POLITECNICO DI TORINO

Master's Degree in Cybersecurity

Master's Thesis

AI Agents for Software Vulnerability Detection

Supervisors
Prof. Adam Alami
Prof. Danilo Giordano
Prof. Matteo Boffa

Candidate
Rebecca De Rosa

Data

Acknowledgements

Summary

Contents

List of Figures

Chapter 1

Introduction

Nowadays, the cybersecurity landscape is continuously challenged by the increasing sophistication of software systems and cyber-attacks [1]. Software vulnerabilities represent an ever-growing problem, increasing risks and having potentially large-scale impact, as demonstrated by incidents like the reported vulnerabilities in Apache Log4j2 [2]. The criticalness of this problem is highlighted in statistics: in the recent years the number of reported CVEs (Common Vulnerability and Exposure) has significantly grown, reaching 29,000 in 2023 [2], with the total number rising by 25% in 2024 alone [1] with more than 49,000 vulnerabilities identified by National Vulnerability Database (NVD) in 2024 [3].
The threat is further aggravated by the widespread adoption of open-source software (OSS) [4]. Potential vulnerability risks can cause significant threats to national and economic security. Incidents such as the 2024 Microsoft IT blackout caused by a unreliable CrowdStrike update [5], which disrupted global services, and the XZ Utils backdoor attack [6] orchestrated by a malicious entity, which compromised numerous Linux systems, underscore the critical need for robust software security measures [4]. Specifically, since the code is open, vulnerabilities can be found easier, explored, and propagated faster due to code reuse, underscoring again the critical need for rapid vulnerability risk mitigation.
Furthermore, in software development, quality assurance (QA) assumes a critical role. In fact, it is fundamental to ensure that software products not only meet functional requirements but also satisfy performance and security standards before being delivered to users [7]. In addition, since software systems become an integral part of business operations and everyday life, the effects of defects and failures can be severe, causing financial loss, reputation damage, and less trust in such products [7].
In this landscape, traditional manual testing, while useful for specific tasks — particularly those requiring human intervention and contextual understanding — has become increasingly limited against the scale and complexity of contemporary software applications [7], prompting researches to investigate other methods, including AI-supported approaches [2].
This transition is evident, with 91% of studies using AI-based methods for Software Vulnerability Detection (SVD) [2]. Large Language Models (LLMs) have increased this capability, driving a new technological revolution in the field of Software Engineering (SE) and generating human-like outputs [4]. This progress is increasingly accelerated by the emergence of AI agents and multi-agent frameworks [1], [3], [4], which aim to create effective automated detection methods [1].

This thesis aims to take advantage of this shift, exploring the integration of multi-agent architectures with source code analysis through Static Application Security Testing SAST tool for advanced vulnerability detection.

1.1 Background

In early traditional vulnerability detection (VD), security experts manually discovered bugs, action that could take months or even years [8], [4], requiring considerable time and specialized personnel [9], [10].
In response, tools such as Static Application Security Testing (SAST) emerged. SAST tools analyze source code without execution, operating as a white-box approach, generating reports on alerts and often relying on predefined rules based on CWE (Common Weakness Enumeration) or CVE [9]. However, these methods are limited by their reliance on *hand-engineered rules* [1] (so, written by hand) and the complexity of their analysis, resulting in high false alarm rates [1], [10]. Their dependency on predefined rules detects only a limited subset of possible errors [11].
To address these limitations, data-driven techniques, including those with LLMs, have been investigated [11], [10]. However, even these advanced approaches came across critical challenges:

- Data Quality and Scale: The effectiveness of AI-based SVD relies critically on large, accurately labeled datasets [1]. Unfortunately, existing vulnerability datasets suffer from poor data quality and high duplication rates [12]. Furthermore, low-quality data can cause machine learning models to learn fake correlations, and this leads to the need for extensive human intervention and effort for the creation of high-quality data [3].

- LLM Reasoning Limitations: Detecting vulnerabilities requires higher code comprehension capabilities from LLMs [4]. Current LLM-based approaches are often limited to function-level VD, since complex vulnerabilities are frequently cross-procedural, i.e. it can't be detected by studying only an isolated function[4]. Most importantly, VD methods are limited by the fact that contextual understanding is restrictive to single-round interactions, the prevent the model from considering a broader context [4]. The challenge consists of overcoming these limitations to achieve optimal accuracy and scalability [10].

1.2 Motivation

The primary motivation for this thesis is to investigate alternative methods to augment existing security practices, necessitated by critical challenges in current approaches. The investigation is motivated by two key limitations in current vulnerability detection methods:

- Manual effort in high-precision detection: high-precision detection tools like CodeQL require security experts to manually write and refine queries, a process that demands substantial manual effort [3], [10]. Furthermore, this labor-intensive process limits scalability and efficiency.

- Standalone LLM usage: while LLMs offer heuristic potential, standalone application may not yield efficient results. First, to achieve the desired outcomes, the user should have adequate knowledge of prompt engineering techniques. Second, a single prompt may not lead to the desired results either. Vulnerabilities detection requires separate, subsequent, and interdependent tasks; each task informs the subsequent task to achieve the desired outcome. Therefore, a synchronized multi-agents architecture is a nice alternative. [4].

This thesis proposes to bridge this gap by introducing an Agent-Based Architecture. My approach exploits the interactive capabilities of multi-agent systems [4] and simulates the reasoning of a human security analyst to automate the high-precision query generation process. Specifically, while neither manual tools nor standalone LLMs can effectively replicate the collaborative reasoning loop utilized by human security analyst, the usage of a multi-agent architecture, managed by an Orchestrator Agent that coordinates the actions of all the other agents, is able to better simulate and most of all accelerate the human reasoning, thereby reducing execution times. More specifically, the goal is to investigate how a multi-agent architecture can be developed to achieve this proposal and, subsequently, to test the efficiency and accuracy of the agents, trained via appropriate prompts, in generating CodeQL queries. In particular, this thesis aims to evaluate how to build a multi-agent architecture and how an agent is able to generate up-to-date and enhanced CodeQL queries. The Results chapter, therefore, outlines the comparison between predefined original CodeQL queries and new agent-generated ones.

1.3 Research Problem and Challenges

The research problem investigated is the lack of an automated, contextually-aware and autonomous system capable of generating high-precision detection rules, moving beyond the limitations outlined in the previous paragraph. The challenge is to create a method that overcomes the limitations of standalone LLM-based SVD (Software Vulnerability Detection) models to perform the necessary iterative and reasoning [4], automatically translating abstract vulnerability patterns into executable CodeQL queries of high-precision. Furthermore, the problem consists of evaluating the actual limit of the agent's autonomy in creating CodeQL queries. In fact, since the CodeQL code (QL) is complex, highly specialized and subject to frequent libraries updates, this study investigates how close an agent can get to generating new and correct queries.

1.4 Research Question

Therefore, I propose to investigate the following research questions (RQ1 and RQ2):

- ***RQ1***: *How can a multi-agent architecture simulate the reasoning process of a security analyst and reduce the limitations of manual effort and the constraints of standalone LLM application in Software Vulnerability Detection (SVD) by:*
  - *combining the analytical power of the CodeQL SAST tool with the Large Language Models (LLMs) capabilities,*
  - *automatically generate and refine high-precision detection CodeQL queries, providing reports which are insights into the understanding of the SDV, potentially assisting their mitigation effort.*
- ***RQ2***: *How does the quality of the agent offer better detection coverage and fewer false positives than the standard, predefine ones?*

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 SVD, the limitations of traditional SAST and LLM-based approaches, and the foundational concepts of AI agent architectures.
- Chapter 3: outlines the Related works to the thesis.
- Chapter 4: presents the proposed Multi-Agent SVD Architecture, detailing the role of each agent, the interaction protocols between those, design principles, and the specific mechanism for generating and refining CodeQL queries.
- Chapter 5: describes the Experimental Setup and method, including the datasets used, the implementation details of the CodeQL-LLM integration, and the metrics used for evaluation.
- Chapter 6: presents the Results offering an analysis of the architecture's performance.
- Chapter 7: presents the Discussion and summarizes the key contributions of the research, discusses the theoretical and practical implications and outlines the limitations.
- Chapter 8: Conclusions and suggested avenues for Future Work.

Chapter 2

Background

As discussed many times above, Software vulnerabilities are the prevalent issues in software systems [13], posing various risks such as the compromise of sensitive information [14] and system failures [15]. To address this challenge, in the recent years, researchers have proposed machine learning (ML) and deep learning (DL) approaches to identify vulnerabilities in source code [16, 17, 18, 19]. While previous ML/DL-based vulnerability detection approaches have demonstrated promising results, they have primarily relied on either medium-size pretrained models such as CodeBERT [17], [20] for training smaller neural networks (such as Graph Neural Networks [18]) from scratch. Recent developments in Large Pre-Trained Language Models (LLMs) have demonstrated impressive few-shot learning capabilities across diverse tasks [21, 22, 23, 24, 25]. However, the performance of LLMs on security-oriented tasks, particularly vulnerability detection, remains largely unexplored. Moreover, LLMs are gradually starting to be used in software engineering (SE), as seen in automated program repair [21].

When it comes to the history of AI agents, we need to go back to the 1950s. In these years, Alan Turing introduced the Turing Test to assess whether machines could exhibit intelligence comparable to humans [26]. These artificial entities, commonly known as "agents", are the core components of artificial intelligence (AI) systems. By definition, AI agents (also known as agentic AI [27]) are autonomous entities able to understanding and responding to human inputs, perceiving their environment, making decisions, and taking actions across physical, virtual, or mixed-reality settings to achieve specific goals [28]. They can be software-based or physical entities and they can work independently or in collaboration with humans or other agents. Since the mid-20th century, significant progress has been made in the development of AI agents [29], [30], such as Deep Blue, AlphaGo, and AlphaZero. Despite these advances, at the beginning the researches often neglected the cultivation of general-purpose capabilities within AI models such as long-term planning, multi-task generalization, and knowledge retention. With the rise of large models (LMs), including large language models (LLMs) such as OpenAI GPT-4, DeepSeek-R1, and Google PaLM 2 new possibilities were unlocked.
LM agents significantly enhanced the inherent capabilities of AI systems, providing a versatile foundation for the next-generation AI agents [31]. Operating as the "brain" of AI agents, LMs empower them with advanced capabilities in human-machine interaction (HMI), few/zero-shot planning, contextual understanding, knowledge retention, and general-purpose task solving across physical, virtual, or mixed-reality environments [28], [32].

Specifically, LM agents are generally devided into two categories:

- Virtual LM agents (such as AutoGPT [33] and AutoGen [34]): they can autonomously interpret human instructions and use various tools (e.g., search engines and external APIs) to gather information and complete intricate tasks [35]. For instance an LM-powered personal assistant can generate personalized travel plans, set reminders, and manage tasks while continuously learning and adapting in dynamic environments.
- Embodied LM agents (such as AI's and Tesla's Optimus): they interact directly with the physical world. These agents perceive and interact with their surroundings, allowing them to solve real-world problems. For instance, an LM-powered household robot analyzes room layouts, surface types, and obstacles, to devise customized cleaning strategies rather than merely following generic instructions.

