Justus Sydänmaa

A Multi-Agent Approach to Vulnerability Detection

School of Technology and Innovations
Master's Thesis in Information Systems
Master's Programme in Information Systems

Vaasa 2026

UNIVERSITY OF VAASA
School of Technology and Innovations
Author: Justus Sydänmaa
Title of the Thesis: A Multi-Agent Approach to Vulnerability Detection
Degree: Master of Science in Economics and Business Administration
Programme: Information Systems
Supervisor: Laura Havinen and Teemu Mäenpää
Year: 2026 **Pages:** 87

ABSTRACT:
Software systems keep getting more complex, and the number of reported software vulnerabilities keeps rising. Traditional security analysis methods, including manual code review and static analysis, are often insufficient to address these challenges efficiently and cost-effectively. They also report findings without the context needed to judge exploitability, which leaves much of the explanation work to human reviewers. Large language models (LLMs) and multi-agent systems open new directions for automated vulnerability analysis, because these agents can divide the task of gathering, interpreting, and reporting findings between them.

This thesis aims to design an artificial intelligence-based multi-agent system architecture that supports automated source code analysis enriched with public vulnerability data. The scope of the work focuses on defining architectural components and agent roles rather than evaluating detection accuracy.

The research follows the Design Science Research methodology. The process includes problem identification, the definition of solution objectives, the design and development of the artifact, its demonstration through a proof of concept (PoC) implementation, and the communication of the results. The artifact was designed based on existing research in software security, multi-agent systems, and vulnerability standards, and is realized through an LLM-based multi-agent implementation that demonstrates feasibility. In the implementation, source code findings are classified against Common Weakness Enumeration categories, while Common Vulnerabilities and Exposures identifiers are attached to outdated third-party dependencies found in the target repository.

This study proposes an architecture that specifies key components and agent roles of an automated code analysis system that is enriched with public vulnerability data, and the practical viability is confirmed through the PoC. The findings indicate that the approach can support the development of adaptable threat-detection pipelines, particularly in settings where an organization prefers to assemble its own analysis pipeline from open components rather than use a closed commercial product. Further work is needed to validate the architecture against larger codebases and to measure its effectiveness in real development workflows.

KEYWORDS: Large Language Models, Multi-Agent Systems, Static Application Security Testing, Vulnerability Detection, Model Context Protocol

VAASAN YLIOPISTO
Tekniikan ja innovaatiojohtamisen akateeminen yksikkö
Kirjoittaja: Justus Sydänmaa
Tutkielman otsikko: A Multi-Agent Approach to Vulnerability Detection
Tutkinto: Kauppatieteiden maisteri
Oppiaine: Tietojärjestelmätiede
Ohjaaja: Laura Havinen ja Teemu Mäenpää
Valmistumisvuosi: 2026 **Sivumäärä:** 87

TIIVISTELMÄ:
Ohjelmistojärjestelmät muuttuvat yhä monimutkaisemmiksi ja raportoitujen ohjelmistohaavoittuvuuksien määrä kasvaa jatkuvasti. Perinteiset tietoturva-analyysimenetelmät, kuten manuaalinen koodikatselmointi ja staattinen analyysi, eivät useinkaan riitä vastaamaan näihin haasteisiin riittävän tehokkaasti ja kustannustehokkaasti. Ne tuovat havaintoja esiin ilman tarvittavaa kontekstia haavoittuvuuden hyväksikäytettävyyden arvioimiseksi, jolloin tulkintatyö jää suurelta osin ihmisen vastuulle. Suuret kielimallit (Large Language Models, LLM) ja moniagenttijärjestelmät avaavat uusia suuntia haavoittuvuuksien automaattiselle analyysille, sillä erikoistuneet agentit voivat jakaa keskenään havaintojen keräämisen, tulkitsemisen ja raportoinnin.

Tämän tutkielman tavoitteena on suunnitella tekoälypohjainen moniagenttijärjestelmän arkkitehtuuri, joka suorittaa automaattista lähdekoodianalyysiä, hyödyntäen julkisia haavoittuvuustietoja. Työn rajaus keskittyy komponenttien ja agenttien roolien määrittelyyn sen sijaan, että arvioitaisiin sen havaitsemistarkkuutta.

Tutkimus noudattaa suunnittelutieteellistä tutkimusmenetelmää (Design Science Research, DSR). Prosessi käsittää ongelman tunnistamisen, ratkaisun tavoitteiden määrittelyn, artefaktin suunnittelun ja kehittämisen, sen demonstroinnin konseptitodistuksen (proof of concept, PoC) muodossa sekä tulosten viestittämisen. Artefakti suunniteltiin olemassa olevan ohjelmistoturvallisuutta, moniagenttijärjestelmiä ja haavoittuvuusstandardeja käsittelevän tutkimuksen pohjalta ja toteutettiin LLM-pohjaisena moniagenttiratkaisuna, joka osoittaa lähestymistavan toteutettavuuden. Toteutuksessa lähdekoodista löytyvät havainnot luokitellaan Common Weakness Enumeration kategorioihin, kun taas Common Vulnerabilities and Exposures tunnisteet liitetään kohdeohjelmistosta löytyviin vanhentuneisiin kolmannen osapuolen riippuvuuksiin.

Tutkimus esittää arkkitehtuurin julkisella haavoittuvuustiedolla rikastetulle automaattiselle koodianalyysijärjestelmälle sekä määrittelee sen keskeiset komponentit ja agenttien roolit. Ratkaisun käytännön toteutus varmistettiin luomalla siitä PoC. Tulokset osoittavat, että lähestymistapa voi tukea joustavien uhkien havaitsemisanalyysiputkien kehittämistä. Tämä on hyödyllistä erityisesti silloin, kun organisaatio haluaa rakentaa oman analyysiputkensa avoimista komponenteista kaupallisen suljetun ratkaisun sijaan. Jatkotyössä arkkitehtuuria tulee validoida suurempia koodikantoja vastaan ja mitata sen vaikuttavuutta todellisissa kehitysprosesseissa.

AVAINSANAT: Large Language Models, Multi-Agent Systems, Static Application Security Testing, Vulnerability Detection, Model Context Protocol

Contents

Figures

Tables

Abbreviations

AI	Artificial Intelligence
API	Application Programming Interface
CI/CD	Continuous Integration and Continuous Delivery
CVE	Common Vulnerabilities and Exposures
CVSS	Common Vulnerability Scoring System
CWE	Common Weakness Enumeration
DSR	Design Science Research
DSRM	Design Science Research Methodology
LLM	Large Language Model
LLM-MA	Large Language Model-based Multi-Agent
LTM	Long-Term Memory
MAS	Multi-Agent System
MCP	Model Context Protocol
NVD	National Vulnerability Database
PoC	Proof of Concept
RAG	Retrieval-Augmented Generation
SAST	Static Application Security Testing
SCA	Software Composition Analysis
SDLC	Software Development Life Cycle
SQLi	SQL Injection
STM	Short-Term Memory
ToM	Theory of Mind
XSS	Cross-Site Scripting

1 Introduction

Software systems are continually evolving, with their complexity increasing. At the same time, new vulnerabilities are discovered. In 2023, 28,961 Common Vulnerabilities and Exposures (CVE) were published, and by 2025, that number had increased to over 48,000 (CVE Program, n.d.-a). This highlights the scale and ongoing trend of growth in the vulnerabilities that software systems face. These vulnerabilities serve as prime entry points for malicious actors, triggering cybercrimes (Taghavi Far & Feyzi, 2025, p. 13). Additionally, vulnerability can lead to a loss of user trust, damage to brand reputation, and ultimately, a decline in the customer base (Anwar et al., 2020, p. 1). Furthermore, the earlier the defects are detected in software systems, the easier and cheaper they are to fix (Tan et al., 2017, pp. 3–4).

Traditional methods, such as manual code inspections and static code analysis alone, are often insufficient to ensure software security, as vulnerabilities may remain despite development-stage precautions (Parizi et al., 2018, p. 1). According to Mweu & Ndia (2025, p. 8), code inspection tools produce numerous false positives, requiring developers to manually verify each alert. This increases their workload and can lead to critical vulnerabilities being missed because of alert fatigue. These methods are slow and laborious, and there is an ever-increasing need for automated tools to detect and address software vulnerabilities (Taghavi Far & Feyzi, 2025, p. 3).

Recent advances in artificial intelligence (AI), particularly large language models (LLMs), offer new possibilities for effective automated vulnerability and threat analysis. LLMs offer enhanced accuracy and natural language processing capabilities, streamlining detection and enabling faster processing than traditional methods (Taghavi Far & Feyzi, 2025, p. 13). According to Kereopa-Yorke (2023, p. 1), the capability of AI to process large datasets in real-time, extract insights, and anticipate potential threats could revolutionize proactive cybersecurity measures.

1.1 Aim and method of the study

In this study, a design for an AI-based multi-agent architecture for automated source code security analysis is proposed. The proposed solution leverages the large language model-based multi-agent (LLM-MA) system to analyze source code and identify potential security vulnerabilities using public vulnerability data retrieved through external application programming interfaces (APIs).

This research topic originates from a software company that identified problems arising from practical challenges encountered in developing software systems. Existing commercial solutions for automated source code analysis have high-cost licenses, while developing an in-house solution would reduce costs and provide tangible value to existing development workflows.

This study addresses the following research questions:

RQ1: Which architectural components are required to design a multi-agent system that supports automated vulnerability analysis?

RQ2: What agent roles are needed in a multi-agent system to perform automated source-code analysis enriched with vulnerability data?

This research uses the design science research (DSR) methodology and follows a design science process developed by Peffers et al. (2007) that includes six activities. Due to the scope of this thesis, not all steps are included. The solution is demonstrated through a proof of concept (PoC) in the *Demonstration* phase, while the *Evaluation* phase is beyond the scope of this study.

The DSR will produce an artifact, which in this study is an architecture for an AI-based multi-agent system that analyzes source code and detects vulnerabilities by integrating CVE data collected through external APIs. The research questions are addressed by demonstrating the DSR artifact in the artifact-building section.

This thesis aims to demonstrate that an AI-based multi-agent system provides significant benefits for detecting vulnerabilities and can operate autonomously, communicate between agents, and clearly report identified deficiencies or issues to the user. The system successfully uses CVE databases via the Model Context Protocol (MCP) and identifies problems in the source code.

The research provides valuable insights for researchers and offers a practical example of how agent systems currently operate, and how they can be enhanced and improved. The results of the study can be applied and used on a larger scale, for example, in more complex projects. Additionally, the architecture serves as a good example of how such systems should be built to ensure the most useful outcomes and to enable efficient vulnerability detection.

This study focuses on component-level testing, which involves evaluating software components in isolation and is usually conducted by developers (ISTQB, 2024, p. 29). It emphasizes identifying defects within individual components and their dependencies rather than testing entire software systems.

1.2 Structure of the thesis

This thesis is organized into six chapters.

