abstract vulnerability patterns into precise CodeQL predicates and class extensions without resulting in code hallucination, i.e., code snippets that appear syntactically correct, but rather are incorrect or logically inconsistent.
The model supports a context window of 128K tokens [78], which is a practical requirement for this pipeline. The AnalyzerAgent must hold entire source files in context during its security audit loop; the SuggestorAgent processes the full gap analysis report produced by the AnalyzerAgent, which can contain multiple source code examples per CWE; and the CreatorAgent must simultaneously reason over the SuggestorAgent's proposals and the existing CodeQL query templates. A 128K context window ensures that none of these inputs need to be truncated, preserving the full contextual grounding that prior work has identified as the primary determinant of LLM-based vulnerability detection accuracy [38].
In terms of cost and speed, GPT-4o-mini offers a good trade-off for iterative agent workflows.
Together, GPT-4o-mini and GPT-5.2 define a performance spectrum that enables this thesis to characterize the relationship between model capability and the quality of the pipeline's output, and to assess whether the computational overhead of a more powerful model translates into significant improvements in vulnerability detection coverage and the accuracy of query generation.

4.3 Implementation details

The system architecture is based on a multi-agent pipeline implemented using LangChain [79] and LangGraph [80], a graph-based framework for agent development. Each agent - AnalyzerAgent, SuggestorAgent, and CreatorAgent -

is isolated in a dedicated module (`analyzer_agent.py`, `suggestor_agent.py`, `creator_agent.py`) to ensure logical independence. All tools are defined in a shared file (`tools.py`) as structured Pydantic classes. The agents communicate through a `SharedMemory` object that separates short-term and long-term memory, as described below.

4.3.1 Prompt engineering

The system prompt is the primary mechanism through which each agent's role, reasoning workflow, and output format are defined. Prompts are not considered static components, but are iteratively refined based on observations of the agents' outputs and evolving system requirements. This is well demonstrated in the AnalyzerAgent, for which three successive prompt versions were developed. The first version, `SYSTEM_ANALYZER_MINIMAL`, defined a minimal role and simple JSON output schema, serving as a baseline to verify the agent's basic detection capability. `SYSTEM_ANALYZER_GENERIC` introduces a structured five-step methodology (i.e., framework identification, source mapping, sink mapping, data-flow tracing, and CWE classification), with explicit rules to prefer specific CWEs over generic ones. `SYSTEM_ANALYZER_SPECIFIC` further increases precision by introducing a list of banned CWEs, enforcing a rule consisting of one CWE per data-flow, and providing exhaustive source-sink pattern catalogues for the three target vulnerability classes. For this evaluation, the `SYSTEM_ANALYZER_GENERIC` prompt was selected as the operative version. Providing an exhaustive source-sink catalogue, as in the specific version, would risk encoding the ground truth into the prompt itself, rather than testing the agent's autonomous reasoning capability. The generic prompt, instead, outlines a structured methodology without indicating the exact patterns to look for, making the evaluation more representative of a real-world scenario. An example is shown in the excerpt below.

Analyzer Agent — System Prompt (excerpt)

```
STEP 2 - MAP ALL EXTERNAL INPUTS (Sources):
- HTTP request parameters: request.args, request.form,
  request.data, request.json
- URL path parameters (route variables)
- HTTP headers: request.headers, request.cookies
STEP 3 - MAP ALL SENSITIVE OPERATIONS (Sinks):
- SQL queries: cursor.execute(), raw(), extra(),
  ORM .filter() with string formatting
- HTML rendering: render_template_string(), Markup(),
  |safe filter, HttpResponse()
- File operations: open(), os.path.join(), send_file()
STEP 4 - TRACE DATAFLOWS:
- For EACH source identified in Step 2, trace where
  the data flows
- Check if any sanitization/validation exists between
  source and sink
- If user input reaches a sink WITHOUT proper
  sanitization, it IS a vulnerability
```

The SuggestorAgent and CreatorAgent prompts follow a different design implementation, prioritizing workflow enforcement over classification precision. The SuggestorAgent's prompt (`SYSTEM_SUGGESTOR`) defines the agent as a `CodeQL Security Expert specializing in query improvement` and provides an explicit four-step cycle to repeat for each CWE, including a guard that prevents the agent from terminating before all CWEs have been processed. In the CreatorAgent's prompt (`SYSTEM_CREATOR`)the agent is explicitly instructed never to write queries from scratch, but always to copy the existing query structure and extend it, with hard constraints on import paths and identifier naming to prevent hallucinated CodeQL classes or predicate names. An excerpt is shown below.

Creator Agent — System Prompt (excerpt)

```
CRITICAL: The existing queries are REAL, COMPILING CodeQL code.
Use them as your base. Copy their import statements, module
structure, and class hierarchy EXACTLY. Then ADD the missing
sources/sinks/sanitizers from the report.

NEVER invent module names, class names, or predicates. If a name
does not appear in the existing query or a WebSearch result,
do not use it.
```

Finally, the three ReAct procedures - Summary, Thought, and Action - share a common prompt structure that defines the agent as a `Cybersecurity analyst specialized in software vulnerability detection` and enforces a one-step-at-a-time reasoning schema, preventing the agent from executing multiple actions in a single step and ensuring that each decision is based on the most recent observation.

4.3.2 Memory Management Structure

Memory management is handled within the `SharedMemory` class, which uses the Pydantic framework to separate memory into two layers. Short-term memory is implemented as a list of `ReActChain` objects, each recording the complete Summary, Thought, Action, and Observation cycle of a single reasoning step, functioning as the agent's scratchpad. Long-term memory is a persistent dictionary of structured global data, which is not temporary and can be read and written by any agent at any time, enabling information to flow across the pipeline without direct agents' communication.
The project uses the `ReActChain` class as a real notepad (scratchpad), in which the agent notes each step. Every time the agent performs an action or receives a result from a tool, the `update_memory` method saves the reasoning and the corresponding action in the shared memory.
Therefore, memory updating takes place in three steps: reception, where the agent obtains the information saved in the `last_step` variable; saving, where the `update_memory` method inserts the new action into `SharedMemory`, making it available to other agents; logging, where, using the `write_logs` method, the entire memory state is saved in a JSON file, which is useful for debugging.

Chapter 5

Results

This chapter presents the experimental evaluation of the proposed multi-agent architecture on the CVEfixes dataset, covering three vulnerability categories: CWE-89 (SQL Injection), CWE-79 (Cross-Site Scripting), and CWE-78 (OS Command Injection). The evaluation is built upon the three research questions outlined in Section 1.2: the ability of the AnalyzerAgent to autonomously validate CodeQL results and identify false negatives (RQ1), the capability of the SuggestorAgent to produce query improvement proposals (RQ2), and the ability of the CreatorAgent to generate compilable CodeQL queries following the SuggestorAgent's guidelines (RQ3). For each component, qualitative and quantitative results are reported. An additional evaluation method is introduced for RQ2 and RQ3 via an *LLM-as-a-Judge* protocol, in which an independent GPT-4o instance scores the SuggestorAgent's report and the CreatorAgent's generated queries.

5.1 AnalyzerAgent Results

The AnalyzerAgent was evaluated on a dataset of 26 real-world Python files from the CVEfixes dataset, analyzing three vulnerability categories: CWE-89 (SQL Injection, 10 files), CWE-79 (Cross-Site Scripting, 9 files), and CWE-78 (OS Command Injection, 7 files). Each file has one ground truth CWE label, while both the agent and CodeQL may report (and actually has) multiple CWEs per file. Therefore, multiple types of evaluation were used: a prediction is considered a True Positive (TP) if the ground truth CWE appears among the predicted CWEs for that file, a False Negative (FN) if it does not, and a False Positive (FP) for every predicted CWE that is not contained in the ground truth. Under this scheme, an agent that correctly identifies a vulnerability but also reports several additional CWEs for the same file accumulates FPs without losing the TP, which decreases precision while leaving recall unaffected. Table 5.1 contains the aggregate confusion

matrix and relative scores.

Table 5.1: Overall detection metrics: Analyzer Agent vs. CodeQL baseline (multi-label evaluation at the CWE level, 26 files).

System	TP	FP	FN	Precision	Recall	F1
Analyzer Agent	20	64	6	0.238	0.769	0.364
CodeQL Baseline	3	31	23	0.088	0.115	0.100

As illustrated, the Analyzer achieves a recall of 0.769, detecting the correct vulnerability of 20/26 files, compared to a recall of only 0.115 for the CodeQL baseline (3/26). Precision is low for both systems, 0.238 for the agent and 0.088 for CodeQL. For the agent's case, the low precision is due to the tendency to over-report, detecting additional CWEs such as CWE-20 (Improper Input Validation) or CWE-74 (Injection) alongside the correct ones. These secondary CWEs are not totally incorrect, but they do not match the ground truth and therefore count as FPs. For CodeQL, the low precision derives from a different failure mode. In fact, the baseline detects vulnerabilities on only 3 files, and 31 of the defects identified are additional (FP), indicating that the existing query suites are not really calibrated for this dataset.
Because the aggregate F1 is sensitive to the large number of FPs for additional detection of CWEs, a more informative measure is the *macro-average F1*, computed independently for each CWE category, reported in Table 5.2. Under this metric, the agent reaches 0.774, compared to 0.209 for CodeQL, an improvement of 3.7×.

Table 5.2: Per-CWE detection metrics: Analyzer Agent vs. CodeQL baseline.