LMM agents are recognized as a significant step towards achieving artificial general intelligence (AGI) and have been widely applied across fields such as web search [36], virtual assistants [37], Metaverse gaming [38], robotics [39], autonomous vehicles [40], and automated attack penetration [41]. As reported by MarketsandMarkets [42], the worldwide market for autonomous AI and autonomous agents was valued at USD 4.8 billion in 2023 and is projected to grow at a CAGR of 43%, reaching USD 28.5 billion by 2028. LM agents have attracted global attention, and leading technology giants including Google, OpenAI, Microsoft, IBM, AWS, Oracle, NVIDIA, and Baidu are venturing into the LM agent industry [26].

Chapter 3

Related works

3.1 Vulnerability Detection

The paper *Benchmarking LLMs and LLM-based Agents in Practical Vulnerability Detection for Code Repositories* [43] contains a detailed and comprehensive description of related works, starting with Vulnerability Detection (VD) and ending with those of LLM-based agents. In particular, it outlines that several benchmarks have been proposed for function-level vulnerability detection. Specifically the paper cites:

- BigVul [44]: it collects C/C++ vulnerabilities from the CVE database, filtering out entries without public Git repositories, and labels functions as vulnerable or non-vulnerable based on commit fixes.
- MegaVul [45]: it improves upon existing benchmarks by using code parsing tools for accurate function extraction and de-duplicating functions referenced by multiple CVEs.
- DiverseVul [46]: it ensures data quality by filtering vulnerabilities by introducing commits with specific keywords and de-duplicating function bodies with hash functions.
- PrimeVul [12]: it addresses data quality challenges by proposing filtering rules to handle noise labels and duplicated functions.
- ReposVul [47] (for repository-level vulnerability detection): it addresses issues with tangled and outdated patches, using trace-based filtering to ensure data quality and integrating repository-level features to provide richer context for detection.
- VulEval [48]: it provides a framework that collects high-quality data from sources like Mend.io Vulnerability Database [49] and National Vulnerability Database [50], including contextual information like caller-callee relationships, which consist of an interaction where one function (the caller) is temporary suspended to allow another function (the callee) to execute its task and then return control to the caller.

3.2 LLM-based Vulnerability Detection

The paper [43] further outlines the related work on LLM-based Vulnerability Detection. It asserts that recent studies have explored the use of LLMs for vulnerability detection, emphasizing their potential to enhance both the identification and the explanation of software vulnerabilities. In particular:

- InferROI [51] enhances static resource leak detection with LLM-inferred resource operation oriented intentions.
- LLM4SA [52] integrates language models with SAST tools, leveraging LLMs to inspect static analysis warnings and significantly reduce false positives.
- LLM4Vuln [53] enhances LLM execution by incorporating more context through a retrieval-augmented generation (RAG) pipeline and static analysis.
- LSAST [54] further explores context augmentation, comparing multiple RAG pipelines using static analysis outputs, vulnerability reports, and code abstraction.
- Vul-RAG constructs a vector database of vulnerability reports alongside an language model engine.

Therefore, based on these latest researches, the integration of LLMs into vulnerability scanning does not aim to replace static analysis tools, but rather to create a better and hybrid system. In such a system, LLMs act as the brain, and thus as an enhanced reasoning engine that improves precision and accuracy.

3.3 LLMs and LLM-based Agents

Finally, the paper [43] explores the literature review of AI agents. It describes that various methods have been explored to enhance LLM performance, with prompt augmentation serving as a key approach for enriching prompts and improving the model's reasoning process. In particular, both Chain-of-thought (CoT) prompting, where instructions like "Let's think step by step" guide the model to break problems into sub-problems, and few-shot prompting, which provides example traces to enable in-context learning without modifying model weights, are frequently employed in LLM-based vulnerability detection. The paper describes the main types of agents:

- Reasoning and Acting (ReAct) agents (introduced by [55]): they generate reasoning and action traces in an interleaved manner to interact with their environment and analyze resulting observations. This iterative process continues until the agent determines a final answer [56] - these are the type of agents used in this thesis.
- Reflexion agents: they are similar to ReAct agents. However, they focus on self-reflection and dynamic memory updates alongside with reinforcement learning.
- Self-refine: they are agents where the same language model is instructed to provide feedback based on the output [57].
- Other multi-agent systems, such as Alpha-Codium, where the agents are represented as nodes on graphs [58].

Furthermore, the paper asserts that *agentic* architectures are among the most promising state-of-the-art technologies, but they remain underexplored in vulnerability detection, at least in the year the paper was written. In fact, in the last year (2025) there were some studies investigating the use of AI Agent in VD. Among them, I mention *MAVUL:*

Multi-Agent Vulnerability Detection via Contextual Reasoning and Interactive Refinement [4], which proposes a multi-agent architecture for VD, focusing on the improvement of the internal agent's reasoning.
Therefore, the proposed multi-agent architecture in thia theaia has the objective to investigate the transition of static analysis from a manual, expert-dependent approach to an adaptive SAST paradigm powered by AI. Traditionally, as mentioned above, CodeQL's effectiveness is limited by the complexity of the Datalog logic language, which could take time to learn best, and the static nature of generic query packages, which often cause false positives and slow response times in the face of increasingly frequent zero-day threats. This project introduces a self-correction cycle in which the Analyzer Agent critically reviews existing queries, identifying logical gaps that human reviewers may miss or take too long to identify. By automating the Reasoning-Suggestion-Creation loop, the system attempts to overcome the constraints of manually writing rules, allowing it to generate custom security specifications.

Chapter 4

Proposed agentic architecture

4.1 Workflow

In this thesis I introduce and evaluate an architecture composed by four LLM Agents that autonomously validate and detect the vulnerabilities contained in the provided dataset and, consequently, try to generate better and new high-precision CodeQL queries.

As illustrated in the following image (Figure 4.1), the workflow begins with the **Or-**

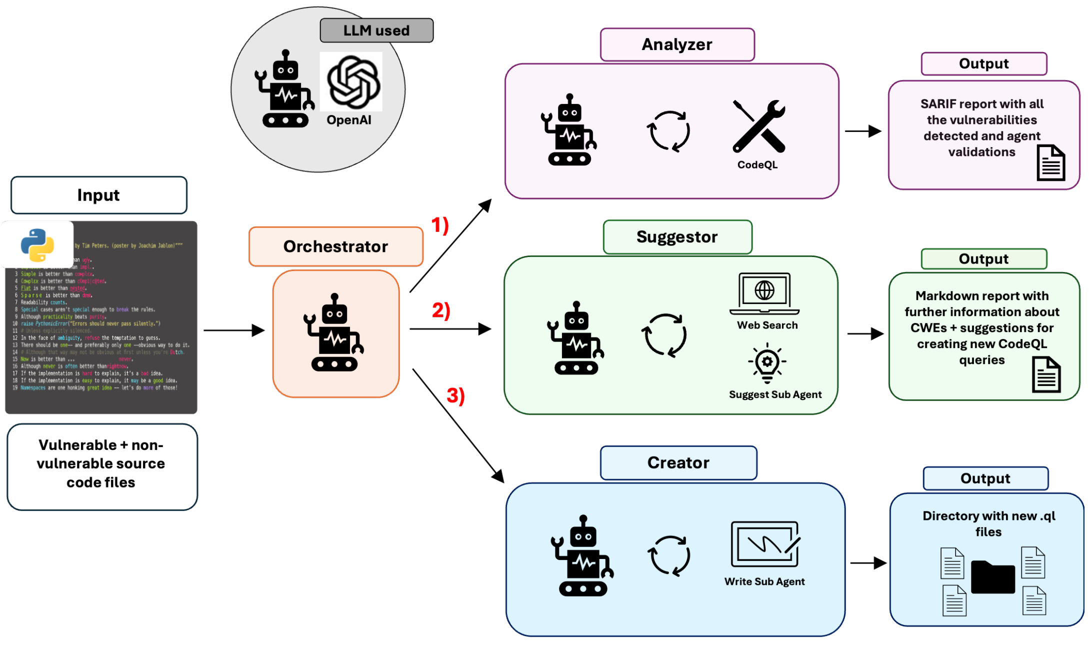

LLM used
OpenAI
Input
Vulnerable + non-vulnerable source code files
Orchestrator
1)
2)
3)
Analyzer
CodeQL
Output
SARIF report with all the vulnerabilities detected and agent validations
Suggestor
Web Search
Suggest Sub Agent
Output
Markdown report with further information about CWEs + suggestions for creating new CodeQL queries
Creator
Write Sub Agent
Output
Directory with new .ql files

Figure 4.1. Overview of agent architecture. The Orchestrator receives the vulnerable source code files and sequentially invokes the system components. First, it calls the Analyzer agent, which produces the SARIF report. This report is returned to the Orchestrator, which passes it as input to the Suggestor Agent. The Suggestor produces a Markdwon report and sends it back to the Orchestrator. Finally, the Orchestrator invokes the Creator, which uses the Markdown report as input to create new .ql files containing new CodeQL queries.

chestrator agent, which manages the execution flow by simply initializing and invoking the other three agents.

The first among these is the **Analyzer agent**, implemented with the ReAct framework[1]. The agent's workflow begins with the CodeQL tool call. As detailed later in this chapter (4.4.1), this tool scans the files and reports the vulnerabilities detected into a standardized SARIF report. Subsequently, the Analyzer performs the reasoning step by validating the SARIF findings to filter out False Positive and evaluating the accuracy of the detection by comparing the identified vulnerabilities with the logic of the existing CodeQL queries used during the preliminary scan.
Subsequently, the **Suggestor Agent** is introduced. As a ReAct agent it interprets the Analyzer Agent's results (SARIF report), identifies the detected vulnerabilities and creates:

- a detailed vulnerabilities report;
- suggestions for new, improved and predictive CodeQL queries;
- structured reasoning, which involves summarizing previous events, developing a thought based on the context and executing the appropriate action.

Lastly, the **Creator Agent**, also a ReAct agent, receives the Suggestor Agent's report and identifies all targeted CWEs that require CodeQL query generation. It, then, iteratively produces one '.ql' file for each CWE using reasoning steps and tool invocations.
At this point, the final result is a new directory containing all the new expected improved CodeQL queries, ready to be tested.

4.2 LLM agent architecture

In order to fully understand this architecture, is important to know the fundamentals of this new generation of AI Agents. An agent is an autonomous system that interacts with its environment to achieve specific goals. It perceives inputs, processes information, and takes actions that influence the environment according to its objectives [60]. When powered by LLMs, agents leverage the model's reasoning and planning abilities to guide decision-making. Key aspects of LLM-based agents include tool use, planning and reasoning, memory and context retention, and multi-agent collaboration. Evaluating these capabilities helps determine an agent's strengths and weaknesses on a more granular level [61].
The agent core components are:

- Tool Use: It enables LLM-based agent to retrieve grounding information, perform actions, and interact with external environments.
- Planning and Reasoning: Planning involves selecting the correct set of tools in an appropriate order. At the same time, reasoning enables agents to make context-aware decisions, either ahead of time or dynamically during task execution [61].
- Actions: Allow the agents to interact with the environment through predefined tools, i.e., executing applications, querying APIs, or updating internal memory.
- Memory: Agents use short-term memory for immediate interactions, and long term memory to retain past knowledge [62].

[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 [59].