Chapter 1: Introduction – offers an overview of the research context, discusses the motivation, identifies the main problems, and outlines the research questions and objectives.

Chapter 2: Theoretical foundations of AI-based vulnerability analysis – introduces software security and vulnerability intelligence, defines the vulnerability specific terminology, and describes the role of vulnerability information in software development. The chapter then surveys static, dependency-based, and dynamic approaches to

vulnerability detection, and finally explains what large-language-model based multi-agent systems are and how their cooperation strategies, reasoning capabilities, memory, and tool use relate to the analysis problem.

Chapter 3: Existing approaches to automated source-code analysis – reviews previous research on AI-based systems that analyze source code for vulnerabilities. The chapter ends by drawing the main insights from these solutions and identifying the research gap addressed in this thesis.

Chapter 4: Design science research approach – describes the research methodology used in the study. First, it introduces Design Science Research and the associated procedures and activities. Then, these activities are described in more detail.

Chapter 5: Artifact development – develops the artifact following best practices from the framework by Peffers et al. (2007).

Chapter 6: Discussion – summarizes key findings, answers the research questions, relates the results to prior research, discusses the limitations of the study, and outlines directions for future research.

2 Theoretical foundations of AI-based vulnerability analysis

This chapter presents the theoretical background required to understand AI-based vulnerability analysis in software systems. The chapter begins by introducing the fundamentals of cybersecurity concepts, including software vulnerabilities, threats, and exploits. Then it explains how vulnerability intelligence is structured and shared through standardized frameworks such as CVE and the Common Vulnerability Scoring System (CVSS). Next, the chapter defines existing approaches for detecting vulnerabilities in software systems, including static, dependency-based methods, and dynamic analysis techniques. Finally, the chapter introduces the LLM-MA system as a modern approach to address challenges in vulnerability detection.

2.1 Software vulnerabilities and vulnerability intelligence

Understanding cybersecurity requires clarifying the differences among terms such as vulnerability, threat, and exploit. A software vulnerability is a flaw or weakness in a system that can lead to a security incident (Sánchez et al., 2020, p. 270). According to Michael & Bellis (2023, p. 4), a vulnerability that can be exploited by an attacker is called a threat, which becomes a concrete security threat after the attacker has created code that exploits the vulnerability.

When an actual event where an exploit successfully leverages a vulnerability, it is turned into exploitation. However, not all exploitations cause the same level of damage. Some exploitations may only reveal non-sensitive information, such as publicly available data, while others could result in serious issues, such as gaining access to a system (Michael & Bellis, 2023, p. 4).

Applications and vulnerabilities can be exploited through attack vectors, which are paths that enable attacks to access them (Ozkaya, 2019, p. 11). According to Ozkaya (2019, p. 11), typical attack vectors include social engineering, phishing schemes, drive-by-

downloads, malicious URLs or scripts, browser-based attacks, supply chain breaches, and network-based attacks. Exploiting these paths allows attackers to have more targeted attacks against applications and systems. This process is illustrated Figure 1, which presents a simplified attack chain showing how an attacker uses an attack vector to exploit a vulnerability and reach a target.

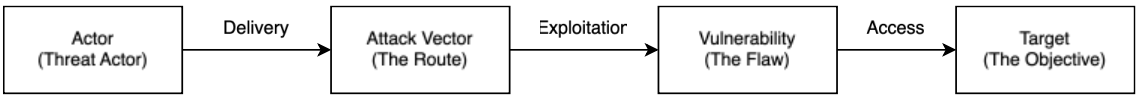

Figure 1. Simplified attack chain illustrating how an attacker exploits a vulnerability through an attack vector to reach a target. Adapted from Ozkaya (2019, p. 11).

Software vulnerabilities can appear in many forms. Some of the most common security issues are SQL injections (SQLi), cross-site scripting (XSS), malware, and denial-of-service attacks, which exploit underlying vulnerabilities in software systems.

SQL injection attacks target databases that are exposed through the web, where attackers can run malicious SQL queries against applications that lack security or are poorly configured (Graham et al., 2010, p. 138). By executing a successful SQL injection, attackers may get access to the databases, allowing them to manipulate or modify data. To prevent SQL injection, developers should ensure that only the desired data is accepted and validate all inputs for malicious content before submitting them to the database for queries (Ozkaya, 2019, p. 15).

The most common web application-focused input vulnerability is XSS, in which malicious client-side scripts are injected into a website, thereby compromising user security (Graham et al., 2010, pp. 172–176). The two main types of XSS attacks are stored and reflected. In stored attacks, as explained by Ozkaya (2019, p. 14), attackers permanently insert malicious scripts into the server. In reflected attacks, attackers typically send a link containing malicious queries, leading the user to a malicious page where sensitive data can be stolen (Ozkaya, 2019, p. 14).

However, not all attacks focus on web application input vulnerabilities. For example, malware is malicious software that is designed to damage systems. According to Ozkaya (2019, p. 14), two common types of malware are worms, which can spread and reproduce themselves, and Trojans, which rely on social engineering to trick users into downloading them onto their computers. Successful malware attacks can lead to data theft, cause extensive damage to targeted computer systems, and create network disruptions (Benmalek, 2024, p. 198). Modern malware variants use advanced techniques to hide themselves from security systems and can remain active for weeks or even years (Ozkaya, 2019, p. 14).

Another common attack is a denial of service (DoS) attack that has a goal to disrupt service availability for its legitimate users (Sutton, 2022, p. 18). In a DoS attack, the attacker overwhelms the targeted system by sending more requests or data than it can handle, causing the system to stop working for its users (Ozkaya, 2019, p. 13). A more advanced version of this type of attack is a distributed denial of service (DDoS) attack, where multiple systems target a single service simultaneously (Ozkaya, 2019, p. 13).

To track existing threats, threat actors, and emerging attack trends, cybersecurity professionals use a threat landscape to analyze these elements and the relationships between threats and assets, providing a comprehensive view of the current cybersecurity situation (Ivanov et al., 2021, p. 1). This analysis is supported by organizations that publish reports analyzing and covering relevant threats in the cyber threat landscape. For example, the European Union Agency for Network and Information Security (ENISA) published a detailed report analyzing nearly 4,900 cybersecurity incidents from 2025 (ENISA, n.d.). Additionally, the Finnish authority Traficom publishes “Cyber Weather” reports monthly to highlight key events from the past month (Traficom, n.d.).

2.1.1 Vulnerability classification systems

The CVE is a standardized list that documents known cybersecurity vulnerabilities, each with a unique ID, a description, and public references (CVE Program, n.d.-b). According to Michael & Bellis (2023, p. 6), it is a valuable community resource for vulnerability management, enabling security professionals to share information effectively, and has become an industry standard in many projects and products. However, the CVE list is not perfect, as many vulnerabilities remain either unknown, undisclosed, or unassigned with CVE IDs (Michael & Bellis, 2023, p. 6). As illustrated in Figure 2, published CVE records are increasing each year.

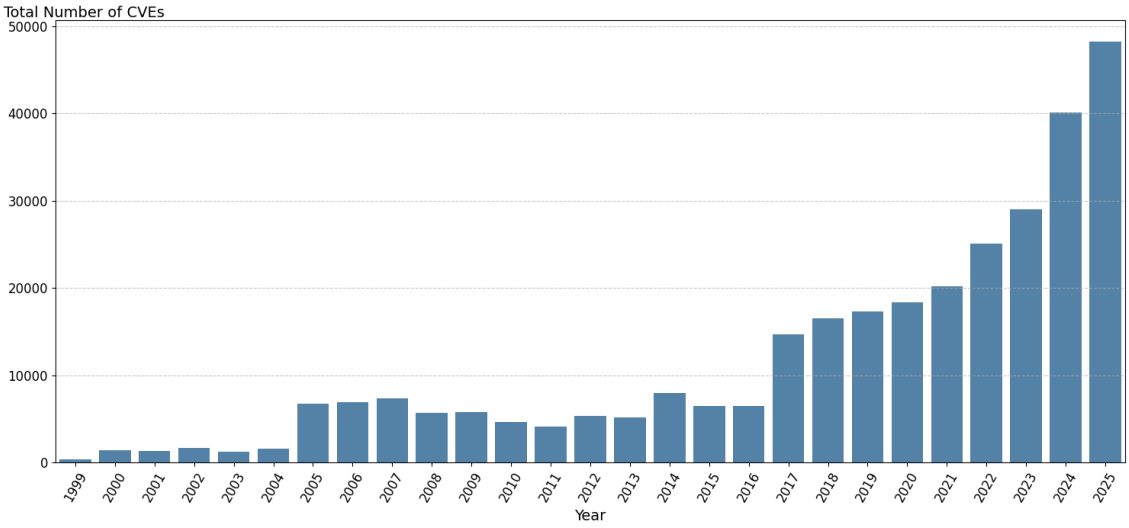

Figure 2. Evolution of the total number of published CVE records from 1999 to 2025. Adapted from CVE Program (n.d.-a).

Since 2022, there has been a significant growth in the number of reported vulnerabilities. This rise of published vulnerabilities can be explained by several factors: an increase in code volume, greater complexity and size of software systems, and more frequent use of third-party components (Dietrich et al., 2024, p. 77). Moreover, the use of advanced vulnerability detection tools and artificial intelligence has improved the ability to identify vulnerabilities (Malkawi & Alhajj, 2026, p. 1).

While CVE catalogues known cybersecurity vulnerabilities in specific products and versions, the common weakness enumeration (CWE) is a community-developed list of software and hardware weakness types maintained by MITRE (MITRE, n.d.-a). Each CWE entry describes a general class of flaw, for example, CWE-89 covers SQL injection, CWE-94 covers code injection, and CWE-798 covers the use of hard-coded credentials. CVE record has usually references to one or more CWE identifiers that categorise the underlying weakness (MITRE, n.d.-b).

2.1.2 Vulnerability severity assessment

After a vulnerability is correctly identified and assigned a CVE-ID, its severity is typically calculated using the CVSS to provide a standardized risk assessment for prioritization. CVSS provides detailed scoring information by defining a set of metrics that explain how each score is calculated and enable consistent comparison between vulnerabilities (Mell et al., 2006, p. 85). CVSS provides standardized vulnerability scores ranging from 0.0 to 10.0 in increments of 0.1, representing the severity of a vulnerability, as shown in Table 1.

Table 1. CVSS v4.0 severity ratings by score range. Adapted from NVD (n.d.).

Score Range	Severity
0.0	None
0.1-3.9	Low
4.0-6.9	Medium
7.0-8.9	High
9.0-10.0	Critical

According to Aggarwal (2023, p. 1182) and Balsam et al. (2024, p. 1), as illustrated in Figure 3, CVSS uses three key metric groups to calculate the final vulnerability score: Base, Threat, and Environment metrics.