CWE	System	TP	FP	FN	Precision	Recall	F1
CWE-78	Agent	7	1	0	0.875	1.000	0.933
	CodeQL	2	0	5	1.000	0.286	0.444
CWE-89	Agent	9	3	1	0.750	0.900	0.818
	CodeQL	1	0	9	1.000	0.100	0.182
CWE-79	Agent	4	1	5	0.800	0.444	0.571
	CodeQL	0	2	9	0.000	0.000	0.000
Macro-average F1		Agent: **0.774**			CodeQL: **0.209**		

As reported in the table above, the agent reports almost perfect recall for CWE-78, correctly identifying all 7 files containing OS Command Injection vulnerabilities. The CodeQL baseline detects only 2/7 files (recall 0.286), totally missing the remaining 5. The strong performance of the agent can be attributed to the directness of the vulnerability patterns, which produce a clear semantic signal that the LLM can easily recognize.
Regarding CWE-89, the agent achieves the most balanced performance, with an F1 of 0.818. The agent correctly detects 9/10 files. The CodeQL baseline detects only one file, confirming that the existing CWE-89 query suite is not really appropriate for this kind of dataset, since it does not cover all the vulnerability patterns in it.
CWE-79 is the most challenging category, with a recall of 0.444 and an F1 of 0.571. The agent correctly identifies vulnerabilities in 4/9 files. In several of the missed files, the agent detected security issues but classified them with different CWE identifiers, suggesting that maybe the vulnerability code patterns were ambiguous and so, as a consequence, supports multiple interpretations. This may indicate that Cross-Site Scripting (CWE-79) vulnerabilities are harder to identify with only static code inspection. The CodeQL baseline achieves zero recall for this category, failing to detect 9/9 files.

5.2 SuggestorAgent's results analysis

Once the AnalyzerAgent consolidated its findings and the overall metrics are computed, the resulting gap analysis is passed to the SuggestorAgent. Its input consists of 16 false negatives: 7 files containing CWE-78, 5 containing CWE-89, and 4 containing CWE-79. The SuggestorAgent's goal consists of translating these detection gaps into actionable technical recommendations that the CreatorAgent can use to generate new CodeQL queries.
The agent completed the task in 8 reasoning steps, successfully producing one structured `QueryProposal` for each of the three CWEs without skipping any sections specified in the prompt. For each CWE, the agent first invoked the `WebSearchTool` to gather additional information on CodeQL modeling patterns and modern Python libraries, and then called `GenerateProposalTool` to produce the structured proposal. However, in some other project runs, the agent skipped the `WebSearchTool` call, apparently overestimating its internal knowledge of CodeQL APIs. This behavior reflects a known limitation of LLM-based agents and may lead to hallucinations, producing incorrect predicates rather than a refusal to answer. This tendency can be mitigated by adding stricter rules in the prompt regarding mandatory external validation before each `GenerateProposalTool` call.
The final output of the SuggestorAgent is a structured Markdown report containing one proposal per CWE. Each proposal follows the CodeQL vulnerability modeling

paradigm:

1. Existing coverage: Description of what the current standard query already covers.
2. Gap identified: The specific pattern of API that the existing query fails to detect.
3. Missing sources: Possible entry points.
4. Missing sinks: Dangerous function calls not treated as sinks in the standard query.
5. Missing sanitizers: Safe patterns or functions that should block taint propagation.
6. Proposed addition: QL predicates that the CreatorAgent should implement.
7. Estimated impact: A qualitative assessment (*low*, *medium*, *high*) of the expected improvement of the new generated queries.

This structure maps each of the AnalyzerAgent's false negative files into the specific extension points of a CodeQL taint-tracking query, providing the CreatorAgent with a clear and actionable plan.

For CWE-78, the SuggestorAgent reports that the detection gap concerns `subprocess` calls where `shell=True` is supplied indirectly (via `**kwargs` dictionary expansion or through a non-literal variable) rather than as an explicit keyword argument at the call site - sentences used by the SuggestorAgent itself at the beginning of the section related to CWE-78. No new sources are proposed. The identified missing sinks and sanitizers are illustrated in the excerpt below.

Excerpt from SuggestorAgent proposal for CWE-78

Missing sinks

- `subprocess.run(...)` when `shell=True` is supplied via `**kwargs`.
- `subprocess.Popen(...)` when `shell=True` is supplied indirectly.
- `subprocess.call(...)`
- `subprocess.check_output(...)`
- `subprocess.check_call(...)`

Missing sanitizers

- Safe execution using argument lists (`args=<list/tuple>`) with `shell=False`, or the absence of `shell=True`.

- Use of `shlex.quote(s: str)` applied to every untrusted fragment before concatenation into a shell command string.

The estimated impact of this extension is *medium*.

Regarding CWE-89, the SuggestorAgent quotes at the beginning of the section that the identified gap concerns SQLAlchemy's textual SQL execution path, which is not covered by the standard DB-API sink model.
The missing sources and sanitizers are illustrated in the following excerpt.

Excerpt from SuggestorAgent proposal for CWE-89

Missing sources

- FastAPI / Starlette: `starlette.requests.Request.query_params.get()` result
- FastAPI / Starlette: `starlette.requests.Request.path_params.get()` result
- Flask: `flask.Request.args.get(...)` result
- Flask: `flask.Request.form.get(...)` result

Missing sinks

- SQLAlchemy: `sqlalchemy.engine.Connection.execute(statement, ...)` when `statement` is a `str` or `TextClause` derived from user-controlled data.
- SQLAlchemy ORM: `sqlalchemy.orm.Session.execute(statement, ...)` when `statement` is a `str` or `TextClause` derived from user-controlled data.
- SQLAlchemy: `sqlalchemy.sql.expression.text(sqltext)` when `sqltext` is user-controlled and the resulting `TextClause` flows to `.execute(...)`.

Missing sanitizers

- SQLAlchemy parameter binding using `sqlalchemy.text("... WHERE x=:x")` with bound parameters passed separately to `execute(..., {"x": value})`.
- `TextClause.bindparams(...)` used to bind parameters instead of formatting user input directly into SQL strings.

The estimated impact of this extension is *medium*.

For the CWE-79, the SuggestorAgent reports that the gap lies in the absence of Jinja2 rendering methods as XSS sinks within the reflected-XSS taint-model.

Specifically, the following methods are not currently modeled as sinks:

- `Template.render`
- `Environment.get_template().render()`
- `Template.generate`
- `Template.stream`

The missing sinks, sources, and sanitizers reported by the agent are shown in the excerpt below.

Excerpt from SuggestorAgent proposal for CWE-79

Missing sources

- Flask/Werkzeug: `flask.request.args.get(...)` (call to `ImmutableMultiDict.get`)
- Flask/Werkzeug: `flask.request.form.get(...)` (call to `ImmutableMultiDict.get`)
- Flask/Werkzeug: `flask.request.values.get(...)` (call to `CombinedMultiDict.get`)
- Flask/Werkzeug: `flask.request.cookies.get(...)` (call to `ImmutableMultiDict.get`)
- Django: `django.http.HttpRequest.GET.get(...)` (call to `QueryDict.get`)
- Django: `django.http.HttpRequest.POST.get(...)` (call to `QueryDict.get`)

Missing sinks

- Jinja2: `jinja2.Template.render(*args, **kwargs)` returning HTML/XML content used in HTTP responses.
- Jinja2: `jinja2.Environment.get_template(name).render(*args, **kwargs)` (via the returned `Template` object).
- Jinja2: `jinja2.Template.generate(*args, **kwargs)` or `jinja2.Template.stream(*args, **kwargs)` when streamed output is used in HTTP responses.

Missing sanitizers

- MarkupSafe: `markupsafe.escape(s: str) -> Markup` for HTML encoding.
- MarkupSafe: `markupsafe.Markup.escape(s: str) -> Markup`
- Jinja2: `jinja2.escape(s)` when used to encode untrusted data before rendering.

- MarkupSafe: `markupsafe.Markup(s)` only when the input is already trusted or pre-escaped.

Appendix A contains the full report.

5.2.1 LLM-as-a-Judge Evaluation

Evaluating technical correctness of CodeQL's proposals requires domain expertise that goes beyond the scope of this thesis. To obtain an independent quality signal, an automated *LLM-as-a-Judge* protocol was applied. Indeed, a separate GPT-4o instance evaluated each of the three proposals across six criteria derived from the CodeQL modeling paradigm:

- Gap identification.
- Source specificity.
- Sink specificity.
- Sanitizers correctness.
- Actionability.
- Overall quality.

Each dimension was scored on a scale from 1 to 5. The judge had access only to the SuggestorAgent's output and the corresponding CodeQL baseline query. The results are reported in Table 5.3.

Table 5.3: LLM-as-a-Judge scores for the SuggestorAgent proposals (GPT-4o evaluator, 1–5 scale).

CWE	Gap Id.	Source Spec.	Sink Spec.	Sanit. Corr.	Action-ability	Overall Quality	Avg
CWE-78	5	3	5	4	5	5	4.50
CWE-89	5	5	5	5	5	5	5.00
CWE-79	5	5	5	4	5	5	4.83
Mean	**5.0**	**4.3**	**5.0**	**4.3**	**5.0**	**5.0**	**4.78**

All three proposals receive the maximum score for *gap identification*, *actionability*, and *overall quality*, resulting in an average score of **4.78/5**.

The only dimension below the maximum score is *source specificity* for CWE-78 (3/5). The evaluator notes that the proposal introduces no new sources and only relies on existing models.
For CWE-89, the evaluator assigns a perfect score of 5/5, describing the proposals as precise, complete, technically sound, and immediately implementable.
The CWE-79 proposal receives a minor score for sanitizer correctness (4/5), but still close to the maximum.

5.3 CreatorAgent Results

The CreatorAgent receives as input the SuggestorAgent's Markdown report and the existing query files for the three target CWEs, loaded from the official `codeql/python-queries` package. For each CWE, the agent had to generate an extended `.ql` file by replacing the source/sink/sanitizers classes of the standard query with new and more appropriate ones. The generated queries were validated via the `WriteQueryTool`, which invokes the CodeQL compiler and returns any syntax or type errors. The agent is allowed a maximum of three compilation attempts to auto-correct the queries before they are marked as failed and the agent moves to the next CWE.

None of the three generated queries compiled successfully without human revision. The agent processed all three CWEs in 10 ReAct steps and terminated by invoking `FinishToolCreator`. Table 5.4 summarizes the outcomes of each CWE.

Table 5.4: CreatorAgent query generation outcomes.