A critical phase in agent development is the prompts drafting. Specifically, the prompt defines the task to achieve, the tools the agent can use, and the input data (which accumulates over interactions).
The agent reasons and decides which actions to perform next, runs them, and observes the environment's reaction to such actions. Through repeated cycles of perception, reasoning, and action, LLM agents autonomously tackle complex tasks that would be infeasible to script manually.
Importantly, the prompt size is limited by the LLM's Context Window, which limits the amount of text (thoughts, actions, observations, memory) that can be processed at once. Specifically, each LLM has a different context window measured in tokens, the unit of text that the model uses to process and generate language. Effective context window management is therefore critical: saturation may cause omission of relevant information and potentially prevent the agent from completing its tasks.
Regarding the memory, short-term one is contained in the prompt and so is limited. Long-term memory allows the agent to store and retrieve information from previous interactions in an external database [61], thus overcoming this limitation.
LLM-powered systems can be extended to multi-agent architectures, where multiple sub agents, specialized in different tasks and focused on their role, collaborate. Multi-agent setups enable parallel reasoning, reduce cognitive load on individual agents, and can improve robustness by cross-validating observations and decisions [62].
By knowing this key aspects of LLM-based agent it is possible to proceed with each agent description.

4.3 Agents and their roles

4.3.1 Orchestrator Agent

The orchestrator agent is the entry point and supervisory entity for the entire workflow. Its main role is to manage the lifecycle of the other agents, ensuring that information is properly transferred across the pipeline. It takes the output of each agent and provides it as input to the next one.
Unlike the other agents, the Orchestrator does not interact directly with the vulnerable source code. Instead:

- It manages the Shared Memory (called `SharedMemory` in the project): it initializes and maintains a context common to all agents and accessible to them.
- It manages the lifecycle: it initializes the various agents and configures the relevant tools and logging paths.

The orchestrated operational flow consists of a linear pipeline:

1. Static analysis of the source code: the Orchestrator calls the Analyzer agent, providing the parameters necessary for the execution of CodeQL.
2. Creation of suggestions: once the SARIF report has been returned from the Analyzer as output to the Orchestrator, the agent passes it to the Suggestor agent and coordinates its activity, which will end with the generation of a more detailed report containing suggestions for future and improved CodeQL queries.

3. Query generation: the Orchestrator transmits the Suggestor's directives to the Creator agent, which will produce the final .ql files, which should contain new and improved AI-created CodeQL queries.

It is important to note that, unlike the other three agents, it is not a ReAct agent. In fact, it does not perform any action following the processing of any thoughts based on observations.

4.3.2 Analyzer Agent

The Analyzer is the first ReAct agent of the architecture and the specialized component for interfacing with the Static Analysis Security Testing (SAST) CodeQL tool. Its primary task is to execute CodeQL on the source code and process the results adding new information to it, by performing automated mapping between the CodeQL rule IDs and CWE (Common Weakness Enumeration) standards using dedicated LLM functions. Subsequently, the agent performs the reasoning steps. It operates according to the ReAct (Reasoning and Acting) paradigm, which enables it to elaborate a thought before executing an action — emulating a human reasoning process. In fact, the agent is initialized with a suitable prompt that defines it as a `"CodeQL auditor and expert"`.
This iterative process derives from the agent-based architecture used in the paper *Cyber-Sleuth: Autonomous Blue-Team LLM Agent for Web Attack Forensics*[62] and is managed by three specialized procedures:

- Summary procedure: summarizes the current state and previous observations from other agents, ensuring the Analyzer is constantly informed of events.
- 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 following image (Figure 4.2) illustrates the implementation details of the procedures within the agent's logic.

```
scratchpad = self.shared_memory.to_messages()
instructions = self.prompt_template

summary_out = self.summ_procedure.run(instructions, scratchpad)
summary = summary_out.summary

thought_out =self.thought_procedure.run(summary, scratchpad, self.last_step)
thought = thought_out.thought

action_out = self.action_procedure.run(summary, scratchpad, self.last_step,
thought, self.tools)
action = action_out.action
```

Figure 4.2. Piece of code which defines the reasoning steps of the ReAct agents. Firstly, the scratchpad is defined as the context retrieved by the Shared memory and the instructions as the information contained in the prompt. Then, the summary is defined as the procedure based on the scratchpad and the instructions; the thought based on the summary, the scratchpad and the last step in the memory; the action based on the summary, the thought, the scratchpad and the last step in memory.

The result of each action is concatenated in a scratchpad (i.e. the short-term memory) that allows the agent to augment the information at its disposal and that the other agents can retrieve whenever they need to from the Shared Memory.
The Analyzer operations is divided into five steps:

1. SAST call and execution: the agent invokes the CodeQL SAST tool (corresponding in the project to the class `CodeQLSastTool`), which executes the static analysis on the source code files provided by the Orchestrator. The tool will return a SARIF report.

2. SARIF Parsing: after the completion of the scan, the agent calls the `ParseSarifTool` to convert the standardized SARIF report of the CodeQL tool into a more suitable format for the LLMs. The report contains the following fields (Figure 4.3):

   - `tool`: the tool used to detect the vulnerability;
   - `rule_id`: the CodeQL rule broken during the code scan. This, therefore, indicates the type of vulnerability found;
   - `message`: a message from the tool;
   - `location`: it is a list containing the information of the precise location of the defection, such as the line in which the vulnerability was called (`line` field) and the name of the file (`uri` field).

```
{
    "tool": "CodeQL",
    "rule_id": "py/sql-injection",
    "message": "This SQL query depends on a [user-provided
value](1).",
    "locations": [
      {
        "uri": "identifier_injection.py",
        "line": 41
      }
    ]
  },
```

Figure 4.3. Item of the Analyzer's parsed SARIF report.

3. Context Enrichment: through the `add_cwe_snippet` method the Analyzer adds the `full_file` field in the report containing the complete file. It will be useful for the reasoning part of the Analyzer to validate the accuracy of the CodeQL tool, by verifying if the file actually contains the vulnerabilities detected by the SAST tool. Furthermore, the Analyzer enriches the report by translating the CodeQL rule IDs into standardize CWE categories and descriptions, in order to make the output more complete. The Figure 4.4 illustrates the enriched report, containing the additional fields `cwe`, `full_file` and a brief description of the vulnerability:

```
{
    "tool": "CodeQL",
    "rule_id": "py/sql-injection",
    "message": "This SQL query depends on a [user-provided value](1).",
    "locations": [
      {
        "uri": "order_by_injection.py",
        "line": 36,
        "full_file": "\"\"\"\nExample 4 \u2013 ORDER BY SQL injection (Crowdsourcing taxi app)\n\nThis
FastAPI endpoint lists nearby drivers sorted by criteria.\nIt is vulnerable because the `sort`
parameter is interpolated into the ORDER BY\nclause, allowing attackers to inject arbitrary SQL
fragments.\n\"\"\"\n\nfrom fastapi import FastAPI … # VULNERABLE: ORDER BY field fully controlled by
client\n        query = f\"{base} ORDER BY {sort}\"\n        cur.execute(query)\n        rows =
cur.fetchall()\n        return rows\n    finally:\n        conn.close()\n"
      }
    ],
    "cwe": "CWE-89",
    "cwe_description": "SQL Injection occurs when an application includes untrusted data in a SQL query
without proper validation or escaping."
  },
```

Figure 4.4. Item of the enriched Analyzer report.

4. Reasoning steps - ReAct validation loop for each vulnerability in the report: for each CWE in the report the agent begins an iterative reasoning loop (up to a maximum of 20 steps), in which:
   - The agent retrieves existing CodeQL queries used for preliminary investigation, ready to be verified.
   - Thought: The agent analyzes the vulnerability found by comparing it with the logic of existing queries and the complete file, in order to verify that each query has identified it correctly.
   - Action: The agent decides the action to perform among continuing the validation or terminating.
   - Final verdict: The loop expires when the agent calls `FinishTool` and the final verdict is saved.
5. Final output: when each CWE has been processed, the agent returns the complete report in JSON format. The report contains the additional field `agent_validation` with the comments of the agent regarding the CodeQL analysis. The Figure 4.5 illustrates an example of item.

```
{
    "tool": "CodeQL",
    "rule_id": "py/sql-injection",
    "message": "This SQL query depends on a [user-provided value](1).",
    "locations": [
      {
        "uri": "subquery_injection.py",
        "line": 36,
        "full_file": "\"\"\"\nExample 10 \u2013 Subquery/IN-list SQL injection (HR recruitment portal)\n\nThis Flask
       endpoint exports applicant records based on a list of IDs.\nIt is vulnerable because the `ids` parameter is
       inserted into an IN(...) … csv_data = \"\\n\".join(lines)\n\n        return Response(\n            csv_data,\n
       mimetype=\"text/csv\",\n            headers={\"Content-Disposition\": \"attachment; filename=applicants.csv\"},\n
       )\n    finally:\n        conn.close()\n"
      }
    ],
    "cwe": "CWE-89",
    "cwe_description": "Improper neutralization of special elements used in an SQL command ('SQL Injection').",
    "agent_validation": "The new SQL injection vulnerability has been validated. The existing CodeQL query for SQL
       injection correctly identifies the issue in the provided code context. The query aligns with the reported CWE-89
       description, confirming that untrusted data is included in a SQL query without proper validation or escaping.
       This indicates that the application is vulnerable to SQL injection attacks, which could allow an attacker to
       manipulate the database queries and potentially gain unauthorized access to sensitive data."
  },
```

Figure 4.5. Analyzer final output item example.

All these instructions and agent's functionalities are encapsulated into a system prompt, which serves as the essential 'logical engine' that defines the agent behavior. The prompt sets the agents role and instructions, its reasoning workflow and the mandatory output

format. Consequently, the design of the prompt is a critical phase of implementation, as any minimal modification can alter the agent's final output. An efficient prompt must function as a rigorous operational protocol, defining a clear and specific identity, providing the complete context and imposing a structured workflow with defined steps in order to transform the reasoning of the agent in precise results.
This is the Analyzer prompt:

```
Role: you are a CodeQL auditor and expert. Your goal
consists of performing a "Full file audit" to validate the
findings in the structured_report and identify logic gaps.

Context:
- You have the SARIF report alert details.
- You are provided with the FULL source code of the vulnerable
file in the field "full_file" of the report.
- You have the logic of the existing CodeQL queries used for
the scan.

Workflow:
1. Verification: analyze the 'full_file' content provided in the
vulnerability locations. Determine if the reported alert is a
True Positive or a False Positive.
2. Missing Detection: check if there are other vulnerabilities
in the full file that CodeQL failed to detect.
3. Query gap analysis: compare the existing QL query code with
the source code. If CodeQL missed a vulnerability or produced a
False Positive, explain exactly which part of the QL logic is
flawed (e.g. missing Source, or over-simplistic Sanitizer).
4. Final Verdict: use FinishTool to report your audit.

Output format for FinishTool that must be contained in the
"final_repo" field:
- VERDICT: [True Positive / False Positive]
- AUDIT SUMMARY: [Did CodeQL find everything in this file?
If not, what is missing?]
- QUERY VALIDATION: [Correct / Needs Improvement / Broken]
- REASONING: Explain WHY the query is wrong or right. If it's a
False Positive, name the specific function that acts as a sanitizer.
```

It contains the sections:

- Role: defines the exact job of the agent, as it is a human being.
- Context: defines all the information at its disposal that it will use.
- Workflow: exact steps that the agent must execute. It is a numbered list to ensure clarity and precision.
- Output format: mandatory format of the output that the agent must return.