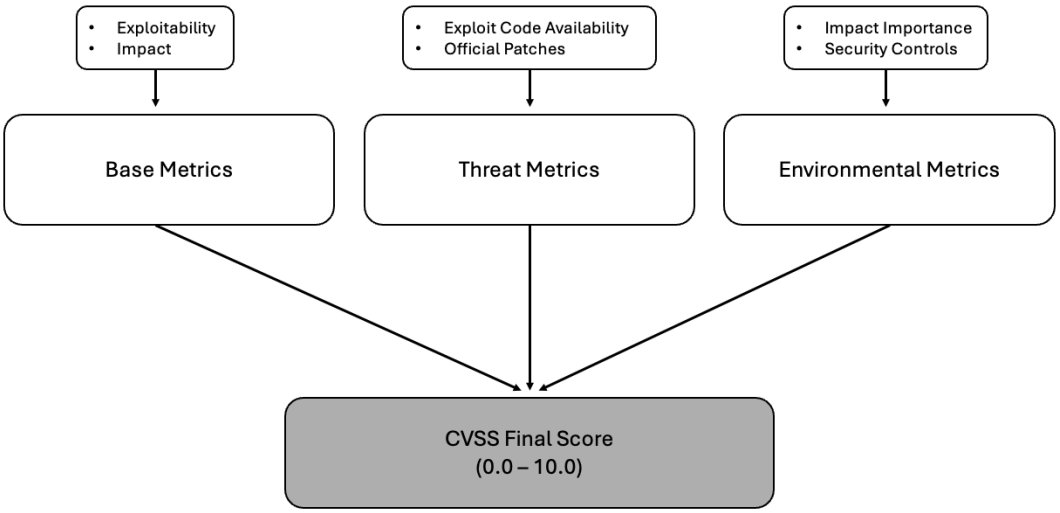

Figure 3. CVSS v4.0 scoring components structure. Adapted from Aggarwal (2023, p. 1182) and Balsam et al. (2024, p. 1).

Base metrics describe the characteristics of a vulnerability that remain constant over time and across different environments. The base score is derived from two metric sets: exploitability, which reflects how easily a vulnerability can be exploited, and impact metrics, which evaluate the potential damage to the system's confidentiality, integrity, and availability if exploited (Aggarwal, 2023, p. 1182; Coutinho et al., 2024, p. 18).

Threat metrics describe the characteristics of a vulnerability that may evolve over time but is not tied to specific user environments, reflecting the present threat landscape, including the availability of exploit code or official patches (Aggarwal, 2023, p. 1182). Environmental metrics reflect the importance of confidentiality, integrity, and availability within the organizational context, as well as how existing security controls may reduce the impact of vulnerabilities (Aggarwal, 2023, p. 1182).

There is also a set of supplemental metrics that do not affect the final score but provide more details about the vulnerability, aiding in a better overall understanding (Balsam et al., 2024, pp. 1–2). These include metrics such as safety, automatability, provided

urgency, recovery, value density, and vulnerability response effort (Aggarwal, 2023, pp. 1182–1183). The base score is the only required metric for the final score, while threat, environmental, and supplement metrics are optional (Aggarwal, 2023, p. 1183).

2.2 Vulnerability detection in software systems

This section discusses various techniques for detecting vulnerabilities in software systems. When reviewing source code, two primary approaches can be identified: static and dynamic analysis, as summarized in Table 2.

Table 2. Comparison of static and dynamic analysis.

Aspect	**Static analysis**	**Dynamic analysis**
Definition	Analysis of code without execution	Analysis through program execution
Purpose	Detect vulnerabilities early	Identify runtime errors
Software Lifecycle	Early phases	Later phases, testing and deployment phases

The table highlights the key differences between static and dynamic analysis. These approaches occur at different stages of the software lifecycle and emphasize different focus areas. After discussing these two primary techniques, the section proceeds to cover dependency-based vulnerability detection and concludes with a summary and potential future research directions.

2.2.1 Static analysis

Static analysis is a process in which a software system component is evaluated based on its form, structure, content, or documentation without executing the program (Liu et al., 2012, p. 1). It is employed in the early stages of the software development life cycle to detect software flaws, logical errors, and other potential issues (Cong et al., 2024, p. 2). Traditional static analysis techniques, such as manual code inspection, require significant

human effort, and inspectors need domain knowledge to detect the vulnerabilities they are trying to identify (Liu et al., 2012, p. 1). For this reason, static analysis tools exist.

Static code analysis tools are commonly used to detect bugs by analyzing source code without executing it (Louridas, 2006, p. 1). Static analysis tools still require labor work as their outputs must be verified by a human and cannot be fully automated (Liu et al., 2012, p. 1). Two typical problems of static analysis tools are false negatives and false positives. It is mathematically impossible to create an algorithm that can correctly determine the behavior of all possible cases (Alqaradaghi & Kozsik, 2024, p. 2). As a result of this limitation, false negatives are simply unavoidable (Alqaradaghi & Kozsik, 2024, p. 2; Liu et al., 2012, p. 2).

These tools can also produce false positives, where vulnerabilities are reported even though they don't exist. Because of this, manual expert evaluation is needed to verify and confirm whether the reported issues represent real vulnerabilities (Liu et al., 2012, p. 2).

2.2.2 Dependency-based vulnerability detection

Many modern software projects rely on third-party open-source libraries because they provide reusable components that speed up implementation and reduce effort. However, the increasing use of external dependencies also introduces additional security risks for software (Schott et al., 2026, p. 1). To identify vulnerabilities originating from components, software composition analysis (SCA) tools are commonly used.

SCA tools analyze software code dependencies, identify potential vulnerabilities from known vulnerabilities catalogues, and help developers to upgrade dependencies to newer versions (Dietrich et al., 2024, p. 77). While these tools are effective at detecting vulnerability packages, they often lack the ability to determine whether a vulnerability is actually exploitable or harmful to the system (Dietrich et al., 2024, p. 78).

A large-scale study by Zhang et al. (2024, p. 2) found that while thousands of projects used vulnerable libraries, only a fifth of them were actually threatened because the applications didn’t call vulnerable APIs. This confirms that most of the identified vulnerabilities remain theoretical because they are not reachable at the application level and therefore not exploitable in the project context.

Furthermore, current SCA tools often produce false positives because they are too generic and lack context awareness, leading to an increase in vulnerability alerts and making it difficult for developers to find critical issues in external components (Lin et al., 2025, pp. 2170–2176).

Theoretical vulnerabilities and false-positive alerts generate a large amount of non-actionable data. Ultimately, this type of information may lead to alert fatigue, which is a psychological state where developers overlook or ignore vulnerability notifications because they feel overwhelmed by the volume of information (Stanton et al., 2016, pp. 28–29).

These limitations suggest that dependency-based detection approaches are scalable but fail to distinguish between theoretical vulnerabilities and actual contextually exploitable risks. Consequently, more advanced analytical methods are needed to better understand how vulnerable components are used in modern software systems.

2.2.3 Dynamic analysis

Sometimes the software code needs to be executed to detect vulnerabilities that occur only at runtime. Dynamic analysis evaluates the system by running it and helps identify vulnerabilities during operation (Cong et al., 2024, p. 1). Dynamic analysis works by monitoring a program's behavior during execution, often by intercepting function or API calls to observe how it interacts with the operating system (Egele et al., 2012, p. 7).

Dynamic tools help to detect problems that static analysis might miss, such as performance bottlenecks, memory leaks, or other issues only detectable at runtime (De Silva et al., 2023, p. 1). A disadvantage of dynamic analysis is that it suffers from incomplete path coverage, only the code paths that are executed during analysis are observed, which means vulnerabilities in unreached paths remain undetected (Egele et al., 2012, p. 18).

Even though static and dynamic analyses have their own flaws, both are essential throughout the software development life cycle, and their quality continues to improve as new technologies and approaches emerge. The accuracy of code analysis can be enhanced by leveraging machine learning and artificial intelligence, making the results more reliable (Humran & Sonmez, 2025, p. 1). Furthermore, recent research has explored the use of artificial intelligence and large language models to support vulnerability detection through contextual reasoning and cross-procedural code analysis (Y. Li et al., 2025, pp. 1, 10).

2.3 LLM-based multi-agent systems

Intelligent agents are computer systems that are capable of operating autonomously within their environments to achieve specific goals (Wooldridge, 2009, p. 32). These agents are capable of reactive, proactive, and social behavior, and can select actions based on plans, goals, and their knowledge base, and may communicate to coordinate activities and cooperate dynamically (Wooldridge, 2009, p. 35).

Multi-agent (MA) systems are defined as systems of multiple agents that interact and can cooperate, coordinate, and negotiate with each other (Wooldridge, 2009, p. 16). According to Tran et al. (2025, p. 5), there are four key components in MA systems: agents, environment, interactions, and organization. In their framework, agents are represented as the primary autonomous actors within systems, as illustrated in Figure 4.

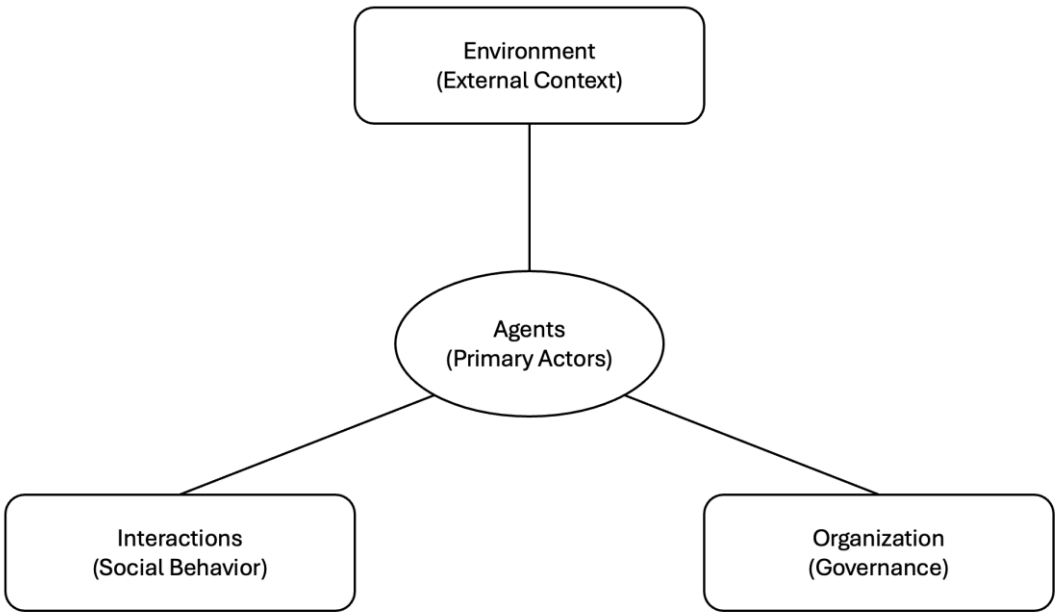

Figure 4. Core components of MA systems. Adapted from Tran et al. (2025, p. 9).