CWE	Compiled	Attempts	Compiler error
CWE-78	No	3	`query module does not have any queries in scope`
CWE-79	No	3	`could not resolve type ReflectedXssFlow::Sink`
CWE-89	No	3	`could not resolve module API`

Regarding CWE-78, the agent, during all 3 attempts, produced only a comment block rather than executable QL code. Specifically, the file written to disk on each attempt consisted of a multi-line comment asking for the baseline query content, without any `import` statements, class definitions, or `select` clause. The compiler rejected all three submissions with the error `query module does not have any`

`queries in scope`, which indicated the absence of a valid `select` statement. The final saved file contains only comment lines.

The results for CWE-79 were more promising. Even though the first two attempts were rejected with the same `no queries at scope` error as CWE-78, the last trial generated a file that includes `import` statements, several class definitions, and a `select` clause - also demonstrating the effectiveness of the auto-recover cycle. This attempt was rejected with two compiler errors: `could not resolve module API` and `could not resolve type ReflectedXssFlow::Sink`.

Regarding CWE-89, all three attempts produced files that include `import` statements, multiple class definitions, and a `select` clause, but all were rejected by the compiler with the error `could not resolve module API`. The agent's reasoning traces show that it revised the class bodies between attempts but did not alter the import block.

To evaluate the quality of the CreatorAgent's output, it is necessary to compare an official standard CodeQL query with a query generated by the agent. CWE-89 examples were selected because the agent produced the most complete result among the three targeted categories. The standard query, shown in the following excerpt, presents a minimal design.

Official CodeQL query for CWE-89 (SQLInjection.ql)

```
/**
 * @name SQL query built from user-controlled sources
 * @kind path-problem
 * @problem.severity error
 * @security-severity 8.8
 * @precision high
 * @id py/sql-injection
 * @tags security
 *       external/cwe/cwe-089
 */

import python
import semmle.python.security.dataflow.SqlInjectionQuery
import SqlInjectionFlow::PathGraph

from SqlInjectionFlow::PathNode source,
     SqlInjectionFlow::PathNode sink
where SqlInjectionFlow::flowPath(source, sink)
select sink.getNode(), source, sink,
  "This SQL query depends on a $@.", source.getNode(),
```

```
  "user-provided value"
```

The generated query is shown in the excerpt below.

Generated CodeQL query for CWE-89

```
/**
 * @name SQL injection (extended: SQLAlchemy execute/text +
 * framework `.get` sources)
 * @description Finds SQL injection vulnerabilities.
 * @kind path-problem
 * @problem.severity warning
 * @id py/sql-injection-extended
 * @tags security
 *       external/cwe/cwe-089
 */

import python
import semmle.python.security.dataflow.SqlInjectionQuery
import SqlInjectionFlow::PathGraph

private class StarletteRequestGetSource extends SqlInjectionFlow::Source {
  StarletteRequestGetSource() {
    exists(API::Call c |
      c = API::moduleImport("starlette.requests")
            .getMember("Request")
            .getMember("query_params")
            .getMember("get")
            .getACall()
      and this.asExpr() = c
    )
  }
}

private class FlaskRequestGetSource extends SqlInjectionFlow::Source {
  FlaskRequestGetSource() {
    exists(API::Call c |
      c = API::moduleImport("flask")
            .getMember("request")
            .getMember("args")
            .getMember("get")
            .getACall()
      and this.asExpr() = c
    )
  }
}
```

```
private class SqlAlchemyExecuteArg0Sink extends SqlInjectionFlow::Sink {
  SqlAlchemyExecuteArg0Sink() {
    exists(API::Call c |
      c = API::moduleImport("sqlalchemy.engine")
            .getMember("Connection")
            .getMember("execute")
            .getACall()
      and this.asExpr() = c.getArgument(0)
    )
  }
}

from SqlInjectionFlow::PathNode source,
     SqlInjectionFlow::PathNode sink
where SqlInjectionFlow::flowPath(source, sink)
select sink.getNode(), source, sink,
  "Possible SQL injection vulnerability."
```

A comparison of the two queries reveals several structural differences. First, in the standard query the metadata block (lines 1–10) is placed before all the `import` statements, while in the generated query the `import` statements appear before and the metadata block after. Second, the standard query's metadata include the `@precision` tag, which is absent in the generated one. Furthermore, the generated query contains three sources classes, four sinks classes, and two sanitizer classes, all targeting APIs identified in the SuggestorAgent's CWE-89 proposal. This also outlines that the communication between agents worked in a proper way.

5.3.1 LLM-as-a-Judge Evaluation

Similarly to the SuggestorAgent's evaluation, the technical correctness of the generated QL code requires CodeQL expertise beyond the scope of this thesis. Therefore, the same GPT-4o judge evaluated the generated queries on six criteria: report alignment, CodeQL *correctness*, *coverage*, *false-positive*, and *overall quality* on a score up to 5. The results are reported in the Table 5.5.

Table 5.5: LLM-as-a-Judge scores for the CreatorAgent's queries (GPT-4o evaluator, 1–5 scale).

CWE	Report Align.	CodeQL Corr.	Coverage	FP Mitig.	Metadata	Overall Quality	Avg
CWE-78	1	1	1	1	1	1	1.00
CWE-79	4	3	4	3	4	3	3.50
CWE-89	5	4	5	4	5	4	4.50
Mean	**3.3**	**2.7**	**3.3**	**2.7**	**3.3**	**2.7**	**3.00**

For CWE-78, the evaluator assigns 1/5 on all dimensions, noting that the submitted files contain no CodeQL logic and therefore cannot be evaluated on any criterion. For CWE-79 the evaluator reports 4/5 for report alignment and coverage, since the file follows the proposals of the SuggestorAgent, but lowers the CodeQL correctness score to 3/5 due to issues with the class hierarchy and the absence of the `@precision` metadata. For CWE-89, the evaluator assigns the highest scores among the three queries.

To summarize, the average score for the SuggestorAgent and CreatorAgent evaluation is **3.89/5**, as illustrated in Table 5.6.

Table 5.6: Pipeline-level LLM-as-a-Judge summary.

Component	Average score (1–5)
Suggestor proposals	4.78
Creator queries	3.00
Overall pipeline	**3.89**

Overall, the experimental results demonstrate that the proposed multi-agent architecture significantly improves vulnerability detection coverage compared to

the CodeQL baseline, while also identifying concrete opportunities for extending the underlying query models.

5.4 Comparison with GPT-4o-mini

To assess how sensitive the pipeline is to the choice of the language model, the entire three-agent workflow was re-run replacing the primary model with **GPT-4o-mini**. All other experimental conditions, including the dataset, the prompt template, and the tool configurations, were unchanged. The following subsections report the outcomes for each agent and discuss where and why the smaller model diverges from the primary results.

5.4.1 AnalyzerAgent

The GPT-4o-mini AnalyzerAgent was evaluated on the same 26-file dataset under the same multi-label CWE evaluation protocol used in Section 5. Table 5.7 reports the aggregate confusion-matrix metric for the GPT-4o-mini agent alongside those of the primary model and the CodeQL baseline.

Table 5.7: Overall detection metrics: GPT-4o-mini agent vs. GPT-5.2 agent vs. CodeQL baseline (multi-label evaluation, 26 files).

System	TP	FP	FN	Precision	Recall	F1
GPT-4o-mini Agent	21	62	5	0.253	0.808	0.385
GPT-5.2 Agent	20	64	6	0.238	0.769	0.364
CodeQL Baseline	3	31	23	0.088	0.115	0.100

At the aggregate level, the two agents are practically equivalent: GPT-4o-mini achieves recall of 0.808 vs 0.769 of GPT-5.2, and both produce a comparably high false-positive count (≈62–64). The per-CWE breakdown in Table 5.8 reveals more nuanced differences.

Table 5.8: Per-CWE detection metrics: GPT-4o-mini agent vs. GPT-5.2 agent vs. CodeQL baseline.

CWE	System	TP	FP	FN	Precision	Recall	F1
CWE-78	GPT-4o-mini	7	0	0	1.000	1.000	1.000
	GPT-5.2	7	1	0	0.875	1.000	0.933
	CodeQL	2	0	5	1.000	0.286	0.444
CWE-89	GPT-4o-mini	9	3	1	0.750	0.900	0.818
	GPT-5.2	9	3	1	0.750	0.900	0.818
	CodeQL	1	0	9	1.000	0.100	0.182
CWE-79	GPT-4o-mini	5	0	4	1.000	0.556	0.714
	GPT-5.2	4	1	5	0.800	0.444	0.571
	CodeQL	0	2	9	0.000	0.000	0.000
Macro-average F1		GPT-4o-mini: **0.844**			GPT-5.2: **0.774**		

GPT-4o-mini achieves a perfect F1-score of 1.000 on CWE-78 and improves on CWE-79 (F1 0.714 vs 0.571), while matching the other model on CWE-89 (same F1 0.818). The overall macro-average F1 of GPT-4o-mini (0.844) is slightly higher than that of the primary model (0.774). These results indicate that for the AnalyzerAgent role, the smaller model is a competitive substitute, since its contextual code reasoning is sufficient to detect the predominant vulnerability patterns in this dataset without a meaningful performance penalty. Perhaps for a dataset with more complex vulnerability patterns GPT-5.2 would outperform.

5.4.2 SuggestorAgent

The GPT-4o-mini SuggestorAgent successfully generated one structured proposal for each of the three CWE categories, without skipping any. However, two behavioral problems occurred. First, at reasoning step 4, the agent invoked `FinishToolSuggestor` prematurely, before CWE-89 and CWE-79 had been processed; the tool's guard clause blocked the completion and the execution continued. Second, when processing CWE-79, repeated `WebSearchTool` calls returned `HTTP 429 rate-limiting errors`, and the agent proceeded to generate the proposal entirely from its internal knowledge, without external validation, highlighting the same risk of hallucinated predicates discussed in Section 5.

5.4.3 LLM-as-a-Judge Evaluation

The same GPT-4o judge applied to GPT-5.2 proposals was used to evaluate the GPT-4o-mini SuggestorAgent output on the same six criteria. The results are reported in Table 5.9.

Table 5.9: LLM-as-a-Judge scores for the GPT-4o-mini SuggestorAgent's proposals (GPT-4o evaluator, 1–5 scale).