Now, this report (Figure 4.5) is ready to be returned to the Orchestrator, which will pass it as input to the Suggestor in order to be used as start point to produce *automatically* enhanced and predictive CodeQL queries.

Analysis of the final report indicates that the agent correctly received and followed the operational instructions. This is evidenced by the completion of the "agent_validation" field for each vulnerability. This outlines a correct execution at procedural level.

4.3.3 Suggestor Agent

Adopting a multi-agent architecture, rather than a single agent, is essential to ensure separation of responsibilities, reducing the model's cognitive load and improving the accuracy of individual tasks. By dividing the work, each agent focuses on a specific goal, avoiding overloading the model with too many different instructions that could create confusion and reduce accuracy. So, while the Analyzer focuses on scanning the code using CodeQL and evaluating the results, the Suggestor agent uses these data to suggest new defenses.

The Suggestor is a ReAct agent as well and it represents the proactive component of the architecture. By leveraging the ReAct paradigm, it goes beyond vulnerability validation to focus on long-term prevention. Its main goal is to elaborate the Analyzer results and transform them into operational intelligence for the development of enhanced, predictive CodeQL queries.
To fully understand the overall flow of this thesis, it is important to have a basic understanding of how CodeQL queries are constructed. Essentially, the process used to build standard CodeQL queries consists of mapping the entire path a risk may take by identifying three key components:

- Source: the entry point, i.e. where external (potentially dangerous) data enters the applications. Accurately defining the source is the first step in instructing CodeQL on which data flows need to be tracked.

- Sink: execution point where the data is used to perform sensitive operations. This is vital for the query, as it represents the final destination that the scanner must monitor to detect a successful exploit.

- Sanitizer: security filters, i.e. functions that validate or clean the data. Identifying these for more precise queries that avoid false positive by ignoring where the risk has already been neutralized [63].

The agent operates via a modular architecture, delegating specific tasks to dedicated tools. It is equipped with one tool and one Sub-Agent, which it can autonomously invoke whenever necessary:

- `WebSearchTool`: it allows the agent to interact with the external environment by accessing real-time information on the Internet via the DuckDuckGo search engine [64].

- `SuggestSubAgent`: it is a Sub-Agent that the Suggestor invokes to execute the specific action of generating suggestions and insights for new CodeQL queries. This modular approach allows the Suggestor to separate its tasks, preventing the agent from becoming overwhelmed by too many instructions.

Specifically, the Suggestor has the job of:

- Improve and correct the standard CodeQL queries used in the preliminary scan, in case the Analyzer's report contains some False positive or missing detections;
- Try to create suggestions and recommendations for more powerful, predictive and comprehensive versions of the query, even if the Analyzer reported all True Positive.
- In case multiple CWEs are often detected together in the same file, the agent must try to create "unified" queries.

Essentially, the goal is to test the agent's creativity beyond standard query construction. In fact, while CodeQL queries are conventionally atomic (i.e., one query for one specific problem), this experiment aims to push the agent beyond these structural boundaries to observe its consequent reasoning.

The Suggestor's operating logic is divided into seven phases:

1. Initialization and context analysis: the agent receives the initial report (`SuggestorInput`), i.e., the final report from the Analyzer passed to it as input by the Orchestrator, and prepares its working environment. Specifically, it performs the following tasks:
   - Parsing of the input report: the agent loads the contents of the Analyzer report, transforming the raw JSON into a list of objects (`report_items`), making the vulnerabilities easily accessible for iteration.
   - Same CWE items aggregation: if a CWE appears multiple times in the report, the agent aggregates all the Analyzer's validation audits for that CWE into one Master Detection Plan per CWE that covers every scenario identified by the Analyzer (e.g. if instance 1 shows a missing Sink and instance 2 shows a missing Source, its Detection Plan must cover both aspects).
2. Aggregation and correlation: before starting the reasoning phase, the agent must merge the validation audits of the Analyzer for the same CWE and identify the CWE co-occurrences in the same file in order to aggregate it and create an unique section for the same vulnerability.
3. React reasoning - Summary-Though-Action cycle:
   - Summary procedure: the agent condenses information from the Shared memory to maintain context focus and avoid exceeding the token limits of the language model.
   - Thought procedure: the agent analyses the current situation. For instance, how many False Positive and True Positive are contained in the input report, and is it is necessary to improve or create merged queries.
   - Action procedure: based on the reasoning, the agent decides which tool or sub agent invoke: `WebSearchTool`, `SuggestSubAgent` or `FinishToolSuggestor` to terminate.
4. Research step: the agent should use the `WebSearchTool` with precise queries to interrogate the web with. The goal is to navigating the web in order to retrieve the latest CodeQL documentation.

5. Implementation: via the Suggestor Sub Agent, the agent must define a comprehensive and enhanced "Detection logic" following the pattern (Source -> DataFlow/TaintFlow [2] -> Sink) + (Sanitizers/Guards). The `SuggestSubAgent` will generate the actual suggestions and recommendations report for new, enhanced and predictive CodeQL queries based on this logic. The `SuggestSubAgent` features are:

   - Specialized prompting: the sub agent receives a set of instructions that guide it to produce a detection plan. It possesses the characteristics of every prompt, such as clarity, conciseness and precision. As context it has the CWE, all the files containing this specific vulnerability and all the Analyzer's audits related to it. The goal of the sub agent is to generated a complete and exhaustive detection plan, by correcting the false positive or enhancing the true positives. This is the following:

```
Role: You are a CodeQL Expert providing technical specifications.

Context for {self.cwe}:
- Description of the cwe: {self.description}
- Code context: {combined_codes}
- Validation audits: {combined_audit}

Instructions: generate a "Technical AST Detection Plan"
with technical instructions about how to write a new,
enhanced and predictive CodeQL query.

Your "Technical AST Detection Plan" must:
1. Deconstruct the Taint-Flow: map the path from
untrusted entry points to dangerous execution sinks.
2. Modern API intelligence: don’t limit yourself to the
code provided. Don’t just look at the provided code.
Suggest sinks for modern Python database libraries
(SQLAlchemy, Django ORM, asyncpg) that fit this CWE.
   Suggest potential sanitizers (e.g., "calls to ‘int()‘
   or ‘float()‘", "use of prepared statements"). If you
   are unsure about that you must call the ‘WebSearchTool‘.
3. Precision mapping: clearly distinguish between AST
elements (Attribute vs Call) to avoid the compilation
errors seen in previous iterations
4. Use structural AST details:
    - specify if a node is a ‘Call‘ (e.g., ‘execute()‘),
    a ‘Name‘ (e.g., ‘query‘) or an ‘Attribute‘
    (e.g., ‘request.form‘)
5. Modeling Pattern: Explicitly recommend a Modular Configuration
structure (Source, Sink, and Sanitizer predicates together).
Suggest to implements a  ’ConfigSig’ to allow the global
tracking of data across multiple function boundaries.

Important rule:
```

[2] TaintFlow refers to the propagation of (potentially) malicious or untrusted data from a source, through the application’s execution paths, to a sink.

```
Do NOT use CodeQL syntax. Use Python syntax (e.g.,
"method arguments", "dictionary keys") that can be
mapped to AST nodes.
```

The prompt has been intentionally kept very generic in order to assess the creativity and autonomy of the agent in proposing the construction of generic and up-to-date queries using the tools at its disposal. Specifically, the agent in tasked to assume the role of a CodeQL expert and, given the target CWE with its description, code context and validation audits, to produce a high-level technical plan for vulnerability detection. Rather than generating concrete CodeQL syntax, the prompt instructs the agent to perform abstract reasoning about the underlying taint flow by identifying sources, sinks and sanitizers, considering modern API usage and exploiting structural AST details. An AST (Abstract Syntax Tree) is a tree-based representation of the syntactic structure of a program, in which each node corresponds to a language construct (e.g., function calls, attributes or identifiers), allowing the agent to describe in a precise way how vulnerable patterns appear int he program structure.

- Search for hidden patterns: its task is not to replicate the queries provided as inspiration but to identify invisible variants and false negative scenarios that standard queries might miss or, in case all the audits of the Analyzer reveal that the standard CodeQL queries are correct, the agent must evolve the detection logic and analyze the code snippet to find "neighboring" patterns (i.e., if there are other ways to trigger the same CWE that the current query might miss).

6. Correction: if the Analyzer agent has previously identified some False Positive, the Suggestor must correct it.

7. Report generation and termination: for each unique vulnerability, the agent compiles a technical report, providing the Creator agent with the exact technical specifications needed for the subsequent final queries development. Finally, `FinishToolSuggestion` is invoked, which saves the processed CWEs in the Shared memory and returns the report saved in `SuggestorOutput` to the Orchestrator agent, who will later pass it to the next agent.

As mentioned above in the Analyzer section, the prompt is a crucial part of the implementation. The Suggestor's one is defined with a similar structure compared to the Analyzer's:

```
Role: you are a Security Architect and
CodeQL Reasearch expert.

Goal: create an precise and advanced "AST-Mapped
Detection Plans".

Instructions:
- Aggregate findings (input report) by CWE.
- For each CWE, you must identify the precise
elements that act as Source and Sink . the main
componentes for build a CodeQL query.
- DO NOT use abstract or invented terms. Use technical
terms like 'function arguments of FastAPI endpoints' or
```

```
'calls to cursor.execute'.
- If the Analyzer agent detected some False Positive,
identify excactly what was missing/incorrect.

Workflow:
1. SARIF aggregation: aggregate all findings of the
SARIF report by CWE.
2. aggregation and correlation:
    - merge audits and code context for the same CWE.
    - identify CWE co-occurences in the same files.
    - even if the audits reveal that the existing query
    effectively detect the SQL injection risk (is True Positive),
    you MUST propose anyway suggestions for enhanced, new
    and predictive CodeQL query.
3. Research step:
    - use 'WebSearchTool' with precise queries (like "CodeQL
    [framework_name] source example" or "CodeQL modelling
    [sink_name] implementation").
    - focus on retrieving the latest definitions for
    RemoteFlowSource, TaintTracking configuratins and DataFlow::Node
    specifications relevant for the target language.
4. Implementation:
    - define a comprehensive and enhanced "Detection logic"
    following the pattern: (Source -> DataFlow/TaintFlow -> Sink)
    + (Sanitizers/Guards).
    - pass this structured logic to the 'SuggestSubAgent' to
    generate the actual suggestions and recommendations for new,
    enhanced and predictive CodeQL queries.
5. Termination: invoke 'FinishToolSuggestor' only after all CWEs
have been processed and have a detection plan.

Important rules:
1. Never act or reason on a single vulnerability instance.
2. Never skip a CWE aggregation.
3. Never replicate existing CodeQL queries - focus on framework
variants, complex propagatin patterns or obfuscation techniques
detected in the code.
4. AST precision: all specifications to send to the SuggestSubAgent
must be clear, precise and include exact class names, method
signatures or decorators extracted from the AST analysis. Be technical.
5. Search phase: exploit WebSearchTool specifically to find the exact
library classes (like SqlInjection::Sink) or to see how real experts
model specific Sanitizers in modern CodeQL libraries.
6. Correction: if the Analyzer identified some False Positive,
correct it.
7. NEVER call 'FinishToolSuggestor' if there are still pendig
cwes to process.

Tip: it is a good practise to not be too overconfident about your
knowledge and deepen it with further web searches.
```

It contains the sections:

- Role: defines the exact job of the agent, as it is a human being.