Agents are defined by their internal knowledge models and cognitive capabilities, such as reasoning, learning, and decision-making (Tran et al., 2025, p. 5). By environment, they refer to the external context in which agents operate, including sensory input and operational space for their actions and interactions (Tran et al., 2025, p. 6).

Interactions occur with communication mechanisms that are often supported by standardized communication languages, enabling complex social behaviors such as cooperation, negotiation, and coordination among agents (Tran et al., 2025, p. 6). And by organization, they refer to the system's structural governance, which can range from formal hierarchical control to decentralized structures shaped by spontaneous behavior.

Agents have different ways to cooperate and communicate with each other. They may cooperate toward shared goals, compete by pursuing conflicting objectives to maximize own interest, or combine both behaviors depending on the system design (X. Li et al., 2024, pp. 21–22). Their communication structures are categorized as centralized, where a coordinating agent oversees interactions, decentralized, allowing agents to

communicate directly without a central controller, or hierarchical, where tasks are distributed across levels (Tran et al., 2025, pp. 16–19). Furthermore, MA system architectures can be either static, with fixed agent roles and domain knowledge, or dynamic, where agents and interactions can adapt at runtime (Tran et al., 2025, pp. 19–20).

2.3.1 Multi-agent cooperation strategies

Multi-agent interactions and behavior depend on their cooperation strategy, which can be rule-based, model-based, or role-based, as illustrated in Table 3.

Table 3. Comparison of multi-agent cooperation strategies. Adapted from Tran et al. (2025), Strachan et al. (2024), and Wooldridge (2000).

Strategy	**Primary Driver**	**Best For**	**Limitation**
Rule-based	Predefined Rules	Stable Tasks	Low Flexibility
Model-based	Probabilistic Models	Social Tasks	Complexity/Cost
Role-based	Organizational Structure	Complex Workflow	Fixed Protocols

In rule-based cooperation, agents' behavior is governed by predefined rules rather than probabilistic or role-specific inputs (Tran et al., 2025, p. 13). This kind of cooperation strategy can be useful in situations where it is beneficial not to stray from a predefined path, such as debate and majority rule votes (Tran et al., 2025, p. 14). However, rule-based cooperation has limited adaptability to uncertainty and may fail in situations without predefined constraints (Tran et al., 2025, p. 14). Also, difficulties may arise when rule-based agents are scaled on complex tasks, as the number of rules can increase exponentially (Tran et al., 2025, p. 14).

Model-based cooperation uses probabilistic models such as the Theory of Mind (ToM) to enable agents to make decisions that are most likely to align with other agents. ToM is a central ability in human social interactions, where a human is capable of tracking other people's mental state (Strachan et al., 2024, p. 1286). Recent research on large language models suggests that LLMs can follow aspects of ToM when solving tasks that

require reasoning about the beliefs or perspectives of others (Strachan et al., 2024, p. 1286). However, their performance is often inconsistent and sensitive to prompt variations, which raises a question of whether these models should rely on learned linguistic patterns instead (Strachan et al., 2024, p. 1286). According to Tran et al. (2025, p. 16), model-based cooperation can be difficult to design and deploy because it requires complex models of the environment and agent interactions. Furthermore, their probabilistic decision-making might be computationally expensive (Tran et al., 2025, p. 16).

Role-based MA systems are built around organizational roles rather than individual agents and can be described as computational organizations composed of interacting roles (Wooldridge et al., 2000, p. 285). A role defines the expected behavior for specific agents within the system and is characterized by responsibilities, permissions, activities, and interaction protocols (Wooldridge et al., 2000, p. 289). Responsibilities specify what an agent must achieve, permissions define what kind of information or resources an agent may access, activities describe an agent's internal computations, and protocols define ways of interaction between roles (Wooldridge et al., 2000, pp. 289–290).

Building MA systems with agents that have clearly defined roles can improve system efficiency and support the organization, as agents can be experts in specific tasks and functionalities (Tran et al., 2025, p. 15). Also, this specialization increases modularity and reusability, as individual agents can be developed, implemented, and updated independently, thereby improving the system's overall performance (Tran et al., 2025, p. 15). However, conflicts or functional difficulties may arise within the system in coordination if agent roles are poorly defined or insufficiently specified (Tran et al., 2025, p. 15).

While traditional MA system research provided the foundation for agent interaction, LLMs have extended these concepts, making them autonomous participants capable of organic specialization. In modern LLM-based multi-agent systems, roles are often used to split responsibilities between agents such as planners, executors, critics, and experts (Vijayaraghavan et al., 2026, p. 2).

2.3.2 LLM-enhanced agent reasoning

Recent advances in LLMs have expanded the capabilities of intelligence agents. These agents use LLMs as their “brain” to sense the environment, make decisions, and take actions through multi-step reasoning (Minaee et al., 2025, p. 2). LLMs are typically pre-trained on various datasets, providing agents with a solid foundation for understanding specific topics (Tran et al., 2025, p. 8).

Combining pre-trained models with external knowledge sources and tools enables agents to communicate effectively with users and their environments, and to adapt their actions based on feedback, which can reduce the likelihood of hallucination (Peng et al., 2023, p. 1).

LLM-MA systems have enhanced traditional system communication architectures by introducing structured coordination mechanisms between agents. According to Guo et al. (2024, pp. 4–5) and Li et al. (2024, pp. 21–22), four common communication architectures are used in LLM-MA systems, as illustrated in Figure 5.

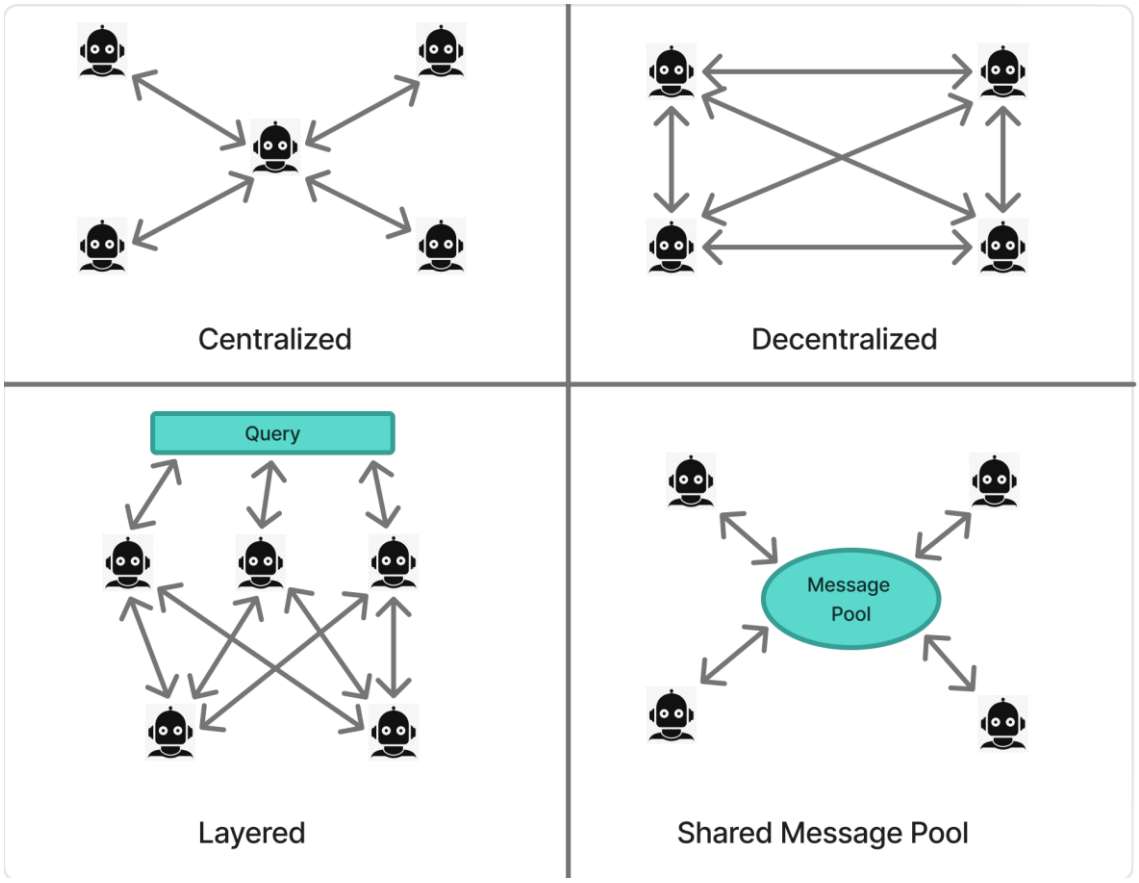

Figure 5. The agent communication structures. Adapted from Guo et al. (2024, p. 4).

These four structures differ mainly in how control and information flow between agents. In a layered structure, work moves in one direction from one level to the next, which suits for tasks that can be split into clear stages. A decentralized structure lets every agent talk to every other agent directly, fitting situations where agents have similar capabilities and no single agent should be in charge. A centralized structure places one agent at the middle of the communication, giving the system a clear point of control but also a single point of failure. Finally, a shared message pool removes links between agents and instead lets them read and write to a common space. Which structure is appropriate depends on the nature of the work the system is built to carry out.

2.3.3 Memory capabilities on LLM-based agents

A key feature of LLM agents is their memory system, inspired by human cognitive processes, in which information is encoded in sensory memory, then transferred to short-term memory, and eventually consolidated into long-term memory (Jia et al., 2026, p. 22). These agents use memory modules to gather experiences, act rationally and effectively, and adapt their actions by referring to past interactions and knowledge.

Short-term memory (STM) functions as temporary storage for LLMs, typically holding recent inputs within the context window, which represents the span of contextual information a model can use to process data and generate outputs (Jia et al., 2026, p. 23; Shan et al., 2025, p. 6). The context window is typically measured in tokens, and its size greatly influences the model's performance. When the maximum token limit is reached, LLMs might lose information beyond their token capacity (J. Guo et al., 2024, p. 4). This means that a larger context window helps the model maintain coherence during longer conversations and manage multiple dialogue threads more effectively (Shan et al., 2025, pp. 6–9). Another part of STM is working memory, which has an important role in multi-step reasoning and decision-making, such as planning (Shan et al., 2025, p. 6). With working memory, LLM agents can hold multiple ideas in mind simultaneously, enabling them to process data across multiple inputs.

Long-term memory (LTM) serves as an external storage system that agents access via queries, keeping relevant data accessible and enabling them to recall information when needed (Jia et al., 2026, p. 23; Shan et al., 2025, p. 7). LTM can be categorized into explicit and implicit memory. Explicit memory allows the agent to dynamically capture context-aware information and to collect knowledge from external memory, such as databases, vector stores, and graph structures, including facts, rules, and structured and unstructured knowledge (Jia et al., 2026, p. 14). Implicit memory enables an agent to develop instincts, such as retaining and recalling the steps needed to complete tasks, functioning unconsciously (Shan et al., 2025, p. 7).