CWE	Gap Id.	Source Spec.	Sink Spec.	Sanit. Corr.	Action-ability	Overall Quality	Avg
CWE-78	4	3	4	3	4	4	3.67
CWE-89	4	4	4	3	4	4	3.83
CWE-79	4	4	4	3	4	4	3.83
Mean	**4.0**	**3.7**	**4.0**	**3.0**	**4.0**	**4.0**	**3.78**

The GPT-4o-mini SuggestorAgent achieves an average score of **3.78/5**, compared to **4.78/5** for GPT-5.2 - a gap of 1.0 point. The difference is consistent across all three CWEs and corresponds to two systematic weaknesses identified by the evaluator. First, sanitizer correctness scores 3/5 across every CWE. Second, source specificity drops to 3/5 for CWE-78 because no exact API signatures or argument position are specified.

5.4.4 CreatorAgent

The most significant gap of performance between the two models appears in the CreatorAgent. While the primary model produced substantive `.ql` file drafts for CWE-89 and CWE-79 - each containing proper `import` statements, multiple class definitions, and a valid `select` clause - GPT-4o-mini failed on all three CWEs across all three compilation attempts. Table 5.10 summarizes the results.

Table 5.10: CreatorAgent outcomes: GPT-4o-mini vs. GPT-5.2.

CWE	Model	Compiled	Attempts	Root cause
CWE-78	GPT-4o-mini	No	3	Hallucinated types: `CommandInjection`, `Subprocess`
	GPT-5.2	No	3	No `select` statement (placeholder only)
CWE-79	GPT-4o-mini	No	3	`import javascript`; hallucinated `XSS`, `Clipboard` types
	GPT-5.2	No	3	`could not resolve type ReflectedXssFlow::Sink`
CWE-89	GPT-4o-mini	No	3	`from DataFlow import *`; hallucinated `SqlInjection`, `UserInput`, `SqlQuery`
	GPT-5.2	No	3	`could not resolve module API`

The failure modes are qualitatively different. The GPT-5.2 agent's queries were structurally coherent CodeQL Python files that imported the correct standard libraries, defined class hierarchies extending the appropriate base types, and included valid `select` clauses. Furthermore, their compilation errors were narrowly scoped and fixable with targeted corrections. In contrast, GPT-4o-mini agent demonstrated a fundamental misunderstanding of the CodeQL Python ecosystem in all attempts. For CWE-79 the agent wrote `import javascript`, treating the query as targeting a JavaScript codebase. For CWE-89 it used `from DataFlow import *`, which is not valid CodeQL syntax. All three files reference non-existent class names - `SQLInjection`, `UserInput`, `CommandInjection`, `XSS` - with no counterpart in the standard library. Furthermore, unlike the primary model which modified class bodies between attempts, GPT-4o-mini submitted identical code across all three retries for each CWE, indicating that the auto-correction cycle entirely failed.
The following excerpt shows the GPT-4o-mini CWE-89 query. Its brevity and fictional vocabulary contrast with the 150-line and a structured valid draft produced by GPT-5.2.

GPT-4o-mini generated query for CWE-89 (all three attempts identical)

```
import python

from DataFlow import *
```

```
class SQLInjection extends DataFlow::PathNode {
  SQLInjection() {
    this.getSource() instanceof UserInput;
  }
}

class DjangoORM extends DataFlow::PathNode {
  DjangoORM() {
    this.getSink() instanceof MethodCall &&
    (this.getMethod().getName() = "filter" ||
     this.getMethod().getName() = "raw");
  }
}

from SQLInjection sqlInjection, DjangoORM orm
select sqlInjection,
  "Potential SQL Injection vulnerability via Django ORM";
```

5.4.5 LLM-as-a-Judge Evaluation

The GPT-4o judge evaluated the three generated queries on the same six criteria used in Section 5.3.1. The results are reported in Table 5.11.

Table 5.11: LLM-as-a-Judge scores for the GPT-4o-mini CreatorAgent's queries (GPT-4o evaluator, 1–5 scale).

CWE	Report Align.	CodeQL Corr.	Cover-age	FP Mitig.	Meta-data	Overall Quality	Avg
CWE-78	1	1	1	1	1	1	1.00
CWE-79	1	1	1	1	1	1	1.00
CWE-89	1	2	1	1	1	1	1.17
Mean	**1.0**	**1.3**	**1.0**	**1.0**	**1.0**	**1.0**	**1.06**

The GPT-4o-mini CreatorAgent receives an average score of **1.06/5**, compared to **3.00/5** for the primary model. The evaluator assigns 1/5 on every dimension for CWE-78 and CWE-79, noting that the files implement none of the proposed sources, sinks, or sanitizers from the SuggestorAgent's proposals and contain fundamental language-level errors. The only score above 1 is CodeQL correctness for CWE-89, which receives 2/5 because the query at least imports the correct language module (`import python`) and adopts a class-based structure, even though the class names are entirely fictional.

5.4.6 Pipeline-level Summary

Table 5.12 provides a direct side-by-side comparison of the AnalyzerAgent detection metrics, the LLM-as-a-Judge averages, and the overall pipeline scores for both model configurations.

Table 5.12: Pipeline-level summary: GPT-5.2 vs. GPT-4o-mini.

Component	**GPT-5.2**	**GPT-4o-mini**
Analyzer macro-F1	0.774	0.844
Suggestor proposals (avg)	4.78	3.78
Creator queries (avg)	3.00	1.06
Overall pipeline score	**3.89**	**2.42**

The comparison reveals clear limits of model capabilities compared to GPT-5.2. The AnalyzerAgent task falls between the competence of GPT-4o-mini, which even achieves a marginally higher macro-F1 (0.844 vs 0.774). The SuggestorAgent task, which requires grounding proposals in a precise vocabulary, is more demanding for the smaller model. Indeed, the outputs are structurally sound, but contain factual errors in sanitizer semantics and miss framework-specific API patterns, reducing the judge score by a full point (3.78 vs. 4.78). The CreatorAgent task, which demands the synthesis of compilable, multi-class QL code against a specific library, is clearly beyond the model's reliable capability. Indeed, GPT-4o-mini confuses languages, invents false APIs, and fails to improve across retries, resulting in a very low score of 1.06 vs 3.00 from the judge and an overall pipeline score of 2.42 vs. 3.89.
These findings suggest that, while a smaller and less expensive model may be a proper substitute for the detection and analysis phase of the pipeline, the query-generation stage benefits substantially from a larger model with stronger grounding in domain-specific APIs and more reliable code-synthesis capabilities.

5.5 Results Summary and Limitations

Table 5.13 summarizes the results produced by the CodeQL tool and each agent, while highlighting the significant limitations and potential solutions.

Table 5.13: Synthesis of results by agent.

Agent	Result	Main limitations	Possible solutions
CodeQL SAST tool	Recall 0.115; macro-F1 0.209; 3 TP out of 26 files	Very limited coverage of the dataset's vulnerability patterns; zero recall on CWE-79	Extend and tune the query suite for the target vulnerability categories
Analyzer agent	Recall 0.769; macro-F1 0.774; 20 TP out of 26 files	High FP rate (64) due to over-reporting of additional CWEs; lowest recall on CWE-79 (0.444)	Stronger prompt constraints to limit secondary CWE predictions; more diverse dataset
Suggestor agent	3/3 structured proposals generated; missing sources, sinks, and sanitizers identified for all CWEs; LLM-as-a-Judge avg 4.78/5	Occasional omission of `WebSearchTool` call; risk of overconfident or hallucinated predicates	Mandatory external-validation step before each proposal; stricter prompt constraints
Creator agent	3/3 CWEs attempted; query drafts produced for CWE-79 and CWE-89; LLM-as-a-Judge avg 3.00/5	CWE-78 produced only a placeholder stub; CWE-79 and CWE-89 failed compilation due to unresolved references	Provide baseline query content explicitly in context; add import-validation step; increase iteration steps

By observing the table, it is possible to note that each agent demonstrates both achievements and limitations, indicating that they are capable of functioning correctly in some cases but still require further refinement.

RQ1 - Autonomous Validation *Can an LLM agent autonomously validate CodeQL results through contextual code reasoning, improving precision, by filtering false positives and recall, by identifying false negatives, compared to CodeQL analysis?* The experimental results provide a positive answer. The AnalyzerAgent achieves a recall of 0.769 and a macro-average F1 of 0.774, compared to 0.115 and 0.209 for the CodeQL tool. The agent correctly identifies the vulnerabilities in 20/26 files and covers all three CWE categories, while the CodeQL achieves zero recall on CWE-79 and detects only 3 files overall. These results confirm that contextual reasoning via an LLM agent substantially improves and extends detection coverage compared to static rules. The main limitation is the high false-positive rate (64 FPs), caused by the tendency of the agent to report additional CWEs. A solution to this may be tighter prompt constraints.

RQ2 - Systematic Gap Identification. *Can a multi-agent system automatically identify which specific sources, sinks and sanitizers are missing from existing standard CodeQL queries?* The experimental results provide a positive answer. The SuggestorAgent produced a structured proposal for each of the three CWEs. For each of them, the proposals enumerate specific missing sources, sinks, and sanitizers, together with QL predicates and an estimate impact. The LLM-as-a-judge evaluation assigns an average of 4.78/5 across all proposals, with perfect scores on *gap identification* and *actionability* for all CWEs. The main limitation is the occasional omission of the `WebSearchTool` step, which may introduce overconfidence in hallucinated predicates when the agent relies only on its internal knowledge.

RQ3 - Query Synthesis. *Can an LLM agent synthesize CodeQL query draft targeting identified gaps, reducing manual effort even when generated queries require human refinement before deployment?* The results provide a partial answer. The CreatorAgent tried to generate all three CWEs and produced substantive `.ql` files for CWE-79 and CWE-89, containing import statements, multiple additional sources, sinks, and sanitizers, and a valid `select` clause. None of the three queries compiled without errors, so none is immediately deployable. However, the RQ explicitly accounts for the need for human intervention and refinement. Following this assumption, the generated CWE-89 represents a structured starting point that a developer can terminate with targeted simply corrections. The CWE-78 case is a more substantial failure, as the agent produced only a placeholder - an event that can be avoided by enforcing the strictness of the prompt. These finding indicate that the pipeline can reduce manual effort for CWE-89 and CWE-79 by providing a well-structured draft, but further improvements.