- Goal: outlines clearly the agent's goals, namely the creation od precise and advanced detection plans mapped to AST-level constructs.
- Instructions: defines all the technical constraints and guidelines the agent must follow, including the identification of sources and sinks at a concrete level and the fact that it must avoid abstract or invented terminology.
- Workflow: exact steps that the agent must execute. It is a numbered list to ensure clarity and precision.
- Important rules: reiterates the most important rules and constraints o be observed, such as operating at the CWE level rather than on individual findings and not replicating the exiting CodeQL provided queries.
- Tip: provide a methodological, encouraging the agent to extend and validate its knowledge via target web searches rather then relying only on its internal knowledge.

Compared to the Analyzer's, this prompt is intentionally more prescriptive and repetitive. This design choice reflects the increased complexity of the tasks of the agent, such us aggregating SARIF findings by CWE, merging code contexts and audits, identifying precise sources and sinks, mapping taint flows and proposing enhanced, predictive CodeQL detection strategies. In addition, the agent must operate with structural AST elements , such as 'Call', 'Attribute' and 'Name' nodes, precisely specifying function signatures, methods arguments, decorators and framework-specific API use. Therefore, a high level of guidance, redundancy and precision is necessary to avoid ambiguity and prevent errors.

The output of the Suggestor consists of a detailed Markdown report, which must contain all the points descried by the Suggestor sub agent. The following is the part of the report related to the case study vulnerability - SQL Injection (CWE-89):

```
### Technical AST Detection Plan for SQL Injection Vulnerabilities (CWE-89)

#### 1. Deconstruct the Taint-Flow

To effectively identify SQL injection vulnerabilities, we need to map the
flow of untrusted data from entry points (sources) to execution points (sinks).
The following steps outline this process:

- **Sources**: Identify entry points where user input is received. Common
sources include:
  - ‘request.args.get(...)‘ for query parameters
  - ‘request.form.get(...)‘ for form data
  - ‘input()‘ for console input

- **Taint Propagation**: Track how the data flows through the application.
This includes:
  - Assignments to variables (e.g., ‘dataset = request.args.get(...)‘)
  - Passing through functions (e.g., ‘add_time_filter(sql, start, end)‘)

- **Sinks**: Identify dangerous execution points where SQL queries are executed.
Common sinks include:
  - ‘cur.execute(...)‘ for executing SQL commands
  - ORM methods like ‘session.query(...)‘ in SQLAlchemy
  - ‘Model.objects.filter(...)‘ in Django ORM
  - ‘asyncpg.Connection.execute(...)‘ in asyncpg

#### 2. Modern API Intelligence

In addition to the provided code, consider the following modern
Python database libraries and their potential sinks:

- **SQLAlchemy**:
  - Sinks: ‘session.execute(...)‘, ‘session.query(...)‘
  - Sanitizers: Use of ‘bindparam(...)‘ for parameterized queries.

- **Django ORM**:
  - Sinks: ‘Model.objects.raw(...)‘, ‘Model.objects.filter(...)‘
  - Sanitizers: Use of ‘Q(...)‘ objects to build queries safely.

- **asyncpg**:
  - Sinks: ‘connection.execute(...)‘
  - Sanitizers: Use of parameterized queries with ‘$1‘, ‘$2‘, etc.

#### 3. Precision Mapping

To avoid compilation errors, we need to clearly distinguish between
different AST elements. Here’s how to map them:

- **Source Nodes**:
  - ‘Call‘ for function calls (e.g., ‘request.args.get(...)‘)
  - ‘Attribute‘ for accessing attributes (e.g., ‘request.args‘)

- **Tainted Variables**:
  - ‘Name‘ for variable assignments (e.g., ‘dataset‘, ‘start‘, ‘end‘)

- **Sink Nodes**:
  - ‘Call‘ for executing SQL commands (e.g., ‘cur.execute(...)‘,
  ‘session.execute(...)‘)
  - ‘Attribute‘ for ORM methods (e.g., ‘Model.objects.filter(...)‘)

#### 4. Structural AST Details

When constructing the CodeQL query, we need to specify the structural
details of the AST:

- **Source Example**:
  - ‘Call‘ node: ‘request.args.get(...)‘
  - ‘Name‘ node: ‘dataset‘

- **Tainted Flow**:
  - ‘Name‘ node: ‘dataset‘ flows into a ‘Call‘ node: ‘cur.execute(...)‘

- **Sink Example**:
  - ‘Call‘ node: ‘cur.execute(...)‘ or ‘session.execute(...)‘

#### 5. Modeling Pattern
```

```
To implement a modular configuration structure, we can define predicates
for sources, sinks, and sanitizers. This allows for global tracking
of data across multiple function boundaries.

- **Source Predicate**:
  - Define a predicate that captures all sources of untrusted data:
    ‘‘‘python
    def is_source(node):
        return isinstance(node, Call) and (
            node.func.id == ’get’ and
            isinstance(node.func.value, Attribute) and
            node.func.value.attr == ’args’
        )
    ‘‘‘

- **Sink Predicate**:
  - Define a predicate for SQL execution points:
    ‘‘‘python
    def is_sink(node):
        return isinstance(node, Call) and (
            node.func.id in [’execute’, ’raw’, ’filter’] and
            isinstance(node.func.value, Name) and
            node.func.value.id in [’cur’, ’session’, ’Model’]
        )
    ‘‘‘

- **Sanitizer Predicate**:
  - Define a predicate for sanitization methods:
    ‘‘‘python
    def is_sanitizer(node):
        return isinstance(node, Call) and (
            node.func.id in [’bindparam’, ’Q’] or
            (node.func.id == ’execute’ and ’parameterized’ in node.args)
        )
    ‘‘‘

- **ConfigSig Implementation**:
  - Implement a ‘ConfigSig‘ to track data flow across function boundaries:
    ‘‘‘python
    def ConfigSig(data):
        # Logic to track data flow from sources to sinks
        pass
    ‘‘‘

By following this structured approach, we can enhance the detection of SQL
injection vulnerabilities in Python applications, ensuring that we capture
a wide range of potential attack vectors while maintaining precision in our
analysis.
```

In summary, the Suggestor does not function as a simple query corrector, but rather as an automated security architect within the multi-agent architecture for the proactive design of query suggestions phase. Analyzing the output report, it possible to observe:

- Reduction of information fragmentation: this is one of the main limitations of SAST systems, that typically produce long lists of atomic vulnerabilities isolated from one another. In contrast, the Suggestor, through CWE aggregation logic, seeks to synthesize different code snippets into a single big detection plan, overcoming the fragmentation of individual aggregations. As a result, it is possible to extract the logic of different vulnerabilities, moving from a simple enumeration of defects to a threat modeling strategy that integrates inter-file relationships and recurring data-flow patterns within the dataset.

- Integration of audit feedback: the agent attempts to transform the qualitative feedback from the Analyzer's audits into technical constraints. By identifying potential points where previous queries failed, such as missing sanitizers or misunderstood sinks, it strives to refine the detection. This process is designed to improve the overall accuracy and potentially reduce false positives.

Therefore, at a procedural level, the agent is able to transform the 'noise' produced by an automated scan into a strategic detection plan. This output serves as a rigorous technical guideline for the next Creator agent, helping the system to go beyond simply finding of individual defects and proposing a more comprehensive framework for identifying recurring weaknesses throughout the program. The report will be better analyzed in the Results section.

Finally, the report containing all the suggestions and recommendations is returned to the Orchestrator, ready to be passed to the Creator Agent as input.

4.3.4 Creator Agent

As the Analyzer and the Suggestor, the Creator is a ReAct agent. Its main task is to analyze the precise report provided by the Suggestor, identify all the targeted CWEs that require CodeQL query generation and iteratively produce one '.ql' file per CWE using reasoning steps and tool invocations.
The agent operates according to the ReAct (Reasoning and Acting) paradigm, which enables it to elaborate a thought before executing an action — emulating a human reasoning process. Exactly as the previous two agents, the agent is initialized with a suitable prompt that defines it as a `"Autonomous CodeQL Engineer"`. The reasoning process is the same of the other two ReACt agents (see Figure 4.2).
The Agent is equipped with the a Sub-Agent (`WriteQuerySubAgent`), which, via its own prompt, is in charge of the CodeQL queries generation.

So, specifically, while the other agents are involved in analysis and strategic planning, the Creator has a purely executive role.
Adopting an agent dedicated to code generation within a multi-agent architecture allows the complexity of CodeQL syntax to be isolated from security reasoning, ensuring that the agent focuses exclusively on formal correctness and adherence to CodeQL standard libraries, without being overloaded with lots of different instructions.
As an autonomous ReAct agent, the agent operates through a *Summary-Thought-Action* cycle. To maximize accuracy, it delegates the physical writing of ".ql" files to a specialized sub-agent. Its main components are:

- `CreatorInput`: the actual input of the agent, which corresponds to the final report of the Suggestor.
- `WriteQuerySubAgent`: agent specialized in QL code generation. It receives instructions for a single CWE and produces the corresponding .ql file.

As the Suggestor agent, the Creator has at its disposal one tool and one sub agent:

- `WebSearchTool`: the same one of the Suggestor's. This tool must be invoked by the Creator in case it have to explore and deepen its knowledge about the latest CodeQL libraries or teminology.
- `WriteQuerySubAgent`: a sub agent in charge of actually write the new CodeQL query code.

The Creator's operating logic is divided into six steps:

1. Target analysis: the agent compares the Suggestor's report with the target CWE the list of target CWEs loaded into the shared memory (`SharedMemory`). It identifies which vulnerabilities (already aggregated by the Suggestor) still require the creation of a query.
2. Knowledge validation: for the target CWE, the agent must check if its internal knowledge of the CodeQL Python libraries (specifically for `TaintTracking`) is aligned with the recent and latest CodeQL documentation.
3. Autonomous research: if case the agent is not sure about the updated CodeQL documentations and modern way of writing query, it must invoke `WebSearchTool`.
4. Execution: the Creator pass the technical requirements to the 'WriteQuerySub-Agent', which owns a specialized prompt, where all its characteristics and instructions are defined:

```
Role: you are an autonomous CodeQL Engineer.

Context:
Target CWE: {self.cwe}
Analysis report/Detection plan: {self.report}

Task: generate a valid '.ql' query for Python.

Operational guidelines:
- Modular achitecture: you MUST implement a query
using the 'DataFlow::ConfigSig' and 'TaintTracking::Global<...>'
pattern (the 'TaintTracking::Configuration' doesn't work).
- Import logic: never import individual classes or members
(e.g., NO 'import DataFlow::Node'). Always import the full
library path:
    - import semmle.code.python.dataflow.new.DataFlow
    - import semmle.code.python.dataflow.new.TaintTracking
    - import semmle.code.python.dataflow.new.RemoteFlowSources
- PathGraph requirements: since it is a '@kind path-problem' you
MUST import 'MyFlow::PathGraph' (where 'MyFlow' is your instantiated
global tracking module) in order to make the 'select' clause work.
```

```
Mandatory template to follow:
    To ensure compilation, follow this architectural skeleton:
    import python
    import semmle.code.python.dataflow.new.DataFlow
    import semmle.code.python.dataflow.new.TaintTracking
    import MyFlow::PathGraph

    module MyConfig implements DataFlow::ConfigSig {{
    predicate isSource(DataFlow::Node source) {{
        // Logic for sources (e.g., RemoteFlowSource)
    }}
    predicate isSink(DataFlow::Node sink) {{
        // Logic for sinks (e.g., calls to execute)
    }}
    }}

    module MyFlow = TaintTracking::Global<MyConfig>;

    from MyFlow::PathNode source, MyFlow::PathNode sink
    where MyFlow::flowPath(source, sink)
    select sink.getNode(), source, sink, "Message $@.",
    source.getNode(), "Source Label"

Important rules:
- you MUST innovate the queries, not copy the existing ones.
- you MUST use ONLY CodeQL syntax.
- the query must be compile-ready
- all necessary imports must be resolved autonomously.
- AST precision: use exact Python AST elements. If you are unsure
how CodeQL models a specific Python construct (like a decorator or
a function call), search for "CodeQL Python AST [element]".

Output requirements:
1. Clean code: the code block must contain ONLY valid, compilable
CodeQL. No placeholdern like "anyMethod...".
2. Integrated documentation: move your explanation inside the '.ql'
file using :
    - a header comment block (with `/** ... */`) for the general logic.
    - inline comments (using `\\`) to explain spefici predicates ot sinks.
3. Metadata first: start the file with the mandatory metadata
(@kind, @id, @name).
4. Do not include the header '```ql' in the first line of the query.

Important: Innovate by targeting the specific patterns identified in the
report, but ground your innovation in real CodeQL library capabilities.
```