2.3.4 Enhancing the capabilities of LLMs with tools

LLMs cannot access knowledge beyond their training data. To overcome this, their capabilities can be extended through external tools such as Retrieval-Augmented Generation (RAG) and MCP. With these external tools, LLMs can attain higher accuracy on complex tasks and enhance their overall performance (Shen, 2024, p. 1).

The limitations of knowledge can be addressed by using RAG. RAG allows LLMs to access relevant data from external sources by querying databases and incorporating this information into their responses to fill gaps in its pre-trained knowledge (Lewis et al., 2021, p. 2). This approach improves accuracy and provides access to domain-specific, up-to-date information, making it well-suited for question-answering systems and enterprise knowledge retrieval (Fan et al., 2024, p. 8).

Another promising technology is the MCP, introduced by Anthropic (n.d.-b), which standardizes how AI applications access external tools and services. Traditionally, LLM-based systems relied on direct API integrations, in which each service was connected individually (see Figure 6), resulting in complex, less scalable system architectures (Hou et al., 2025, p. 4).

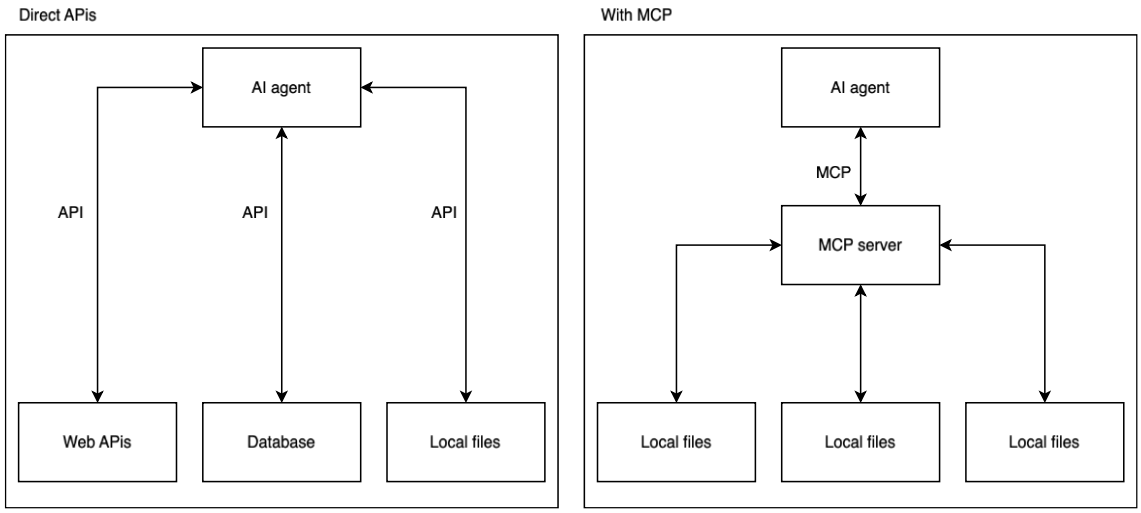

Figure 6. Comparison of AI agent communication using direct API calls and the MCP. Adapted from Hou et al. (2025, p. 3).

MCP provides a unified protocol that enables dynamic connections to multiple tools without requiring separate integrations for each service (Hou et al., 2025, p. 4). Additionally, many third-party services now provide prebuilt MCP interfaces, enabling faster integration.

Furthermore, to enhance MCP connectivity, Anthropic released agent skills that prepare agents for specific tasks (Anthropic, n.d.-a). These skills are documented in a markdown file, where users define the agent's behavior and the connections to utilize in natural language. Essentially, skills dictate "what to do," while MCPs determine “how to connect” (Xu & Yan, 2026, p. 2). With these skills, the agent can determine which MCPs to use, what kind of problem it is addressing, and how it should approach the task. This interaction between the agent, skills, and the MCP layer is illustrated in Figure 7.

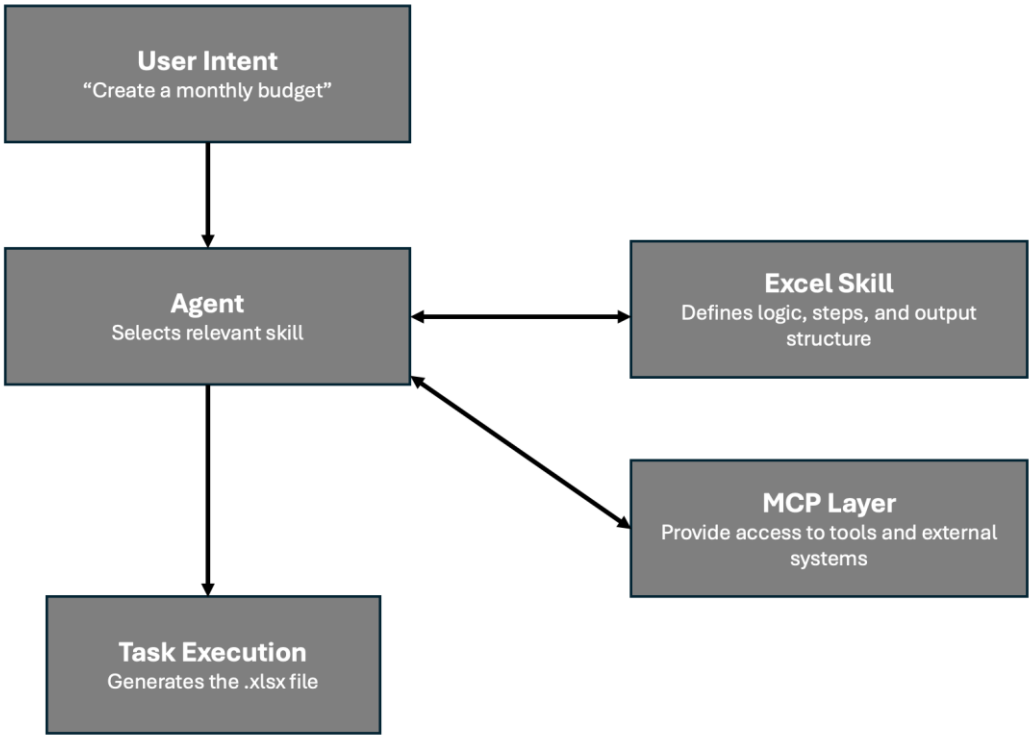

Figure 7. Interaction between agent, skills, and MCP layer in task execution. Adapted from Xu & Yan (2026, p. 6).

As shown in Figure 7, the agent starts by processing user intent, then it selects relevant skills, interacts with the MCP layer, and finally generates structured output.

3 Existing approaches to automated source code analysis

In this chapter, various LLM-based vulnerability analysis solutions and case studies are examined, focusing on their workflows and results, particularly false positives. To ensure relevance, this chapter reviews only research published after 2024.

Research on automated source code analysis has evolved from traditional rule-based static analysis towards more advanced solutions in which LLMs are part of the process. Current methods for LLM-based vulnerability detection fall into two main categories: fully LLM-driven solutions, where the model handles the entire analysis process, and hybrid approaches that combine LLMs with traditional static analysis tools to improve detection effectiveness.

3.1 Implemented fully LLM-based approaches

Fully LLM-based approaches perform vulnerability detection without relying on traditional static analysis tools. Commonly, this is achieved with contextual reasoning, prompt engineering, and multi-agent coordination to analyze source code and identify potential vulnerabilities.

An example of fully LLM-based vulnerability detection is a multi-agent framework proposed by Li et al. (2025). The framework uses contextual reasoning and interactive refinement to support the process and consists of role-based agents that collaborate through iterative feedback loops to improve detection accuracy (Y. Li et al., 2025, p. 1). The system comprises three agents: the architecture agent, analyst agent, and evaluation agent (Y. Li et al., 2025, p. 4).

Results show that employing contextual reasoning allows for more accurate identification of the vulnerability's location within the system. Additionally, using the evaluation agent yields a lower error rate than other systems (Y. Li et al., 2025, p. 10).

Another example is presented by ZeMicheal et al. (2024) who introduced a multi-agent system that leverages LLMs for vulnerability analysis in container environments. This approach employs Plan-and-Execute reasoning and comprises three core parts: a checklist generator, an LLM agent planner, and components for justifying and summarizing the results into human-readable text (ZeMicheal et al., 2024, pp. 3–6).

The results demonstrate the effectiveness of streamlining vulnerability analysis tasks and help eliminate the need for manual verification of CVEs, allowing developers to review only the outcomes after agents have completed their actions (ZeMicheal et al., 2024, pp. 1, 11). However, the results also indicate that the system's accuracy is not perfect and requires fine-tuning to improve its performance (ZeMicheal et al., 2024, pp. 10–11).

3.2 Implemented hybrid approaches combining LLMs and static analysis

While some approaches rely fully on LLM reasoning, others integrate LLMs with traditional techniques. For example, Huang et al. (2025) propose a system that combines static analysis with the reasoning capabilities of LLMs. Their method generates custom prompts to guide LLMs in dependency analysis and chain-of-thought reasoning. Results show that this solution outperforms traditional vulnerability detection methods in source code analysis (Y. Huang et al., 2025, p. 1).

Similarly, Chen et al (2025, p. 1) introduce a process in which static analysis is first used to identify relevant code lines, and LLM applies detailed chain-of-thought prompting to refine the analysis. Their results show that detection accuracy improves when static code analysis is combined with LLM reasoning (Chen et al., 2025, p. 1). However, Armando et al. (2026, p. 1) found that LLMs combined with static application security testing (SAST) tools do not yet outperform traditional static analysis tools. Their findings suggest that LLMs can improve results, especially for business-critical software, the best performance is still achieved by traditional SAST tools.

The LLM's ability to understand context can improve with prompt engineering, a practice of creating detailed, more effective prompts that yield more accurate outputs. Highly detailed prompts can be used as reusable templates to guide LLM behavior and improve output quality. For example, Daneshvar et al. (2024, pp. 1–2) propose an approach where RAG retrieves code samples and incorporates them into prompt templates to generate new vulnerable code samples. This helps address cases where these models lack sufficient examples of vulnerable code, which negatively affects their detection performance (Daneshvar et al., 2024, p. 2). Their results show that vulnerability detection models benefit from training data that is more diverse and enriched with retrieved contextual information (Daneshvar et al., 2024, p. 14).

3.3 Improving vulnerability detection with external knowledge

RAG can also be used to collect data on historical vulnerabilities and their fixes. Du et al. (2024) propose a multi-dimensional framework that leverages historical vulnerabilities and their corresponding fixes to enhance LLM-based detection accuracy. Their approach builds a knowledge base by extracting vulnerability causes, functional semantics, and fixing solutions from CVE data, which is then retrieved to guide the analysis (Du et al., 2024, p. 2). The results show that this approach significantly improves vulnerability detection and enables the identification of previously unknown vulnerabilities (Du et al., 2024, p. 1).