5.5.1 Limitations of the LLM-as-a-Judge Protocol

Although the LLM-as-a-Judge protocol evaluation provides a practical way to obtain structured feedback without requiring manual CodeQL expertise, limitations should be considered when interpreting the scores.
The judge evaluates queries by inspecting their source code rather than running them on a real database. While this means that some subtle runtime errors may not be detected, it is also consistent with how a human reviewer would perform an initial code review, making the evaluation methodology reasonable for the purpose of this work. A potential concern is self-assessment bias, as the judge and the agents share the same pre-training distribution. However, this is a well-known compromise and accepted in LLM-based assessment, and the structured scoring criteria adopted here mitigate arbitrary judgments by anchoring the evaluation to specific and well-defined dimensions. Finally, since it is a generic model, the judge may not have deep experience in CodeQL-specific constructs. However, the criteria selected for evaluation, such as report alignment, coverage and false positive mitigation, are largely independent of language and fall largely within the jurisdiction of the judge, in support of the reliability of scores as a comparative signal of quality between models and the CWE categories.

Chapter 6

Conclusion and Future Works

6.1 Summary of Contributions

This thesis investigated how LLM-based agents can be integrated with a static analysis tools (CodeQL) to improve software vulnerability detection. In particular, the work focused on extending the capabilities of the CodeQL static analysis framework through a multi-agent architecture designed to analyze, diagnose, and extend existing vulnerability queries.

The proposed system consists of three specialized agents that simulate different stages of the workflow typically performed by a security analyst. The AnalyzerAgent validates CodeQL detections through contextual reasoning on source code and identifies potential false negatives. The SuggestorAgent analyzes the gaps and generates structured proposals describing missing sources, sinks, and sanitizers in the existing queries. Finally, the CreatorAgent attempts to synthesize new and improved CodeQL queries following the SuggestorAgent's proposals.

The experimental evaluation was conducted on a subset of the CVEfixes dataset containing Python vulnerabilities belonging to three categories: CWE-89 (SQL Injection), CWE-79 (Cross-Site Scripting), and CWE-78 (OS Command Injection). The results show that the AnalyzerAgent significantly improves vulnerability detection coverage compared to the CodeQL tool, achieving a recall of 0.769 compared to 0.115 for the static analysis tool.

Furthermore, the SuggestorAgent successfully produced structured proposals for query extension for all vulnerabilities considered, with a quality score of 4.78/5

in the LLM-as-a-Judge evaluation. While the CreatorAgent was able to generate structured CodeQL query drafts, none of the generated queries compiled successfully without manual revision.

Overall, the results demonstrate that a multi-agent architecture can support several stages of the vulnerability analysis workflow, helping identify detection gaps, and providing structured starting points for extending static analysis queries.

6.2 Limitations and Threats to Validity

Despite the positive results, there are some limitations that need to be considered when interpreting the findings of this thesis.
First, the experimental evaluation was conducted on a relatively small subset of the CVEfixes dataset (26 files). Although these samples represent real-world vulnerabilities, the limited dataset size may restrict the generalization of the results. Larger and more varied datasets would provide a more reliable evaluation of the proposed architecture.
Second, the behavior of LLM-based agents introduces inherent uncertainty. The AnalyzerAgent occasionally over-reports vulnerabilities, resulting in a high number of false positives. Similarly, the SuggestorAgent may omit external validation steps or rely on incomplete internal knowledge when generating proposals. These behaviors reflect the known limitations of large language models in reasoning tasks.
A third limitation relates to the evaluation of the generated proposals and queries. The quality assessment relies partially on an *LLM-as-a-Judge* protocol, which introduces a degree of subjectivity into the evaluation process. Although the judge model was provided with the baseline queries and structured evaluation criteria, its judgments should be interpreted as an approximate quality signal rather than a definitive measure of correctness.
Finally, the CreatorAgent's inability to generate fully compilable CodeQL queries highlights the difficulty of production correct static analysis rules through automated generation. Therefore, the results suggest that LLM-based query synthesis is currently better suited to producing structured drafts rather than fully deployable ones.
These limitations indicate that the proposed system should be viewed as an assistive tool, supporting security analysts rather than a fully autonomous vulnerability detection solution.

6.3 Future Work

Several directions could extend and improve the approach presented in this thesis. First, the query synthesis capabilities of the CreatorAgent could significantly be improved. Future work may explore stronger prompt constraints, improve tool feedback loops, and the inclusion of the baseline CodeQL query content directly in the agent's context to reduce generation errors.
Second, the evaluation could be expanded to larger datasets with more CWE categories and programming languages in order to provide a more comprehensive assessment of the system's effectiveness.
Furthermore, another interesting direction could be the integration of the multi-agent architecture into continuous security analysis pipelines. For example, the idea would be that the AnalyzerAgent is used to automatically review the results of static analysis tool, while the SuggestorAgent could assist security engineers in improving detection rules as new vulnerability patterns emerge.
Further research could also explore hybrid approaches combining LLM-based reasoning with analysis techniques for programs, such as dynamic analysis, in order to improve the precision of vulnerability detection.

Appendix A

SuggestorAgent Output Report

This appendix contains the full output produced by the SuggestorAgent at the end of its execution. The report is organized by CWE category and documents, for each vulnerability class, the existing CodeQL coverage, the identified detection gap, the missing sources, sinks, and sanitizers, the proposed QL additions, and the estimated impact of the proposed changes.

Summary

Processed CWEs	CWE-89, CWE-78, CWE-79
Skipped CWEs	None

Table A.1: SuggestorAgent execution summary.

CWE-78 — OS Command Injection

Existing Coverage

The existing queries rely on the standard `semmle.python.security.dataflow.CommandInjectionQuery` and `UnsafeShellCommandConstructionQuery` configurations, which already model common external sources (web-request and environment-derived user input), common command-execution sinks (`os.system`, `subprocess.*` variants, and other command-launching APIs), and known sanitizer patterns (e.g.,

passing arguments as a list rather than a shell string). They also include path rendering via `PathGraph` and report flows from modeled sources to modeled sinks.

Gap Identified

A frequent missed pattern for CWE-78 in Python is **command execution via `subprocess` APIs where `shell=True` is supplied indirectly** — for example through a `**kwargs` dict, a variable passed as `shell=flag`, or a wrapper function that forwards keyword arguments. Many CodeQL models catch direct `subprocess.run(user, shell=True)` style sinks, but miss cases where the `shell=True` condition is not a literal at the call site, or where the call is made through `Popen.__init__` wrappers. As a result, attacker-controlled strings reaching `subprocess.run(..., **opts)` (with `opts={"shell": True}`) or `Popen(cmd, **opts)` may not be recognized as shell-command execution sinks.

Missing Sources

No new sources are proposed. The existing external-input source modeling in `CommandInjectionQuery` and `UnsafeShellCommandConstructionQuery` is sufficient.

Missing Sinks

Sink	Condition
`subprocess.run(args, *, shell=True, ...)`	`shell=True` provided via `**kwargs` or non-literal boolean
`subprocess.Popen(args, *, shell=True, ...)`	`shell=True` provided via `**kwargs` or non-literal boolean
`subprocess.call(args, *, shell=True, ...)`	`shell=True` provided via `**kwargs` or non-literal boolean
`subprocess.check_output(args, *, shell=True, ...)`	`shell=True` provided via `**kwargs` or non-literal boolean
`subprocess.check_call(args, *, shell=True, ...)`	`shell=True` provided via `**kwargs` or non-literal boolean

Table A.2: Missing sinks for CWE-78.

Missing Sanitizers

- `subprocess.*(args=<list/tuple>, shell=False)` — explicit safe execution pattern when arguments are not a shell string.

- `shlex.quote(s: str) -> str` — applied to every untrusted fragment before concatenation into a shell string (partial sanitizer for shell-string contexts).

Proposed QL Addition

Listing A.1: Proposed CodeQL extension for CWE-78.

```
import python
import semmle.python.security.dataflow.CommandInjectionQuery
import CommandInjectionFlow::PathGraph

/**
 * Adds sinks for subprocess invocations where shell=True is supplied
 * indirectly (e.g., via **kwargs dict expansion or non-literal values).
 */
private class IndirectShellTrueSubprocessSink
    extends CommandInjectionFlow::Sink {
  IndirectShellTrueSubprocessSink() {
    this = any(Call c |
      isIndirectShellTrueSubprocessCall(c)).asSink()
  }

  private predicate isIndirectShellTrueSubprocessCall(Call c) {
    exists(AttributeAccess callee |
      callee = c.getCallee() and
      callee.getQualifier() instanceof Name and
      callee.getQualifier().(Name).getId() = "subprocess" and
      callee.getAttributeName() in
        ["run", "call", "check_call", "check_output", "Popen"]
    )
    and hasShellTrueViaKwargsOrNonLiteral(c)
  }

  /**
   * Matches cases where shell=True is not a simple literal at the call
   * site, including **opts where opts = {"shell": True}, or shell=flag.
   */
  private predicate hasShellTrueViaKwargsOrNonLiteral(Call c) {
    // Case 1: explicit keyword but value is not the literal False.
    exists(KeywordArgument ka |
      ka = c.getKeywordArgument("shell") and
      not ka.getValue() instanceof False
    )
    or
    // Case 2: dict expansion **kwargs present with no explicit shell=False.
    exists(StarStarArgument ssa | ssa = c.getStarStarArgument())
    and not exists(KeywordArgument ka2 |
      ka2 = c.getKeywordArgument("shell") and
      ka2.getValue() instanceof False
    )
  }
}

/**
 * Recognizes shlex.quote as a sanitizer for shell-string contexts.
 */
private class ShlexQuoteSanitizer
    extends CommandInjectionFlow::Sanitizer {
  override predicate isSanitizer(Node n) {
```

```
53      exists(Call c, AttributeAccess callee |
54        c = n.asExpr() and
55        callee = c.getCallee() and
56        callee.getQualifier() instanceof Name and
57        callee.getQualifier().(Name).getId() = "shlex" and
58        callee.getAttributeName() = "quote"
59      )
60    }
61  }
```

Estimated Impact

Medium. The proposed additions extend detection coverage to indirect `shell=True` patterns that are common in wrapper functions and configuration-driven subprocess invocations, reducing false negatives in codebases that do not use literal keyword arguments at subprocess call sites.