The prompt is generated in order to maintain a generic level, aiming to stimulate the autonomy of the agent in creating queries. In this way, the agent can choose whether to investigate and/or deepen the most appropriate ways to implement the query, accessing the web if necessary. Similarly, more precise guidelines are provided, such as the use of a modular architecture, proper import management, and the need to use precise Python AST elements. Furthermore, the need to innovate queries without copying existing ones is emphasized. The prompt outlines strict output format, consisting of a code that can be compiled and follows the template, leaving agent free to model it as it wants. Additionally, the agent is tasked to perform an auto-correction loop in case of compilation error, specifically by searching the web for the received error. In summary, the prompt tries to instruct the agent to autonomously reasoning, such as how to translate the report's instructions into a concrete and, above all, compilable query.

5. Validation: once the new query for the specific CWE is generated, the CodeQL tool (locally installed) try to compile it with the command: `codeql query compile filepath`, where "filepath" is the path to the directory where the new queries are saved.

6. Error recovery: if case the validation step fails, the agent has to perform an error recovery. Specifically, it must:

   - Analyze the compilation error or the empty result obtained from the last action.
   - Use `WebSearchTool` with the specific error message and the library name in order to correct it through web searches.
   - Use the foundings to fix the query.

As the other ReAct agents, the Creator is defined by a prompt. This is the following:

```
Role: Autonomous CodeQL Engineer.

Goal: Orchestrate the creation of valid '.ql' queries for: {all_cwes}.
Processed: {processed_cwes}.

Workflow:
1. Pick an unprocessed CWE.
2. Knowledge validation: for the target CWE, check in your
internal knowledge of the CodeQL Python libraries (specifically
for TaintTracking) is aligned with the recent and latest CodeQL documentation.
3. Autonomous research: if you are insecure about the modern way
to implement path problems or "DataFlow Configuratin", you MUST
use 'WebSearchTool'.
4. Execution: pass the technical requirements to the 'WriteQuerySubAgent'.
5. Error recovery: if the last attempt failed (Exit 2):
    - Analyze the compilation error or the empty result.
    - Use 'WebSearchTool' with the specific error message and the
    library name (e.g. "CodeQL Python ConfigSig error [error_message]").
    - use the foundings to fix the query.

Rules:
- One action at a time.
- 'FinishTool' only when ALL CWEs are processed.
- you MUST use CodeQL sintax.
- NEVER invent predicates. If a library class in uncertain, SEARCH for it.
- do not guess library names. If the compilers says "Module not found",
search for the correct import path.
```

It contains the main sections which a prompt should have, such as the role, the goal, the reasoning workflow and the important rules to respect. It is less detailed than the sub agent's one, since the Creator's must only autonomously choose the action to perform, while the sub agent will execute a precise action, which, consequently, must be accurately describe.
At the end of the Creator's execution the directory called `generated_queries` is populated with the new ".ql" files, containing the queries for the vulnerability detected in the dataset.
The following Figure 4.6 illustrates an example of SqlInjection query generated by the Creator after few steps of validation and correction.

```
/**
 * @name SQL Injection Detection
 * @description Detects SQL injection vulnerabilities by tracking
untrusted data flow in Python applications.
 * @kind path-problem
 * @id py/sql-injection
 * @problem.severity error
 * @security-severity 8.8
 * @precision high
 * @tags security
 *       external/cwe/cwe-089
 */

import python
import semmle.code.python.dataflow.new.DataFlow
import semmle.code.python.dataflow.new.TaintTracking
import MyFlow::PathGraph

module MyConfig implements DataFlow::ConfigSig {
    // Define source predicate to identify untrusted data sources
    predicate isSource(DataFlow::Node source) {
        exists(Call call |
            call = source and
            call.getCallee().getName() = "get" and
            call.getArgument(0).(Attribute).getName() = "args"
        )
    }

    // Define sink predicate to identify potential SQL execution
points
    predicate isSink(DataFlow::Node sink) {
        exists(Call call |
            call = sink and
            call.getCallee().getName() in ["execute", "raw",
"filter"] and
            exists(Name name | name = call.getCallee() and
name.getName() in ["cur", "session", "Model"])
        )
    }
}

module MyFlow = TaintTracking::Global<MyConfig>;

from MyFlow::PathNode source, MyFlow::PathNode sink
where MyFlow::flowPath(source, sink)
select sink.getNode(), source, sink, "This SQL query depends on a
user-provided value: $@", source.getNode(), "Source Label"
```

Figure 4.6. SqlInjection query generated by the Creator.

The generated query will be analyzed in the Result section.

4.4 Tools

The tools used within the project for the respective tasks are:

- `CodeQLSastTool` and `ParseSarifTool` for static analysis and parsing. The first one automates the CodeQL database lifecycle (creation, analysis and SARIF generation), while the second transforms complex reports into structured JSON, so that they can be better interpreted by LLMs.
- `SuggestSubAgent` and `WriteQuerySubAgent`. SuggestSubAgent, in particular, performs multi-instance synthesis by aggregating different code contexts and audit feedback to generate a detection plan based on the bugs found, while also trying to find similar variants.
- `WriteQueryAgent`: responsible file compilation.
- `WebSearchTool`, used by the Suggestor and the Creator to interact with the external environment. This resource is invoked by the agents whenever they need further information about vulnerabilities or up-to-date documentation regarding CodeQL syntax or whenever the Creator needs to correct its generated query.
  The tool is built upon the `duckduckgo-search 8.1.1` [64] Python library, which enables the integration with the *DuckDuckGo* search engine without the need for an official API. This integration is useful for mitigating the risk of hallucinations, a common phenomenon in LLMs where the system generates information that appears consistent but has no basis in reality. The agent, therefore, invokes the tool when necessary and formulates a query for the search engine. *DuckDuckGo* returns the first three results.

4.4.1 SAST tool

This section focuses on the SAST tool. A detailed description of the history and main features of SAST tools is outlined in the paper *Techniques of SAST Tools in the Early Stages of Secure Software Development: A Systematic Literature Review* [65].
Static application security testing (SAST) tools allow for the analysis of source code without needing a compiled version through static analysis [65]. They are commonly used for the vulnerability detection, providing a solution by focusing on analyzing code without having to execute it (indeed, *statically*). SAST tools are defined as a set of techniques that analyze the source code scan without having to execute the code, operating as a white-box approach, unlike its counterpart, DAST (Dynamic Application Security Testing) tools, that require code execution to perform an analysis.
Many of these tools are predefined with certain rules based on CWE (Common Weakness Enumeration) or CVE (Common Vulnerabilities and Exposures), which describe various known and discovered vulnerabilities, providing a broad framework for automatic vulnerability detection by SAST.
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, rather than when they are already executed. 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 [65].

CodeQL

The SAST tool used in this thesis via the `CodeQLSastTool` is CodeQL, a language and toolchain for code analysis, developed in the class `CodeQLSastTool`. It is designed to allow security researchers to scale their knowledge of a single vulnerability to identify variants of that vulnerability across a wide range of codebases. It is also designed to allow developers to automate security checks and integrate them into their development workflows.
CodeQL analysis consists of three steps [63]:

- CodeQL database creation: To create a database, CodeQL first extracts a single relational representation of each source file in the codebase. There is one extractor for each language supported by CodeQL to ensure that the extraction process is as accurate as possible. For multi-language codebases, databases are generated one language at a time. After extraction, all the data required for analysis (relational data, copied source files, and a language-specific database schema, which specifies the mutual relations in the data) is imported into a single directory, known as a CodeQL database (in my project `codeql_db`) [63]. The exact command run the project is:
  ```
  codeql database create codeql_db --language=python
  --source-root=source_root.
  ```
- CodeQL analysis by running CodeQL queries against the new database: after the CodeQL database is created, one or more queries are executed against it. CodeQL queries are written in a specially-designed object-oriented query language called QL [63]. The precise command is:
  ```
  codeql database analyze codeql_db queries_to_run --format=sarifv2.1.0
  --output=output_report_filepath.
  ```
- Interpreting the query results: the final step converts the 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 results should be interpreted. For instance, some queries display a simple message at a single location in the code. Others display a series of locations that represent steps along a data-flow or control-flow path, along with a message explaining the significance of the result. Queries that don't have metadata are not interpreted— their results are output as a table and not displayed in the source code [63].

CodeQL Queries

CodeQL includes queries to find the most relevant and interesting problems (i.e. vulnerabilities) for each supported language.
The important types of query are:

- Alert queries: queries that highlight issues in specific locations in your code.
- Path queries: queries that describe the flow of information between a source and a sink in your code.