The model's vulnerability-detection performance can be improved through fine-tuning, a process in which the model is trained on specific data. Yang et al. (2026) demonstrate that integrating fine-tuned LLMs into a multi-agent vulnerability analysis system improves both detection accuracy and efficiency, and identifies previously undetected vulnerabilities (Yang et al., 2026, p. 1).

Another way to improve models' detection performance, besides fine-tuning, is context-aware analysis, in which the model incorporates the broader code context. For example, Qiu et al. (2026) introduce a method in which LLM-based vulnerability detection leverages project-level contextual information to enhance its accuracy. Their approach focuses on code dependencies and relationships between components, allowing more accurate detection analysis compared to isolated code evaluation (Qiu et al., 2026, p. 1). The results show that providing contextual information reduces misinterpretation of code and enhances detection accuracy (Qiu et al., 2026, p. 1).

3.4 Main insights from current solutions

LLM-based vulnerability detection approaches can be grouped into several categories, including multi-agent systems, prompt engineering, context-aware analysis, retrieval-augmented generation, hybrid analysis, and model optimization techniques. Table 4 highlights their core ideas and key contributions.

Table 4. Summary of LLM-based vulnerability detection approaches.

Approach	Core Idea	Key Contribution	Author
Multi-agent approaches	LLM-based agents collaborate through context reasoning and iterative refinement	Improves detection accuracy and enables advanced reasoning	Li et al. (2025); ZeMichael et al. (2024)
Prompt Engineering	Uses structured prompting techniques such as chain-of-thought, in-context learning, and prompt templates	Enhances vulnerability detection accuracy without model retraining	Chen et al. (2025); Daneshvar et al. (2024)
Context-Aware Analysis	Enhances project-level contextual information into LLM-based vulnerability detection	Improves detection accuracy by reducing isolated code interpretation	Qiu et al. (2026)
Retrieval-Augmented Generation	Integrates external knowledge sources such as vulnerability databases into LLM reasoning	Enhances detection accuracy and reduces hallucination	Du et al. (2024)
Hybrid Analysis	Combines traditional static analysis tools with LLM-based reasoning	Reduces false positives by leveraging complementary methods	Huang et al. (2025)
Fine-tuning	Adapts LLMs vulnerability detection by using domain-specific datasets	Improves performance in specialized domains	Yang et al. (2026)

Despite recent advancements in automated source code vulnerability detection, they still face several limitations. A key challenge is the high number of false positives, which increases the workload for developers and reduce trust for automated tools. Traditional static analysis tools, such as SonarQube, continue to play an important role in the SDLC and are widely used in the industry on a daily basis. However, these tools rely on predefined rules and often cannot determine whether a detected vulnerability is truly exploitable, often resulting in a high number of false positives.

A recent study by Ni et al. (2025, pp. 9–10) shows that current LLM-based methods are not as effective at vulnerability detection as alternative methods. They perform worse than alternative learning models, including sequence-based and graph-based approaches, emphasizing the necessity to develop a dedicated vulnerability detection

model. But this doesn't mean using LLMs is unsuitable for this task. Ni et al. (2025, p. 19) also found that their ability to leverage contextual understanding and natural language reasoning, combined with effective prompt design, can be helpful. Furthermore, existing research emphasizes using RAG to include external knowledge, but less focus has been given to newer methods like the MCP for retrieving vulnerability data dynamically and in more a structured way.

LLM-based multi-agent systems are a promising new technology, but they also face several limitations. According to Guo et al. (2024, p. 10), LLM-MA systems encounter challenges in multi-modal environments, where agents must process and integrate diverse data types beyond textual information, making it difficult for them to understand one another and respond to complex, non-textual information. Another significant issue is hallucination, where LLMs produce false or misleading information (L. Huang et al., 2023, p. 2). Hallucination can be particularly severe in layered systems, where hallucinations produced by one agent may propagate to others, which can lead to cascading errors (T. Guo et al., 2024, p. 10). This shows that detecting and mitigating hallucinations is critical at both the individual-agent level and the overall information-sharing level between agents.

In a study by Noever (2023), LLMs were compared to Snyk, a popular static code analysis tool used by software developers that provides detailed vulnerability reports. The study found that GPT-4 detected four times more vulnerabilities than Snyk, suggesting that large language models are more advanced and capable of understanding and reasoning on a higher level than traditional static code analysis tools (Noever, 2023, p. 6). This implies that LLMs could serve as valuable tools for automating code analysis in complex systems. However, another study found that some of these advanced models achieved less than 50% accuracy, suggesting that even a random guesser could outperform them (Steenhoek et al., 2024, p. 6). This highlights the need for fine-tuning these models, which could significantly improve their accuracy and overall effectiveness.

Additionally, LLMs' internal reasoning processes are not fully transparent, making them still considered as black-box systems. This lack of transparency makes it difficult to understand how these models arrive at their conclusions and complicates the evaluation of their reliability in complex systems (Taghavi Far & Feyzi, 2025, p. 60). Furthermore, increasing the number of agents doesn't always improve performance. In sequential reasoning tasks, multi-agent systems have been shown to reduce performance by 39-70% compared to a single agent baseline (Kim et al., 2025, p. 1).

Another important consideration for LLM-MA systems is their high computational cost, which is tied to token usage. Recent studies show that prompt and output token lengths have increased significantly, as LLMs are being used for more complex, content-rich tasks (Aubakirova et al., 2026, p. 15). Since LLM usage is priced based on token consumption, this directly results in higher operational costs (Aubakirova et al., 2026, pp. 28–30). Consequently, selecting an appropriate model is an important factor in cost efficiency, as there are significant differences among models in pricing and token consumption (Aubakirova et al., 2026, p. 31).

4 Design science research approach

This study is conducted using the DSR methodology. The DSR will produce an artifact that will be demonstrated through a PoC, and its functionality will be evaluated. This study will follow the framework by Peffers et al. (2007).

The goal of design science is to solve problems by producing an artifact. According to Hevner et al. (2004, p. 91), the core idea of DSR is to create unique artifacts and prove their usefulness with evidence. The artifacts must reflect real-world contexts, and there cannot be existing solutions, or else the artifacts are irrelevant, meaning they don't provide any utility, as other solutions already satisfy the problem (Hevner et al., 2004, p. 91).

4.1 DSR knowledge and guidelines

According to Gregor & Hevner (2013, p. 343), DSR knowledge can be divided into *descriptive* and *prescriptive* knowledge, with descriptive knowledge focusing on explaining the world as it is, covering insights into natural phenomena, their characteristics, and the principles that govern them. Prescriptive knowledge, on the other hand, guides action and provides knowledge of how to create and apply artifacts to practical problems (Gregor & Hevner, 2013, p. 343).

The design science has often relied on descriptive research methods borrowed from the social and natural sciences, which primarily produce explanatory knowledge rather than applicable, solution-oriented artifacts. To address this, Peffers et al. (2007, p. 46) argue that a common framework and a shared mental model are necessary to guide researchers in both conducting and evaluating DSR.

Similarly, Hevner et al (2004, p. 83) propose seven guidelines to support the conduct and evaluation of effective design science research. The guidelines emphasize that DSR should produce a viable and relevant artifact, address a clearly defined problem, provide

demonstrated utility, contribute novelty, be grounded in rigor and internal consistency, use a structured search process for solutions, and communicate results effectively to both academic and professional audiences (Hevner et al., 2004, p. 83). The DSR approach is well-suited to this study's research problem, as it addresses a real-world issue in which stakeholders seek practical solutions grounded in theory.

4.2 Research process

This study follows the design science research methodology presented by Peffers et al. (2007), which comprises six core activities. The process is illustrated in Figure 8. The first step of the methodology is to identify the problem and its motivation by defining the research problem and justifying why and how the solution works (Peffers et al., 2007, p. 52). In this study, the problem is the limited ability of traditional static analysis tools to reason about the context and exploitability of the detected vulnerabilities, and the high cost of commercial alternatives that attempt to close the gap.

The next step is to establish objectives for the solution, whether quantitative or qualitative. The aim is to determine what is achievable and realistic within the project's scope (Peffers et al., 2007, p. 55). In this study, the objective was to design an architecture for an AI-based multi-agent system that combines a static analysis scanner with external vulnerability intelligence and produces a structured vulnerability report.

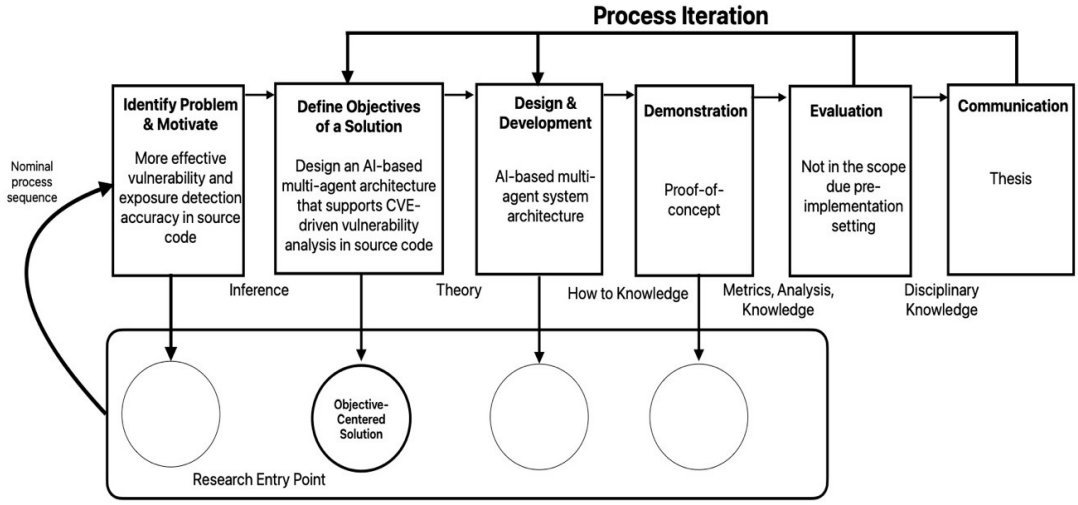

Figure 8. The design science research methodology process. Adapted from Peffers et al. (2007, p. 54).

The third step is to create the artifact, which can be any designed object (Peffers et al., 2007, p. 55). In this study, the artifact is the multi-agent system architecture that consists of four specialized agents, is coordinated by an orchestration engine, and leverages MCP servers for external capabilities.

The fourth step is to demonstrate the feasibility of the proposed artifact and show how it solves the problem (Peffers et al., 2007, p. 55). In this study, the artifact was demonstrated through a PoC implementation that was executed across five iteration rounds, first against a purpose-built test repository and finally against the real-world pyLoad project.

The fifth step is to evaluate the artifact by comparing its objectives to the actual results of its use. In this study, a formal evaluation was not conducted. Instead, observations gathered during the five demonstration rounds served as a lightweight evaluation, allowing each weakness detected by a run to inform the next version of the artifact.