Evidence Files

- `data/src/lib/Bcfg2/Server/Plugins/Trigger.py`
- `data/Data/views.py`
- `data/web/reNgine/common_func.py`
- `data/IPython/utils/terminal.py`
- `data/endpoints/lollms_advanced.py`
- `data/autogpts/autogpt/autogpt/commands/execute_code.py`

CWE-89 — SQL Injection

Existing Coverage

The existing query delegates detection to `semmle.python.security.dataflow.SqlInjectionQuery` and reports paths via `SqlInjectionFlow::flowPath`. The standard library model covers common user-controlled sources (web framework request inputs, environment and CLI parameters), common SQL construction and execution sinks (DB-API `cursor.execute`/`executemany`, some ORM raw-execution APIs), and basic sanitization patterns where parameters are passed separately via parameterized queries rather than concatenated into SQL text.

Gap Identified

A frequent miss in Python SQL injection models is the **non-DB-API execution surface** — particularly SQLAlchemy's textual SQL execution path — where the SQL string is supplied to `Connection.execute(...)` or `Session.execute(...)` as a `str` or as a `TextClause` created by `sqlalchemy.text(...)`. If the model only recognizes classic `cursor.execute(query, params)` style sinks, it misses SQLAlchemy code that builds SQL dynamically using f-strings or string concatenation and executes it via SQLAlchemy's `execute`, which remains vulnerable unless parameters are properly bound.

Missing Sources

Framework	**Source**
FastAPI / Starlette	`starlette.requests.Request.query_params.get()`
FastAPI / Starlette	`starlette.requests.Request.path_params.get()`
Flask	`flask.Request.args.get()`
Flask	`flask.Request.form.get()`

Table A.3: Missing sources for CWE-89.

Missing Sinks

Sink	**Condition**
`sqlalchemy.engine.Connection.execute(statement, ...)`	`statement` is a `str` or `TextClause` derived from user-controlled data
`sqlalchemy.orm.Session.execute(statement, ...)`	`statement` is a `str` or `TextClause` derived from user-controlled data
`sqlalchemy.sql.expression.text(sqltext)`	`sqltext` is user-controlled and the resulting `TextClause` flows to `.execute(...)`

Table A.4: Missing sinks for CWE-89.

Missing Sanitizers

- SQLAlchemy parameter binding via `sqlalchemy.text("... WHERE x=:x")` with bound parameters passed separately to `execute(..., {"x": value})`.
- `TextClause.bindparams(...)` used to bind parameters rather than formatting values directly into SQL text.

Proposed QL Addition

Listing A.2: Proposed CodeQL extension for CWE-89.

```
import python
import semmle.python.security.dataflow.SqlInjectionQuery
import SqlInjectionFlow::PathGraph

/**
 * Adds SQLAlchemy textual execution sinks not covered by the default model.
 */
private class SqlAlchemyExecuteSink extends SqlInjectionFlow::SinkNode {
  SqlAlchemyExecuteSink() {
    exists(Call c |
      this.asNode() = c and
      c.getCallee() instanceof Attribute and
      c.getCallee().(Attribute).getAttr() = "execute" and
      // First positional argument: the SQL string or TextClause.
      this.getNode() = c.getArg(0)
    )
  }
}

/**
 * Adds sqlalchemy.text(<sql>) as a sink: if user input reaches text(...),
 * it becomes a TextClause that is subsequently executed.
 */
private class SqlAlchemyTextSink extends SqlInjectionFlow::SinkNode {
  SqlAlchemyTextSink() {
    exists(Call c |
      this.asNode() = c and
      c.getCallee() instanceof Name and
      c.getCallee().(Name).getId() = "text" and
      this.getNode() = c.getArg(0)
    )
  }
}
```

Estimated Impact

Medium. The proposed additions extend detection to SQLAlchemy-based execution paths, which are common in modern Python web applications that use ORM frameworks rather than raw DB-API cursors, covering a significant class of SQL injection patterns currently invisible to the standard query.

Evidence Files

- `data/django/contrib/postgres/aggregates/general.py`
- `data/mod_fun/__init__.py`
- `data/auth/controllers/group_controller.py`
- `data/auth/controllers/user_controller.py`

- `data/app.py`

- `data/redports-trac/redports/model.py`

- `data/flair.py`

CWE-79 — Cross-Site Scripting

Existing Coverage

The existing `py/reflective-xss` query relies on `semmle.python.security.dataflow.ReflectedXssQuery` and `ReflectedXssFlow::flowPath`, which model common user-controlled input sources (web framework request parameters) flowing to HTML response sinks with a set of known sanitizers and encoders. In addition, the separate `py/jinja2/autoescape-false` query flags creation of `jinja2.Environment(...)` or `jinja2.Template(...)` where `autoescape` is missing or explicitly `False`, but it does not connect that configuration to an actual data flow from user input to rendering.

Gap Identified

The main gap is **missing coverage for Jinja2 rendering sinks** — especially `Template.render(**kwargs)` and `Environment.get_template(..).render(..)` — in the reflected-XSS taint model. When untrusted request data is passed into Jinja2 rendering in projects that disable autoescaping globally or per-template, no flow is reported because the `render` call is not treated as an HTML output sink by `ReflectedXssQuery`'s default sink set.

Missing Sources

Framework	Source
Flask / Werkzeug	`flask.request.args.get(...)` (`ImmutableMultiDict.get`)
Flask / Werkzeug	`flask.request.form.get(...)` (`ImmutableMultiDict.get`)
Flask / Werkzeug	`flask.request.values.get(...)` (`CombinedMultiDict.get`)
Flask / Werkzeug	`flask.request.cookies.get(...)` (`ImmutableMultiDict.get`)
Django	`django.http.HttpRequest.GET.get(...)` (`QueryDict.get`)
Django	`django.http.HttpRequest.POST.get(...)` (`QueryDict.get`)

Table A.5: Missing sources for CWE-79.

Missing Sinks

Sink	Notes
`jinja2.Template.render(*args, **kwargs)`	Returns HTML/XML string used as response content
`jinja2.Environment.get_template(name).render(*args, **kwargs)`	Variant returned `Template` object
`jinja2.Template.generate(...)` / `.stream(...)`	Streamed output used in HTTP responses

Table A.6: Missing sinks for CWE-79.

Missing Sanitizers

- `markupsafe.escape(s: str) -> Markup` — encodes HTML special characters.
- `markupsafe.Markup.escape(s: str) -> Markup`.
- `jinja2.escape(s)` — in some versions re-exports `markupsafe.escape`; treat as a taint barrier when used to encode untrusted data before rendering.

- `markupsafe.Markup(s)` — only when input is already trusted and escaped; treat as a sanitizer barrier to suppress false positives where developers deliberately mark safe content.

Proposed QL Addition

Listing A.3: Proposed CodeQL extension for CWE-79.

```
import python
import semmle.python.security.dataflow.ReflectedXssQuery
import ReflectedXssFlow::PathGraph

/**
 * Identifies Jinja2 rendering calls as HTML output sinks.
 */
private predicate isJinja2RenderCall(CallNode call) {
  exists(API::Node tmpl |
    // jinja2.Template.render(...)
    tmpl = API::moduleImport("jinja2").getMember("Template") and
    call = tmpl.getMember("render").getACall()
  )
  or
  exists(API::Node env |
    // jinja2.Environment.get_template(...).render(...)
    env = API::moduleImport("jinja2").getMember("Environment") and
    exists(CallNode getT |
      getT = env.getMember("get_template").getACall() and
      call.getReceiver() = getT
    ) and
    call.getCalleeName() = "render"
  )
}

/**
 * Sink node: values flowing into Template.render parameters are treated
 * as potentially reflected into HTML output.
 */
class Jinja2RenderSink extends ReflectedXssFlow::SinkNode {
  Jinja2RenderSink() { this.asCfgNode() instanceof CallNode }

  override predicate isSinkNode(DataFlow::Node n) {
    exists(CallNode call |
      call = n.asCfgNode().(CallNode) and
      isJinja2RenderCall(call) and
      (
        exists(Expr a | a = call.getArg(_) and n.asExpr() = a)
        or
        exists(Expr kw | kw = call.getKeywordArg(_) and n.asExpr() = kw)
      )
    )
  }
}

/**
 * Sanitizer node: MarkupSafe and Jinja2 escaping functions as taint barriers.
 */
class MarkupSafeEscapeSanitizer extends ReflectedXssFlow::SanitizerNode {
  override predicate isSanitizerNode(DataFlow::Node n) {
```

```
51      exists(CallNode call |
52        call = n.asCfgNode().(CallNode) and
53        (
54          call = API::moduleImport("markupsafe")
55                    .getMember("escape").getACall()
56          or
57          call = API::moduleImport("markupsafe").getMember("Markup")
58                    .getMember("escape").getACall()
59          or
60          call = API::moduleImport("jinja2")
61                    .getMember("escape").getACall()
62        )
63      )
64    }
65  }
```

Estimated Impact

High. Jinja2 is the dominant templating engine in Python web applications built on Flask and related frameworks. Adding its rendering methods as XSS sinks substantially closes a systematic coverage gap that affects a large proportion of real-world Python web codebases.

Evidence Files

- `data/src/OFS/Image.py`
- `data/reviewboard/reviews/templatetags/reviewtags.py`
- `data/django/contrib/admin/widgets.py`

Bibliography

[1] Amine Lbath, Massih-Reza Amini, Aurelien Delaitre, and Vadim Okun. «AI Agentic Vulnerability Injection And Transformation with Optimized Reasoning». In: *arXiv preprint arXiv:2508.20866* (2025) (cit. on p. 1).