Queries written with CodeQL have the file extension ".ql", and contain a select clause. Many of the existing queries include additional optional information and have the following structure:

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

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

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

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

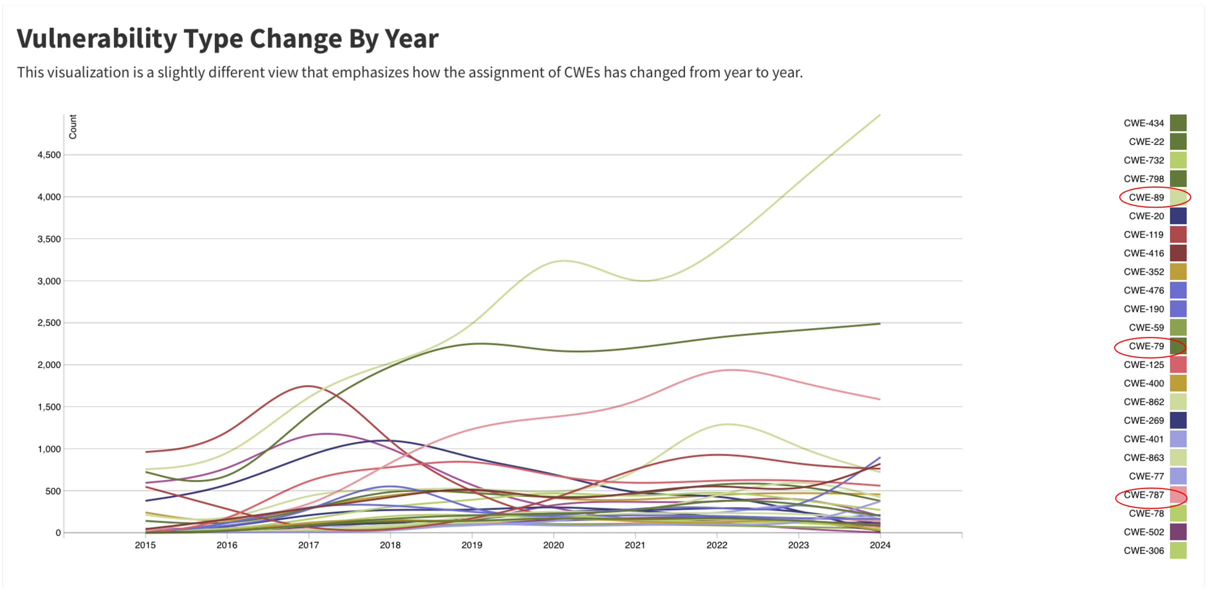
`Query metadata` provide information about the query's purpose, and also specifie how to interpret and display the query results.
Each query generally contains one or more `import` statements, which define the libraries or modules to import into the query. When writing alert queries, the standard library for the language of the queried project should be typically imported - in my case `python`.
The `from` clause declares the variables that are used in the query. Each declaration must be of the form <type> <variable name>.
The `where` clause defines the logical conditions to apply to the variables declared in the from clause to generate your results. This clause uses aggregations, predicates, and logical formulas to limit the variables of interest to a smaller set, which meet the defined conditions. The CodeQL libraries group commonly used predicates for specific languages and frameworks.
The `select` clause specifies the results to display for the variables that meet the conditions defined in the where clause. The valid structure for the select clause is defined by the `@kind` property specified in the metadata [63].

Chapter 5

Experimental Setup

5.1 Dataset

The input of the project consists of a set of fifteen LLMs-generated Python files containing vulnerable source code. In particular, I selected the most common vulnerability - i.e., the one with the highest number of occurrences (see Figure 5.1) detected by the National Institute of Standards and Technology (NIST) [66].


Vulnerability Type Change By Year

This visualization is a slightly different view that emphasizes how the assignment of CWEs has changed from year to year.

Figure 5.1. Graph that shows how the assignment of CWEs has changed over the years.

The Table5.1 shows the details of the vulnerability.

CWE	**Name**	**Description**
CWE-89	SQL Injection	The software constructs all or part of an SQL command using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the intended SQL command when it is sent to a downstream component.

Table 5.1. Vulnerabilities description.

The dataset was generated using a LLM (ChatGPT) based on the following prompt:

```
I need a dataset of Python code with various levels of complexities to test
an LLM-agents based orchestration for autonomous security vulnerabilities
detection. Generate the dataset of 15 different Python code each code has
an SQL Injection presenting the 15 distinct categories that cover most
real-world SQLi cases.
For each example, use a different database use case,
e.g., students, hospital, ecommerce website, crowdsourcing taxi app, etc.
In each of the 15 Python example, document the code sufficiently and at the
start of the file write a short summary what the code does and the SQL
injection in the code. This is an example of a Python code with an SQL
Injection:
"import pymysql
from flask import Flask, request, session
@app.route("/login", methods=['POST'])
def login():
    db = None
    try:
        username = request.values.get('username')
        password = request.values.get('password')
        db = pymysql.connect(host="localhost",
        user="root", passwd="", db="test")
        cursor = db.cursor() # vulnerable query
        query = "SELECT * FROM users WHERE
        username = '%s' AND password = '%s'" % (username, password)
        cursor.execute(query)
        record = cursor.fetchone()
        if record:
            session['logged_user'] = username
            return "Login succed!"
        else:
            return "Unvalid credentials."
    except Exception as e: return f"Error: {e}"

    finally:
        if db:
            db.close() "
Each file must be .py with the category name as file name.
```

Through this prompt, the model produces fifteen distinct Python files, each representative of a different real-world application scenario and each containing a specific SQL Injection category. The prompt emphasizes that each example must exclude a specific vulnerability, be sufficiently documented, and contain a brief initial description of the application's behavior and the existing flaw.
This results in a heterogeneous dataset with different levels of complexity and different contexts. Furthermore, each file contains the explanation of where the vulnerability is contained, facilitating the validation of the architecture results by using it as ground-truth.

5.2 Model

In this experiment I use GPT-4o-mini as the backbone model for all agents. GPT-4o mini enables a broad range of tasks with its low cost and latency, such as 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 interact with customers through fast, real-time text responses (e.g., customer support chatbots) [67]. This model balances logical reasoning ability and execution speed in ReAct iterative cycles. Despite being a "mini" model, i.e. reduced in size, the model demonstrates high efficiency in the field of software development, as evidenced by its score of 87.2% in the HumanEval benchmark [67]. This syntactic accuracy is useful for the Creator, which must map abstract vulnerabilities to specific AST nodes without resulting in code hallucination, i.e. code snippets that appear syntactically correct yet are incorrect or logically inconsistent. In addition, the 128K token context window [68] allows for the simultaneous management of large SAST scan and Suggestor's large reports.

5.3 Implementation details

The system architecture is based on a multi-agent pipeline implemented using LangChain [69] and LangGraph [70], a graph-based framework for agent development. Each component, except for the Orchestrator - so the Analyser, Suggestor, and Creator - operates as an autonomous unit based on the ReAct paradigm. The process begins with the `AnalyzerAgent`, which integrates SAST (CodeQL) tools to map detected vulnerabilities through Pydanitic structure output, enriching the SARIF report with contextual code snippets to mitigate semantic hallucination phenomena. Next, the `SuggestorAgent` analyses the Analyzer's output and a local repository of existing CodeQL queries to try to generate other, more predictive ones. It performs this task by aggregating the Analyzer's validation audits with the results it can obtain by interacting with the external environment via the `WebSearchTool` and with its internal knowledge of query construction. The final phase is executed by the `CreatorAgent`, which manages the generation of ".ql" files. Through the use of dedicated summary (`SummaryProcedure`), thinking (`ThoughtProcedure`) and action (`ActionProcedure`) procedures, the agents carry out the instructions and workflows indicated in their respective prompts and updates its memory after every step and uses it to inform subsequent decisions. The agent connects to the desired LLM backend via API. More specifically, the architecture adopts a modular design and separation of duties. Each agent is isolated in a dedicated module (`analyzer_agent.py`, `creator_agent.py`, `orchestrator.py`, `suggestor_agent.py`) to ensure logical independence. All tools are defined in the `tools.py` file, and each one is a structured class (based on Pydantic) containing the execution logic.

5.3.1 Memory Management Structure

Memory management is handled within the `SharedMemory` class, which uses the Pydantic framework to separate memory into:

- Short-term memory: a list of `ReActChain` objects that implements the agent's scratchpad. Each element records the complete cycle of an iteration: *Summary, Thought, Action, and Observation.*
- Long-term memory: the model uses a dictionary dedicated to global structured data, which is not temporary and can be exchanged between agents at any time.

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: the agent obtains the information saves in the `last_step` variable.
- Saving: the `update_memory` method inserts the new action into `SharedMemory`, making it available to other agents.
- Logging: using the `write_logs` method, the entire memory state is saved in a JSON file, which is useful for debugging.

Chapter 6

Results

6.1 Analysis of Results

This thesis aims to demonstrate the implementation of a multi-agent architecture designed to, firstly, automate the validation of vulnerability detection using the CodeQL static analysis tool. Secondly, the objective is to develop a predictive detection strategy, and translating it into compilable and effective CodeQL code.
Each of these tasks was performed within the architecture by a different agent, in order to avoid to overload a single component with too many instructions and optimize the scalability of the system.
The Analyzer initiates the process by comparing the standard CodeQL results with its own independent file analysis and generates a validation report that includes a custom `agent_validation` field containing the vulnerability location in the file the eventual false positives. Building upon this validation, The Suggestor develops a detection strategy, producing a detailed technical report. Finally, the Creator interprets these strategic requirements to generate new CodeQL queries tailored to the identified security context.
In this section I evaluate the performance and results of each component.

6.1.1 Analyzer validation

The Analyzer invokes the CodeQL SAST tool to scan fifteen files with SQL Injection vulnerability.
This is a SARIF report's item returned from the tool:

Chapter 7

Discussions

7.0.1 Limitations and threats to validity

Chapter 8

Conclusions

8.1 Future works

Bibliography

[1] Aurelien Delaitre Vadim Okun Amine Lbath, Massih-Reza Amini. Ai agentic vulnerability injection and transformation with optimized reasoning. *Paper*, 2025.

[2] Mona Rahimi Samiha Shimmi, Hamed Okhravi. Ai-based software vulnerability detection: A systematic literature review. *Paper*, 2025.

[3] Baleegh Ahmad Ramesh Karri Brendan Dolan-Gavitt Hammond Pearce, Benjamin Tan. Examining zero-shot vulnerability repair with large language models. *Paper*, 2022.

[4] Xinda Wang Eric Wong Youpeng Li, Kartik Joshi. Mavul: Multi-agent vulnerability detection via contextual reasoning and interactive refinement. *Paper*, 2025.

[5] Cybersecurity and Infrastructure Security Agency (CISA). Widespread it outage due to crowdstrike update, 2024. Accessed: 2025-08-16.

[6] Cybersecurity and Infrastructure Security Agency (CISA). Reported supply chain compromise affecting xz utils data compression library, cve-2024-3094, 2024. Accessed: 2025-08-16.

[7] Siddhartha Varma Nadimpalli Noone Srinivas1, Nagaraj Mandaloju2. Leveraging automation in software quality assurance: Enhancing efficiency and reducing defects. *Paper*, 2024.

[8] Elissa Redmiles Jeremy Hu Michelle Mazurek Daniel Votipka, Rock Stevens. Hackers vs. testers: A comparison of software vulnerability discovery processes. *Paper*, 2018.

[9] José Antonio Vergara Camachoc-Gerardo Contreras Vegad Adrián Herrera Jerónimoa, Patricia Martínez Morenob. Techniques of sast tools in the early stages of secure software development: A systematic literature review. *Paper*, 2024.

[10] Thomas Vogela Timo Kehrera Lars Grunskea Laura Wartschinski, Yannic Nollera. Vudenc: Vulnerability detection with deep learning on a natural codebase for python. *Paper*, 2022.

[11] Lei Hamilton Tomo Lazovich Jacob Harer-Onur Ozdemir Rebecca Russell, Louis Kim. Automated vulnerability detection in source code using deep representation learning. *Paper*, 2018.

[12] Omniyyah Ibrahim Chawin Sitawarin Xinyun Chen-Basel Alomair David Wagner Baishakhi Ray Yizheng Chen Yangruibo Ding, Yanjun Fu. Vulnerability detection with code language models: How far are we? *Paper*, 2024.

[13] Xin Zhou, Ting Zhang, and David Lo. Large language models for vulnerability detection: Emerging results and future directions. *XXXX*, XXXX.

[14] Microsoft exchange flaw: Attacks surge after code published, 2022. Accessed: 2025.

[15] Dean Turner, Marc Fossi, Eric Johnson, Trevor Mack, Joseph Blackbird, Stephen Entwisle, Mo King Low, David McKinney, and Candid Wueest. Symantec global internet security threat report: Trends for july–december 2007. Technical report, Symantec Enterprise Security, 2008.