The final step is to communicate the problem, artifact, and its utility to the audience. In this study, communication is handled through this thesis, which documents the problem, design decisions, the iteration history, and the final artifact.

Although the activities are presented in a structured order, the process is iterative and flexible, allowing researchers to start at any step, depending on the context, which can be problem-centered, objective-centered, design- and development-centered, or context-initiated (Peffers et al., 2007, p. 56). For example, the researcher may return to step 3 if the artifact requires improvements before moving forward (Peffers et al., 2007, p. 56). The corresponding key activities are described in Table 5.

Table 5. Key activities followed the design science research process. Adapted from Peffers et al. (2007).

Common Design Process Elements (Peffers et al., 2007)	**Key Activities in This Study**
Problem identification and motivation	Analyze challenges in vulnerability management; conduct a literature review on LLM-MA systems fixing this problem and establish the research gap
Definition of the objectives for a solution	Design the architecture of LLM-MA system for vulnerability detection, specify agent roles and capabilities
Design and development	LLM-MA system architecture for vulnerability detection process enhanced with static analysis tool
Demonstration	Demonstrate solution by building a PoC prototype
Evaluation	Formal empirical evaluation is excluded, as the PoC demonstrates feasibility
Communication	Applied through the documentation and reporting of the research outcomes in this thesis

Chapter 5 presents the resulting artifact and its iterative development across five iterations rounds, followed by the final version of the multi-agent system architecture.

5 Artifact development

In this chapter, the artifact is developed following the DSMR framework by Peffers et al. (2007). The proposed solution will be built by acquiring prior knowledge from existing LLM-based vulnerability detection solutions. However, step 5, evaluation, is excluded from the solution due to project scope. The solution's feasibility will be demonstrated by building a PoC prototype, and the process and results will be documented.

5.1 Entry point: objective-centered solution

Following Peffers et al. (2007, p. 56), this study enters the DSR process at the objective-centered solution point. The objective is to design an LLM-MA system architecture for automated vulnerability analysis, set in advance and grounded in a practical concern identified within a Finnish digital engineering and consulting firm. Existing commercial solutions come with expensive license costs and limited room for adaptation. This motivated the development of an in-house pipeline that integrates a static analysis tool with language-model agents and is configured to the organization's own development workflow.

5.2 Problem identification and motivation

Traditionally, software development primarily relies on static analysis tools such as SonarQube to identify vulnerabilities and dependency issues. However, these tools cannot determine if their findings are false positives, that is, whether the source code is truly at risk or if the issue is harmless and ignorable (Liu et al., 2012, pp. 1–2). These alerts increase the effort for developers, who must manually review and distinguish false positives (Tymchuk, 2017, p. 1). The results suggest a demand for more efficient, context-aware solutions that go beyond static rule-based pattern matching and offer greater modularity and adaptability to organizational workflows.

5.3 Objectives of a solution

A better artifact would not only flag potentially vulnerable code but also reason about its surrounding context, distinguish exploitable findings from harmless ones, and present the results in a form that a developer can act on without further analysis. It would treat the static analyzer as one component among several rather than as the sole authoritative source. This analyzer's output is then enriched with public vulnerability data and code-level reasoning, and gathered into a structured report.

The proposed solution defines distinct agent roles and external integrations for the objectives stated in 5.1. The artifact is a system architecture, demonstrated via a PoC prototype. In terms of DSR, it is an instantiation that demonstrates how multiple specialized agents can coordinate reasoning, retrieve external vulnerability information, and analyze source code in a structured way.

The solution defines agent roles for retrieving vulnerability data, analyzing source code, reviewing the architectural context, and generating structured reports. These agents communicate via a role-based coordination strategy. External knowledge sources, public CVE and CWE catalogue are integrated through MCP servers to ensure that the analysis remains aligned with current vulnerability information. The utility of the proposed solution can be evaluated by examining whether the system successfully identifies relevant vulnerability patterns and produces explainable analysis outputs.

The central requirements of the solution are adaptability to new vulnerability information, scalability to large and complex repositories, and structured support for human security analysis. It is intended for use by software development organizations to help identify and address software vulnerabilities more effectively in dynamic, knowledge-intensive environments.

The contribution is not only the prototype itself but also insights into agent roles, coordination strategies, and the integration of external CVE/CWE data through MCP servers. The main limitations are that the solution is small-scale and that LLM behavior can vary with prompts and model settings. Additionally, LLMs may produce plausible but factually incorrect outputs (hallucinations).

5.4 Design and development

The design and development phase creates the artifact that is the LLM-MA system architecture for vulnerability analysis. The proposed artifact features specialized agents, each with a distinct focus, communicating in a layered manner. The analysis process is divided into sequential phases: data retrieval, code analysis, and report generation.

A single-agent system was considered, but a single LLM's context window can quickly be exhausted during iterative reasoning. Once exceeded, the model begins to forget earlier reasoning, potentially causing repetitive thought processes and unpredictable outcomes (Y. Li et al., 2025, p. 3). Distributing the work across multiple specialized agents has been proposed in the literature as a way to mitigate this limitation, since each agent operates within a smaller, role-specific context (T. Guo et al., 2024, p. 3).

For this reason, the architecture proposed in this thesis is multi-agent rather than single agent. Dividing tasks across agents reduces the amount of context each agent must handle and keeps the reasoning process more stable.

5.4.1 Core architectural components of the LLM-MA system

The proposed multi-agent system architecture comprises four core components, each interconnected to maximize the effectiveness of the vulnerability detection process, as illustrated in Figure 9. A central component to manage the sequence of agent execution

and the flow between them is an orchestration engine. The orchestration engine is the backbone of LLM applications. It enables LLM agents to interact and share knowledge with each other. By using an orchestration engine, these applications can scale, manage resources, balance loads, and automate workflows.

The second component is a specialized agent's layer, comprising autonomous agents, each with a distinct role and a set of skills for a specific purpose. The third component is a tool layer that works as a standardized interface for agents to interact with external data sources and tools, such as MCPs. In this study, selected MCP tools provide access to SonarQube, CVE databases, and the files of the selected repository, where vulnerabilities exist.

Finally, the fourth component is the execution environment where the entire system is deployed and run. It can be an isolated environment, such as Azure Pipelines or a container system like Docker. Together, these components create a foundation for a multi-agent system.

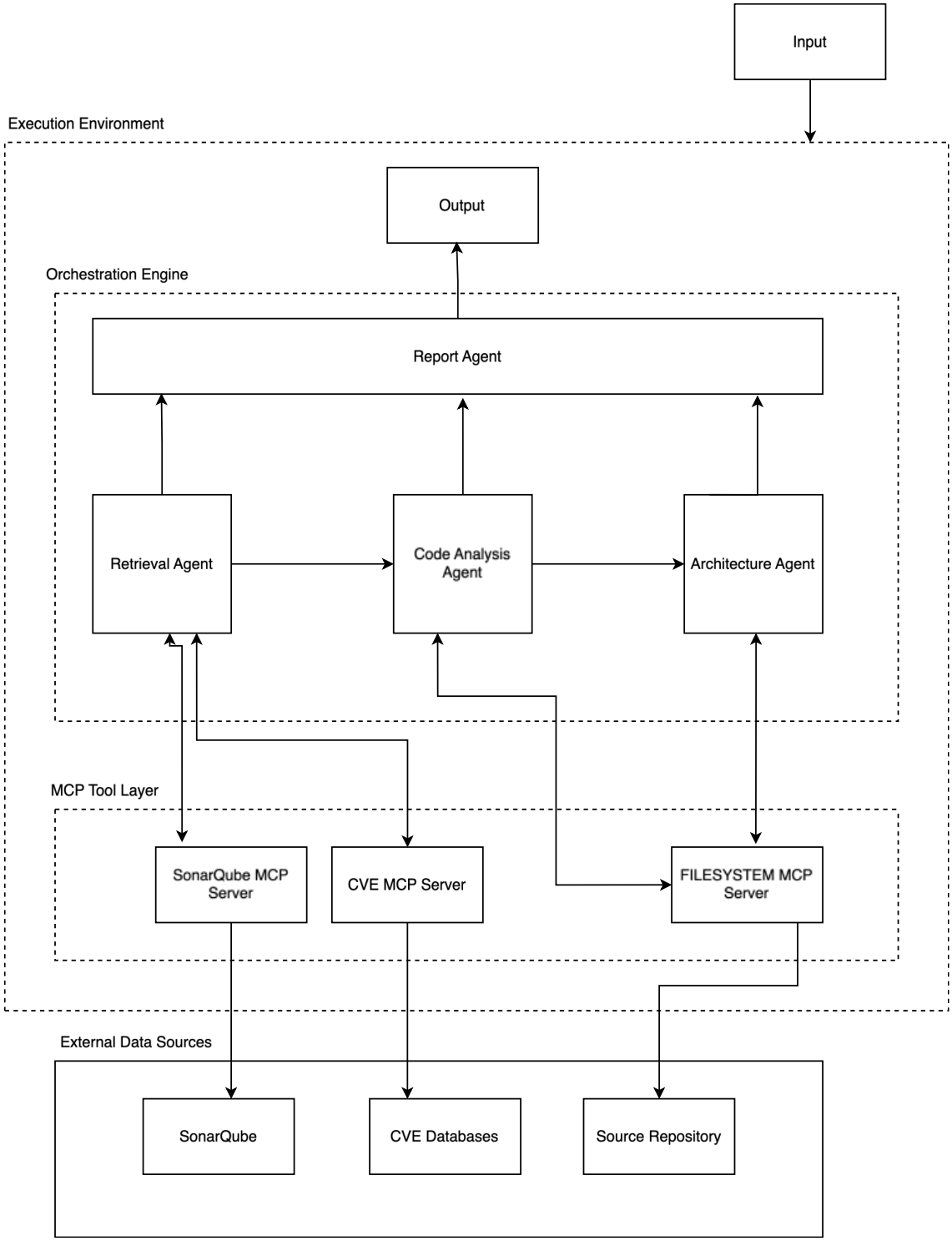

Figure 9. Overview of the multi-agent system architecture.

As Figure 9 shows, these four components form a single pipeline with a clear direction of data flow. The pipeline takes a target source-code repository that has been pre-

scanned with SonarQube as input. SonarQube is invoked before the pipeline starts, reads the target repository, and uploads the results into the SonarQube server. Inside the execution environment, the orchestration engine acts as the entry point of the run. It manages how agents are invoked and maintains a shared state object that carries data between them. The agents call tool layer via MCP servers, which gives them appropriate API or filesystem calls and return the results back to the agent. After all agents have contributed to the shared state, the pipeline produces a structured Markdown report as output.