[2] Jerry Gamblin. *2024 CVE Data Review*. Accesso: 21 febbraio 2026. Jan. 2025. URL: `https://jerrygamblin.com/2025/01/05/2024-cve-data-review/` (cit. on p. 1).

[3] cve.icu. *Indice dei CVE*. Accesso: 21 febbraio 2026. 2026. URL: `https://cve.icu/index.html` (cit. on p. 1).

[4] Kaixuan Li, Sen Chen, Lingling Fan, Ruitao Feng, Han Liu, Chengwei Liu, Yang Liu, and Yixiang Chen. «Comparison and evaluation on static application security testing (sast) tools for java». In: *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2023, pp. 921–933 (cit. on p. 1).

[5] Pavel Avgustinov, Oege De Moor, Michael Peyton Jones, and Max Schäfer. «QL: Object-oriented queries on relational data». In: *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. 2016, pp. 2–1 (cit. on p. 1).

[6] Mingjie Shen, Akul Abhilash Pillai, Brian A Yuan, James C Davis, and Aravind Machiry. «Finding 709 Defects in 258 Projects: An Experience Report on Applying CodeQL to Open-Source Embedded Software (Experience Paper)». In: *Proceedings of the ACM on Software Engineering* 2.ISSTA (2025), pp. 1077–1100 (cit. on pp. 2, 11).

[7] Goran Piskachev, Matthias Becker, and Eric Bodden. «Can the configuration of static analyses make resolving security vulnerabilities more effective?-A user study». In: *Empirical Software Engineering* 28.5 (2023), p. 118 (cit. on p. 2).

[8] Zachary Douglas Wadhams, Clemente Izurieta, and Ann Marie Reinhold. «Barriers to using static application security testing (sast) tools: A literature review». In: *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering Workshops*. 2024, pp. 161–166 (cit. on pp. 2, 10).

[9] Wachiraphan Charoenwet, Patanamon Thongtanunam, Van-Thuan Pham, and Christoph Treude. «An empirical study of static analysis tools for secure code review». In: *Proceedings of the 33rd ACM SIGSOFT international symposium on software testing and analysis*. 2024, pp. 691–703 (cit. on pp. 2, 11).

[10] Tom Brown et al. «Language models are few-shot learners». In: *Advances in neural information processing systems* 33 (2020), pp. 1877–1901 (cit. on pp. 2, 14).

[11] Ze Sheng, Zhicheng Chen, Shuning Gu, Heqing Huang, Guofei Gu, and Jeff Huang. «Llms in software security: A survey of vulnerability detection techniques and insights». In: *ACM Computing Surveys* 58.5 (2025), pp. 1–35 (cit. on p. 2).

[12] Yifan Yao, Jinhao Duan, Kaidi Xu, Yuanfang Cai, Zhibo Sun, and Yue Zhang. «A survey on large language model (llm) security and privacy: The good, the bad, and the ugly». In: *High-Confidence Computing* 4.2 (2024), p. 100211 (cit. on p. 2).

[13] Avishree Khare, Saikat Dutta, Ziyang Li, Alaia Solko-Breslin, Rajeev Alur, and Mayur Naik. «Understanding the effectiveness of large language models in detecting security vulnerabilities». In: *2025 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE. 2025, pp. 103–114 (cit. on p. 2).

[14] Ziyang Li, Saikat Dutta, and Mayur Naik. «IRIS: LLM-assisted static analysis for detecting security vulnerabilities». In: *arXiv preprint arXiv:2405.17238* (2024) (cit. on p. 3).

[15] Junda He, Christoph Treude, and David Lo. «Llm-based multi-agent systems for software engineering: Literature review, vision, and the road ahead». In: *ACM Transactions on Software Engineering and Methodology* 34.5 (2025), pp. 1–30 (cit. on pp. 3, 18).

[16] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. «React: Synergizing reasoning and acting in language models». In: *The eleventh international conference on learning representations*. 2022 (cit. on pp. 5, 16, 21).

[17] Hazim Hanif, Mohd Hairul Nizam Md Nasir, Mohd Faizal Ab Razak, Ahmad Firdaus, and Nor Badrul Anuar. «The rise of software vulnerability: Taxonomy of software vulnerabilities detection and machine learning approaches». In: *Journal of Network and Computer Applications* 179 (2021), p. 103009 (cit. on pp. 8, 12, 13).

[18] National Institute of Standards and Technology (NIST). *NVD - Search and Statistics*. `https://nvd.nist.gov/vuln/search/nvd/home?resultType=statistics`. Accessed: 2026-02-22 (cit. on p. 8).

[19] Stefano Simonetto and Peter Bosch. «Comprehensive threat analysis and systematic mapping of CVEs to MITRE framework». In: *Proceedings of the First International Conference on Natural Language Processing and Artificial Intelligence for Cyber Security*. Ed. by Ruslan Mitkov, Saad Ezzini, Tharindu Ranasinghe, Ignatius Ezeani, Nouran Khallaf, Cengiz Acarturk, Matthew Bradbury, Mo El-Haj, and Paul Rayson. Lancaster, UK: International Conference on Natural Language Processing and Artificial Intelligence for Cyber Security, July 2024, pp. 32–41. URL: `https://aclanthology.org/2024.nlpaics-1.4/` (cit. on p. 9).

[20] The MITRE Corporation. *Common Weakness Enumeration (CWE)*. `https://cwe.mitre.org`. Accessed: 2026-02-22. MITRE, 2026 (cit. on p. 9).

[21] Philipp Kühn, David N Relke, and Christian Reuter. «Common vulnerability scoring system prediction based on open source intelligence information sources». In: *Computers & Security* 131 (2023), p. 103286 (cit. on p. 9).

[22] Adrián Herrera Jerónimo, Patricia Martínez Moreno, José Antonio Vergara Camacho, and Gerardo Contreras Vega. «Techniques of SAST Tools in the Early Stages of Secure Software Development: A Systematic Literature Review». In: *2024 IEEE International Conference on Engineering Veracruz (ICEV)*. 2024, pp. 1–8. DOI: `10.1109/ICEV63254.2024.10766004` (cit. on pp. 9, 29).

[23] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. «Why don't software developers use static analysis tools to find bugs?» In: *2013 35th International Conference on Software Engineering (ICSE)*. IEEE. 2013, pp. 672–681 (cit. on p. 9).

[24] José D'Abruzzo Pereira, Naghmeh Ivaki, and Marco Vieira. «Characterizing buffer overflow vulnerabilities in large c/c++ projects». In: *IEEE Access* 9 (2021), pp. 142879–142892 (cit. on p. 9).

[25] Junjie Wang, Song Wang, and Qing Wang. «Is there a" golden" feature set for static warning identification? an experimental evaluation». In: *Proceedings of the 12th ACM/IEEE international symposium on empirical software engineering and measurement*. 2018, pp. 1–10 (cit. on p. 9).

[26] Alessandro Marchetto. «Can explainability and deep-learning be used for localizing vulnerabilities in source code?» In: *Proceedings of the 5th ACM/IEEE International Conference on Automation of Software Test (AST 2024)*. 2024, pp. 110–119 (cit. on p. 10).

[27] Dinis Barroqueiro Cruz, João Rafael Almeida, and José Luís Oliveira. «Open source solutions for vulnerability assessment: A comparative analysis». In: *IEEE Access* 11 (2023), pp. 100234–100255 (cit. on p. 10).

[28] CodeQL. *CodeQL Documentation*. Accessed: December, 2025. 2025. URL: `https://codeql.github.com/docs/codeql-language-guides/basic-query-for-python-code/#` (cit. on pp. 10, 11, 24, 30).

[29] Jorrit Kronjee, Arjen Hommersom, and Harald Vranken. «Discovering software vulnerabilities using data-flow analysis and machine learning». In: *Proceedings of the 13th international conference on availability, reliability and security*. 2018, pp. 1–10 (cit. on p. 12).

[30] Muhammad Noman Khalid, Humera Farooq, Muhammad Iqbal, Muhammad Talha Alam, and Kamran Rasheed. «Predicting web vulnerabilities in web applications based on machine learning». In: *International Conference on Intelligent Technologies and Applications*. Springer. 2018, pp. 473–484 (cit. on p. 12).

[31] Hoa Khanh Dam, Truyen Tran, Trang Pham, Shien Wee Ng, John Grundy, and Aditya Ghose. «Automatic Feature Learning for Predicting Vulnerable Software Components». In: *IEEE Transactions on Software Engineering* 47.1 (2021), pp. 67–85. DOI: `10.1109/TSE.2018.2881961` (cit. on p. 12).

[32] Kazi Zakia Sultana and Byron J Williams. «Evaluating micro patterns and software metrics in vulnerability prediction». In: *2017 6th International Workshop on Software Mining (SoftwareMining)*. IEEE. 2017, pp. 40–47 (cit. on p. 12).

[33] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. «Vuldeepecker: A deep learning-based system for vulnerability detection». In: *arXiv preprint arXiv:1801.01681* (2018) (cit. on p. 12).

[34] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. «Automated vulnerability detection in source code using deep representation learning». In: *2018 17th IEEE international conference on machine learning and applications (ICMLA)*. IEEE. 2018, pp. 757–762 (cit. on p. 13).

[35] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. «Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks». In: *Advances in neural information processing systems* 32 (2019) (cit. on p. 13).

[36] National Institute of Standards and Technology. *Software Assurance Reference Dataset (SARD)*. Accessed: 2026-02-26. SAMATE, NIST. 2025. URL: https://samate.nist.gov/SARD/ (cit. on p. 13).

[37] Guanjun Lin, Sheng Wen, Qing-Long Han, Jun Zhang, and Yang Xiang. «Software Vulnerability Detection Using Deep Neural Networks: A Survey». In: *Proceedings of the IEEE* 108.10 (2020), pp. 1825–1848. DOI: 10.1109/JPROC.2020.2993293 (cit. on p. 13).