[16] Hazim Hanif and Sergio Maffeis. Vulberta: Simplified source code pre-training for vulnerability detection. In *2022 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2022.

[17] Michael Fu and Chakkrit Tantithamthavorn. Linevul: A transformer-based line-level vulnerability prediction. In *Proceedings of the 19th International Conference on Mining Software Repositories (MSR)*, pages 608–620, 2022.

[18] Van-Anh Nguyen, Dai Quoc Nguyen, Van Nguyen, Trung Le, Quan Hung Tran, and Dinh Phung. Regvd: Revisiting graph neural networks for vulnerability detection. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pages 178–182, 2022.

[19] 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*, volume 32, 2019.

[20] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547. Association for Computational Linguistics, 2020.

[21] Chunqiu Steven Xia and Lingming Zhang. Keep the conversation going: Fixing 162 out of 337 bugs for $0.42 each using chatgpt, 2023.

[22] Ting Zhang, Ivana Clairine Irsan, Ferdian Thung, and David Lo. Cupid: Leveraging chatgpt for more accurate duplicate bug report detection, 2023.

[23] Ting Zhang, Ivana Clairine Irsan, Ferdian Thung, and David Lo. Revisiting sentiment analysis for software engineering in the era of large language models, 2023.

[24] Martin Weyssow, Xin Zhou, Kisub Kim, David Lo, and Houari Sahraoui. Exploring parameter-efficient fine-tuning techniques for code generation with large language models, 2023.

[25] Xin Zhou, Bowen Xu, Kisub Kim, DongGyun Han, Thanh Le-Cong, Junda He, Bach Le, and David Lo. Patchzero: Zero-shot automatic patch correctness assessment, 2023.

[26] 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, XXXX. Preprint.

[27] Yair Shavit, Sandhini Agarwal, Miles Brundage, Steven Adler, Cullen O'Keefe, Ryan Campbell, Tom Lee, Pamela Mishkin, Tyna Eloundou, Alex Hickey, Kieran Slama, Lama Ahmad, Peter McMillan, Alex Beutel, Alexandre Passos, and David G. Robinson. Practices for governing agentic ai systems. Research paper, OpenAI, December 2023.

[28] Zhiheng Xi, Wen Chen, Xiaobin Guo, Wei He, Yifan Ding, Bowen Hong, Ming Zhang, Jie Wang, Shun Jin, Encheng Zhou, Rui Zheng, Xiaohui Fan, Xinyu Wang, Li Xiong, Yujia Zhou, Wei Wang, Chen Jiang, Yifan Zou, Xiaolong Liu, Zhe Yin, Shuang Dou, Rui Weng, Wen Cheng, Qi Zhang, Wei Qin, Yuxuan Zheng, Xipeng Qiu, Xuanjing Huang, and Tao Gui. The rise and potential of large language model based agents: A survey. *Science China Information Sciences*, 68(2):1–44, 2025.

[29] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy P. Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, 2017.

[30] Carlos Ribeiro. Reinforcement learning agents. *Artificial Intelligence Review*, 17:223–250, 2002.

[31] C. H. Song, J. Wu, C. Washington, B. M. Sadler, W.-L. Chao, and Y. 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 (ICCV)*, pages 2998–3009, 2023.

[32] Y. Cheng, C. Zhang, Z. Zhang, X. Meng, S. Hong, W. Li, Z. Wang, Z. Wang, F. Yin, J. Zhao, and X. He. Exploring large language model based intelligent agents: Definitions, methods, and prospects. *arXiv preprint arXiv:2401.03428*, pages 1–55, 2024.

[33] M. Fırat and S. Kuleli. What if gpt4 became autonomous: The auto-gpt project and use cases. *Journal of Emerging Computer Technologies*, 3(1):1–6, 2023.

[34] Q. Wu, G. Bansal, J. Zhang, Y. Wu, B. Li, E. E. Zhu, L. Jiang, X. Zhang, S. Zhang, A. Awadallah, R. W. White, D. Burger, and C. Wang. Autogen: Enabling next-gen llm applications via multi-agent conversation. In *Proceedings of COLM*, pages 1–43, 2024.

[35] Y. Chen, J. Yoon, D. S. Sachan, Q. Wang, V. Cohen-Addad, M. Bateni, C. Lee, and T. Pfister. Re-invoke: Tool invocation rewriting for zero-shot tool retrieval. In *Proceedings of EMNLP (Findings)*, pages 4705–4726, 2024.

[36] R. Nakano, J. Hilton, S. Balaji, J. Wu, L. Ouyang, C. Kim, C. Hesse, S. Jain, V. Kosaraju, W. Saunders, X. Jiang, K. Cobbe, T. Eloundou, G. Krueger, K. Button, M. Knight, B. Chess, and J. Schulman. Webgpt: Browser-assisted question-answering with human feedback. *arXiv preprint arXiv:2112.09332*, pages 1–32, 2022.

[37] J. Wang, H. Xu, H. Jia, X. Zhang, M. Yan, W. Shen, J. Zhang, F. Huang, and J. Sang. Mobile-agent-v2: Mobile device operation assistant with effective navigation via multi-agent collaboration. In *Proceedings of NeurIPS*, pages 2686–2710, 2024.

[38] S. Hu, T. Huang, F. Ilhan, S. Tekin, G. Liu, R. Kompella, and L. Liu. A survey on large language model-based game agents. *arXiv preprint arXiv:2404.02039*, pages 1–23, 2024.

[39] M. Ahn, A. Brohan, Y. Chebotar, and et al. Do as i can, not as i say: Grounding language in robotic affordances. In *Proceedings of the Annual Conference on Robot Learning (CoRL)*, pages 1–34, 2022.

[40] Y. Jin, R. Yang, Z. Yi, X. Shen, H. Peng, X. Liu, J. Qin, J. Li, J. Xie, P. Gao, G. Zhou, and J. Gong. Surrealdriver: Designing llm-powered generative driver agent framework based on human drivers' driving-thinking data. In *Proceedings of IROS*, pages 966–971, 2024.

[41] G. Deng, Y. Liu, V. Mayoral-Vilches, P. Liu, Y. Li, Y. Xu, T. Zhang, Y. Liu, M. Pinzger, and S. Rass. Pentestgpt: Evaluating and harnessing large language models for automated penetration testing. In *Proceedings of USENIX Security*, pages 847–864, 2024.

[42] MarketsandMarkets. Autonomous ai and autonomous agents market. `https://www.marketsandmarkets.com/Market-Reports/autonomous-ai-and-autonomous-agents-market-208190735.html`, 2023. Accessed: July 30, 2023.

[43] A. Yildiz, S. G. Teo, Y. Lou, Y. Feng, C. Wang, and D. M. Divakaran. Benchmarking llms and llm-based agents in practical vulnerability detection for code repositories. In W. Che, J. Nabende, E. Shutova, and M. T. Pilehvar, editors, *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2025*, pages 30848–30865, Vienna, Austria, July 27–August 1 2025. Association for Computational Linguistics.

[44] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. A c/c++ code vulnerability dataset with code changes and cve summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories (MSR)*, pages 508–512, 2020.

[45] Chao Ni, Liyu Shen, Xiaohu Yang, Yan Zhu, and Shaohua Wang. Megavul: A c/c++ vulnerability dataset with comprehensive code representations. In *Proceedings of the 2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*, pages 738–742, 2024.

[46] Yizheng Chen, Zhoujie Ding, Lamya Alowain, Xinyun Chen, and David Wagner. Diversevul: A new vulnerable source code dataset for deep learning-based vulnerability detection. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses (RAID '23)*, pages 654–668, New York, NY, USA, 2023. Association for Computing Machinery.

[47] Xinchen Wang, Ruida Hu, Cuiyun Gao, Xin-Cheng Wen, Yujia Chen, and Qing Liao. Reposvul: A repository-level high-quality vulnerability dataset. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings (ICSE-C)*, pages 472–483, 2024.

[48] Xin-Cheng Wen, Xinchen Wang, Yujia Chen, Ruida Hu, David Lo, and Cuiyun Gao. Vuleval: Towards repository-level evaluation of software vulnerability detection. *arXiv preprint arXiv:2404.15596*, 2024.

[49] Mend.io. Mend.io vulnerability database: The largest open source vulnerability database. `https://www.mend.io/vulnerability-database/`, 2025. Accessed: 2025.

[50] NIST. National vulnerability dataset, 2025. Accessed: December 8, 2025.

[51] Chong Wang, Jianan Liu, Xin Peng, Yang Liu, and Yiling Lou. Boosting static resource leak detection via llm-based resource-oriented intention inference. In *Proceedings of the 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 668–668. IEEE Computer Society, 2025.

[52] Cheng Wen, Yuandao Cai, Bin Zhang, Jie Su, Zhiwu Xu, Dugang Liu, Shengchao Qin, Zhong Ming, and Tian Cong. Automatically inspecting thousands of static bug warnings with large language models: How far are we? *ACM Transactions on Knowledge Discovery from Data*, 18(7):1–34, 2024.

[53] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Wei Ma, Lyuye Zhang, Yang Liu, and Yingjiu Li. Llm4vuln: A unified evaluation framework for decoupling and enhancing llms' vulnerability reasoning. *arXiv preprint arXiv:2401.16185*, 2024.

[54] Mete Keltek, Rong Hu, Mohammadreza Fani Sani, and Ziyue Li. Lsast – enhancing cybersecurity through llm-supported static application security testing. *arXiv preprint arXiv:2409.15735*, 2024.

[55] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *Proceedings of [Conference/Workshop Name]*, page [page numbers], [year].

[56] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

[57] Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *arXiv preprint arXiv:2303.11366 [cs]*, 2023.

[58] Ted Ridnik, Dedy Kredo, and Itamar Friedman. Code generation with alphacodium: From prompt engineering to flow engineering. `https://www.example.com/alphacodium-paper`, 2025.

[59] IBM. What is a react agent?, 2025. Accessed: December, 2025.

[60] Theodore R. Sumers, Shunyu Yao, Karthik Narasimhan, and Thomas L. Griffiths. Cognitive architectures for language agents. *arXiv preprint arXiv:2309.02427*, 2023.

[61] Mahmoud Mohammadi, Yipeng Li, Jane Lo, and Wendy Yip. Evaluation and benchmarking of llm agents: A survey. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '25), Volume 2*, pages 6129–6139, New York, NY, USA, 2025. Association for Computing Machinery.

[62] Kai Huang Marco Mellia Dario Rossi-Matteo Boffa Zied Ben Houidi Stefano Fumero, Danilo Giordano. Cybersleuth: Autonomous blue-team llm agent for web attack forensics. *Paper*, 2025.

[63] CodeQL. Codeql documentation, 2025. Accessed: December, 2025.

[64] 2025. Accessed: December, 2025.

[65] 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)*. IEEE, 2024. ©2024 IEEE.

[66] NIST. National vulnerability dataset - cwe over time, 2025. Accessed: December, 2025.

[67] OpenAI. Gpt-4o mini: advancing cost-efficient intelligence, 2025. Accessed: December, 2025.

[68] OpenAI. Gpt-4o mini, 2025. Accessed: December, 2025.

[69] LangChain. Langchainbuild agents faster, your way, 2025. Accessed: December, 2025.

[70] LangChain. Balance agent control with agency, 2025. Accessed: December, 2025.