[38] Yue Li, Xiao Li, Hao Wu, Minghui Xu, Yue Zhang, Xiuzhen Cheng, Fengyuan Xu, and Sheng Zhong. *Everything You Wanted to Know About LLM-based Vulnerability Detection But Were Afraid to Ask*. 2025. arXiv: 2504.13474 [cs.CR]. URL: https://arxiv.org/abs/2504.13474 (cit. on pp. 14, 15, 33, 34).

[39] Zeyu Gao, Hao Wang, Yuchen Zhou, Wenyu Zhu, and Chao Zhang. «How far have we gone in vulnerability detection using large language models». In: *arXiv preprint arXiv:2311.12420* (2023) (cit. on p. 14).

[40] Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. «Vulnerability detection with code language models: How far are we?» In: *arXiv preprint arXiv:2403.18624* (2024) (cit. on p. 14).

[41] Xueying Du et al. «Vul-rag: Enhancing llm-based vulnerability detection via knowledge-level rag». In: *ACM Transactions on Software Engineering and Methodology* (2024) (cit. on p. 14).

[42] Benjamin Steenhoek, Md Mahbubur Rahman, Monoshi Kumar Roy, Mirza Sanjida Alam, Hengbo Tong, Swarna Das, Earl T Barr, and Wei Le. «To err is machine: Vulnerability detection challenges llm reasoning». In: *arXiv preprint arXiv:2403.17218* (2024) (cit. on p. 14).

[43] Yu Nong, Mohammed Aldeen, Long Cheng, Hongxin Hu, Feng Chen, and Haipeng Cai. «Chain-of-thought prompting of large language models for discovering and fixing software vulnerabilities». In: *arXiv preprint arXiv:2402.17230* (2024) (cit. on p. 14).

[44] Saad Ullah, Mingji Han, Saurabh Pujar, Hammond Pearce, Ayse Coskun, and Gianluca Stringhini. «Llms cannot reliably identify and reason about security vulnerabilities (yet?): A comprehensive evaluation, framework, and benchmarks». In: *2024 IEEE symposium on security and privacy (SP)*. IEEE. 2024, pp. 862–880 (cit. on p. 14).

[45] Liangming Pan, Michael Saxon, Wenda Xu, Deepak Nathani, Xinyi Wang, and William Yang Wang. «Automatically correcting large language models: Surveying the landscape of diverse self-correction strategies». In: *arXiv preprint arXiv:2308.03188* (2023) (cit. on p. 14).

[46] Dan Ristea, Vasilios Mavroudis, and Chris Hicks. «Benchmarking OpenAI o1 in Cyber Security». In: *arXiv preprint arXiv:2410.21939* (2024) (cit. on p. 15).

[47] Yuntao Wang, Yanghe Pan, Zhou Su, Yi Deng, Quan Zhao, Linkang Du, Tom H. Luan, Jiawen Kang, and Dusit Niyato. «Large Model-Based Agents: State-of-the-Art, Cooperation Paradigms, Security and Privacy, and Future Trends». In: *IEEE Communications Surveys Tutorials* 28 (2026), pp. 1906–1949. DOI: 10.1109/COMST.2025.3576176 (cit. on p. 16).

[48] Zhiheng Xi et al. «The rise and potential of large language model based agents: A survey». In: *Science China Information Sciences* 68.2 (2025), p. 121101 (cit. on p. 16).

[49] Yuheng Cheng et al. «Exploring large language model based intelligent agents: Definitions, methods, and prospects». In: *arXiv preprint arXiv:2401.03428* (2024) (cit. on p. 16).

[50] Mehmet Fırat and Saniye Kuleli. «What if gpt4 became autonomous: The auto-gpt project and use cases». In: *Journal of Emerging Computer Technologies* 3.1 (2023), pp. 1–6 (cit. on p. 16).

[51] Qingyun Wu et al. «Autogen: Enabling next-gen LLM applications via multi-agent conversations». In: *First conference on language modeling*. 2024 (cit. on pp. 16, 18).

[52] Yanfei Chen, Jinsung Yoon, Devendra Singh Sachan, Qingze Wang, Vincent Cohen-Addad, Mohammadhossein Bateni, Chen-Yu Lee, and Tomas Pfister. «Re-invoke: Tool invocation rewriting for zero-shot tool retrieval». In: *Findings of the Association for Computational Linguistics: EMNLP 2024*. 2024, pp. 4705–4726 (cit. on p. 16).

[53] Michael Ahn et al. «Do as i can, not as i say: Grounding language in robotic affordances». In: *arXiv preprint arXiv:2204.01691* (2022) (cit. on pp. 16, 17).

[54] Lei Wang et al. «A survey on large language model based autonomous agents». In: *Frontiers of Computer Science* 18.6 (2024), p. 186345 (cit. on pp. 16, 17).

[55] Chan Hee Song, Jiaman Wu, Clayton Washington, Brian M Sadler, Wei-Lun Chao, and Yu Su. «Llm-planner: Few-shot grounded planning for embodied agents with large language models». In: *Proceedings of the IEEE/CVF international conference on computer vision*. 2023, pp. 2998–3009 (cit. on p. 16).

[56] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. «Chain-of-thought prompting elicits reasoning in large language models». In: *Advances in neural information processing systems* 35 (2022), pp. 24824–24837 (cit. on p. 16).

[57] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. «Tree of thoughts: Deliberate problem solving with large language models». In: *Advances in neural information processing systems* 36 (2023), pp. 11809–11822 (cit. on p. 16).

[58] Maciej Besta et al. «Graph of thoughts: Solving elaborate problems with large language models». In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 38. 16. 2024, pp. 17682–17690 (cit. on p. 16).

[59] Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. «Reflexion: Language agents with verbal reinforcement learning, 2023». In: *URL https://arxiv. org/abs/2303.11366* 8 (2024) (cit. on p. 16).

[60] Minki Kang, Seanie Lee, Jinheon Baek, Kenji Kawaguchi, and Sung Ju Hwang. «Knowledge-augmented reasoning distillation for small language models in knowledge-intensive tasks». In: *Advances in Neural Information Processing Systems* 36 (2023), pp. 48573–48602 (cit. on p. 17).

[61] Harsh Trivedi, Niranjan Balasubramanian, Tushar Khot, and Ashish Sabharwal. «Interleaving retrieval with chain-of-thought reasoning for knowledge-intensive multi-step questions». In: *Proceedings of the 61st annual meeting of the association for computational linguistics (volume 1: long papers)*. 2023, pp. 10014–10037 (cit. on p. 17).

[62] Na Liu, Liangyu Chen, Xiaoyu Tian, Wei Zou, Kaijiang Chen, and Ming Cui. «From llm to conversational agent: A memory enhanced architecture with fine-tuning of large language models». In: *arXiv preprint arXiv:2401.02777* (2024) (cit. on p. 17).

[63] Patrick Lewis et al. «Retrieval-augmented generation for knowledge-intensive nlp tasks». In: *Advances in neural information processing systems* 33 (2020), pp. 9459–9474 (cit. on p. 17).

[64] Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V Chawla, Olaf Wiest, and Xiangliang Zhang. «Large language model based multi-agents: A survey of progress and challenges. arXiv 2024». In: *arXiv preprint arXiv:2402.01680* 10 (2024) (cit. on p. 17).

[65] Huao Li, Yu Chong, Simon Stepputtis, Joseph P Campbell, Dana Hughes, Charles Lewis, and Katia Sycara. «Theory of mind for multi-agent collaboration via large language models». In: *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. 2023, pp. 180–192 (cit. on p. 17).

[66] Zane Durante et al. «Agent ai: Surveying the horizons of multimodal interaction». In: *arXiv preprint arXiv:2401.03568* (2024) (cit. on p. 17).

[67] Ceyao Zhang et al. «Proagent: building proactive cooperative agents with large language models». In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 38. 16. 2024, pp. 17591–17599 (cit. on p. 17).

[68] Sirui Hong et al. «MetaGPT: Meta programming for a multi-agent collaborative framework». In: *The twelfth international conference on learning representations*. 2023 (cit. on pp. 17, 18).

[69] Chen Qian et al. «Chatdev: Communicative agents for software development». In: *Proceedings of the 62nd annual meeting of the association for computational linguistics (volume 1: Long papers)*. 2024, pp. 15174–15186 (cit. on p. 18).

[70] Accessed: December, 2025. 2025. URL: https://pypi.org/project/duckduckgo-search/ (cit. on pp. 24, 29).

[71] starsofchance. *CVEfixes_v1.0.8*. https://huggingface.co/datasets/starsofchance/CVEfixes_v1.0.8. Hugging Face dataset, licenza MIT, accesso il 7 marzo 2026. 2026 (cit. on p. 32).

[72] National Institute of Standards and Technology (NIST). *National Vulnerability Database (NVD)*. https://nvd.nist.gov/. Accessed: 2026-03-07. 2026 (cit. on p. 32).

[73] MITRE. *CWE-89*. Accessed: December, 2025. 2025. URL: https://cwe.mitre.org/data/definitions/89.html (cit. on p. 32).

[74] MITRE. *CWE-79*. Accessed: December, 2025. 2025. URL: https://cwe.mitre.org/data/definitions/79.html (cit. on p. 32).

[75] MITRE. *CWE-78*. Accessed: December, 2025. 2025. URL: https://cwe.mitre.org/data/definitions/78.html (cit. on p. 32).

[76] OpenAI. Accessed: December, 2025. 2025. URL: https://developers.openai.com/api/docs/models/gpt-5.2 (cit. on p. 33).

[77] OpenAI. *GPT-4o mini: advancing cost-efficient intelligence*. Accessed: December, 2025. 2025. URL: https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/ (cit. on p. 34).

[78] OpenAI. *GPT-4o mini*. Accessed: December, 2025. 2025. URL: https://platform.openai.com/docs/models/gpt-4o-mini (cit. on p. 34).

[79] LangChain. *LangChainBuild agents faster, your way.* Accessed: December, 2025. 2025. URL: https://www.langchain.com/langchain (cit. on p. 34).

[80] LangChain. *Balance agent control with agency.* Accessed: December, 2025. 2025. URL: https://www.langchain.com/langgraph (cit. on p. 34).