5.4.2 Specialized agents and their responsibilities

As illustrated in Table 6, the multi-agent system uses four specialized agents that import SonarQube findings, enrich them with CVE data, analyze surrounding code, identify architectural patterns, and output a structured security report. Each agent uses an LLM tailored to their specific task, considering the required reasoning complexity. They utilize detailed prompt templates and specific skills to enhance their reasoning accuracy. The agent's skills are documented in a standardized markdown file. The skills always start with metadata, followed by details of the skill, defining their core responsibilities, key skills, tools used, and expected outputs.

Table 6. Responsibilities, tools, and outputs of vulnerability detection agents.

Agent	Responsibility	Tools	Output
Retrieval agent	Handles SonarQube response formats, extracts CVE ID, fetches CVSS score	SonarQube MCP, CVE MCP Server	Enriched vulnerability records with CVSS
Code Analysis agent	For each vulnerability, reads the affected file, and surrounding code, identifies the vulnerability code pattern, data flow, existing mitigations	Filesystem MCP	Code context including issue id, code snippets, and risk assessment
Architecture agent	Examines the project structure, dependency files, configuration files, and architectural patterns	Filesystem MCP	A summary object with details of the issues and an overall architectural assessment
Report agent	Synthesizes the outputs from all previous agents into a structured, human readable report	None	A structured document containing an executive summary and code snippets

The Retrieval agent: Serves as the primary data input for the analysis pipeline. It collects report from SonarQube and provides more information to make the report more comprehensive. It gathers findings, extracts identifiers, and fetches vulnerability records with CVSS, exploitability, and remediation notes. It enhances its knowledge by utilizing the CVE MCP Server, which collects pertinent CVE data. For each issue, it references the CVE ID and retrieves its CVSS score and exploitability metrics. The complete prompt and skill description for the Retrieval agent are reproduced in Appendix 2.

The Code Analysis agent: Reads the affected file and the 50 lines of code around it. In code-level analysis, it identifies vulnerable code patterns, data flow, whether inputs are user-controlled, and existing mitigations, such as try/catch blocks. The agent's output is a detailed analysis of the issues, including relevant code snippets. The complete prompt and skill description for the Code Analysis agent are reproduced in Appendix 3.

The Architecture agent: Examines the project structure and takes a high-level view of the project. It detects broad patterns on dependency files, configuration files, and architectural patterns that code-level analysis might miss. It identifies what frameworks are being used, common security patterns, and whether vulnerabilities are present in the libraries in use. The complete prompt and skill description for the Architecture agent are reproduced in Appendix 4.

The Report agent: Final agent in the pipeline that synthesizes all agent outputs into a structured report, prioritizing findings by CVSS, exploitability, and architectural exposure. It outputs an executive summary of the detailed findings, including code snippets.

5.4.3 The system's workflow and data flow

The proposed solution works as part of a continuous integration and deployment (CI/CD) pipeline, serving as a post-processing layer for static analysis results. Figure 10 illustrates the integration as sequential workflow that begins on the developer side and ends with the delivery of structured security report back to the same side.

The workflow is triggered when a developer pushes code changes to a source-code repository. The pipeline first performs build and test activities, including compilation and validation. Then SonarQube performs static analysis to detect potential vulnerabilities and code quality issues. After the static analysis tool detects a new vulnerability, the multi-agent system begins to work on the problem.

The LLM-MA system employs role-based agents, with communication organized into layered coordination where agents communicate with others at the same or lower level. First, the Retrieval Agent connects to the CVE MCP server to fetch the latest information about the CVEs. Then, the Code Analysis agent and Architecture agent use the filesystem MCP, which provides a comprehensive understanding of the project and enables more accurate detection analysis (Qiu et al., 2026, p. 1).

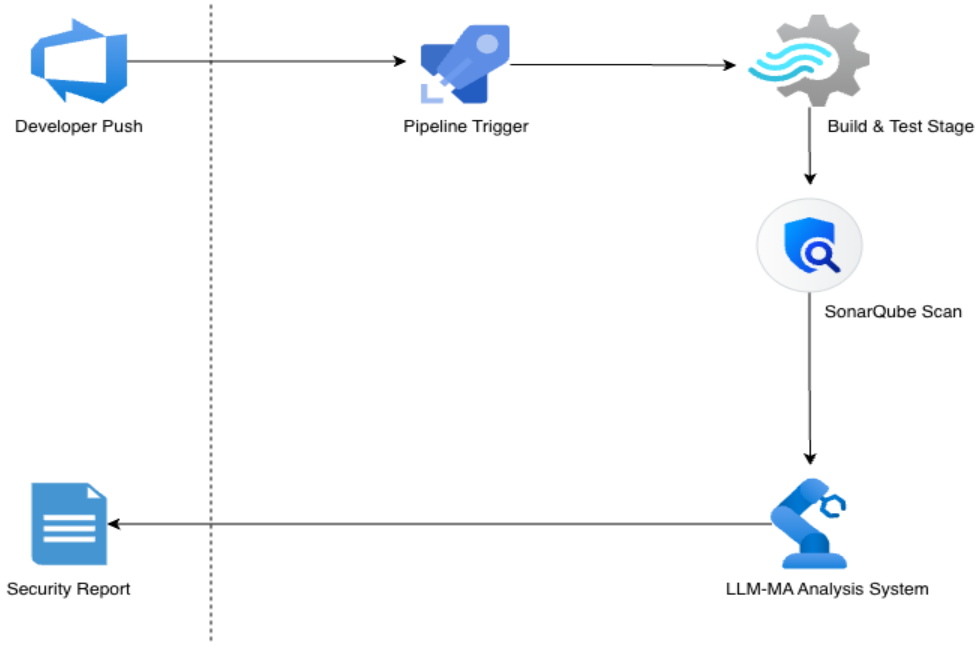

Figure 10. Integrating the LLM-MA analysis system artifact into the CI/CD pipeline.

The results from the previous analysis and all data collected from each agent are provided to the Report agent, which consumes the results from previous agents and synthesizes them into a security report describing the vulnerable code, its severity, and remediation guidance. The completed report is delivered back to the developer side of the pipeline, either as an artifact attached to the build run or as status message on the pull request.

5.5 Demonstration

To demonstrate the viability of the proposed artifact, a prototype is created. The prototype will be built and executed locally, without using the Azure pipeline. The most critical part of the solution is to demonstrate that SonarQube analysis works with the agent system's logic. The prototype runs locally using the Claude Code CLI for agents and hosts SonarQube in a Docker container. The multi-agent system has been built using Python. The system leverages Claude Code CLI's non-interactive mode to execute each agent as

a subprocess. This approach uses existing Claude subscription for authentication and model access, without a need for separate API keys or per-token billing. The primary model for all agent reasoning is Claude Opus 4.7, but Claude Code additionally uses Claude Haiku 4.5 as a helper model for internal support tasks such as tool output processing.

Each agent invocation passes a structured prompt via stdin and receives the model's response via stdout in JSON format. The Claude Code CLI also manages MCP server lifecycle, it spawns and connects to the configured MCP servers automatically based on configuration file. Each agent has been configured with a maximum number of turns (agentic loop iterations), where each run includes evaluating the current context, optionally invoking a tool, and receiving the tool's results. This parameter acts as a resource constraint that prevents unbounded tool invocations and ensures that agents have sufficient iterations to complete their tasks.

The demonstration will be conducted in two stages. First, tests will be performed on a controlled target to confirm that the agent's behavior functions as expected against ground truth. Next, the focus shifts to a naturalistic target, an open-source project called pyLoad, consisting of 577 Python files. The purpose is to evaluate the system against real-world, unmodified code to assess external validity and scalability.

5.5.1 Round 1 – Baseline and CVE misattribution discovery

The first pipeline run targeted a purpose-built test repository (vulnerable-test-app) containing twelve known vulnerabilities. The objective was to verify that SonarQube could detect all issues, and to observe how agents processed these results, whether they could enrich findings, find new ones, or flag false positives.

The system behaved exactly as designed and transformed raw SonarQube alerts into a synthesized report. However, the Retrieval agent incorrectly attributed CVEs. It selected

a real CVE record with matching CVSS metadata but assigned it to findings outside its actual scope. In one case, the agent assigned CVE-2026-3320, a vulnerability affecting a digital photo frame, to an “active debug code” finding in the Flask application because both records share the CWE-798 category but have no product relationship. The failure is therefore a misattribution, not a hallucination, as every referenced CVE existed and had a correct CVSS score. This observation motivated the round 2 design change, in which the Retrieval agent’s prompt was edited to require product-level correspondence before a CVE could be assigned to any finding. Overall, the report was of good quality, featuring an executive summary, clearly structured findings, and a remediation roadmap, all aligned with the prompt template.

5.5.2 Round 2 – Attribution accuracy fix

Round 2 updated the Retrieval agent prompt to include a critical rule to only map CVEs with matching dependencies. The agent correctly enriched all twelve findings and didn’t assign any CVE IDs, since none of the issues corresponded to a known vulnerable library version. The agent’s own justification was:

> "*None contain explicit CVE IDs... Per the CRITICAL attribution rule, I will not assign CVE IDs to these findings — these are generic Flask-application code weaknesses (hardcoded secrets, insecure temp files, missing cookie flags, etc.), not vulnerabilities in specific library versions that match known CVEs.*"

The critical rule prompt change eliminated the misattribution problem entirely, and the agent now requires an actual product/dependency match before assigning a CVE, and recognized that a matching CWE category is not sufficient alone.

The Code Analysis agent likewise produced a solid output. It read each source file, extracted the relevant snippet, and analyzed the vulnerability pattern rather than repeating the SonarQube message. It also traced data flow across findings. For example, it

reasoned the interaction between the reflected-XSS endpoint (/search=?<script>) and the missing HttpOnly cookie flag, concluding that the injected script could read session cookies because HttpOnly was disabled. The agent was also overriding SonarQube's severity rating with a written justification. As illustrated in Table 7, three cases stood out.

Table 7. Severity rating overrides by the Code Analysis agent (round 2).

Finding	SonarQube severity	Agent
app.run(debug=True)	Low	Critical
MD5 password hashing	Low	High
Cleartext HTTP to payment API	Low	High

In each case, the agent provided an explanation that tied the code-level weakness to a real-world impact, which SonarQube doesn't attempt.

The Architecture agent complemented the Retrieval agent by analyzing the repository's structural and dependency context. It correctly characterized the system as having "no security architecture" where controls were absent rather than misconfigured. It also identified auth.py as "an anti-pattern, not a control".

The Architecture agent identified dependency-level vulnerabilities that the Retrieval agent had not reported, each tied to a specific declared version in the project's manifest. Six declared packages matched known CVEs on NVD, enumerated in Table 8.

Table 8. Known CVEs identified by the Architecture agent.

Dependency	Version	Risk	Representative CVE
pyyaml	5.3.1	HIGH	CVE-2020-14343
cryptography	3.3.2	HIGH	CVE-2023-23931
pillow	8.2.0	HIGH	CVE-2022-22817; CVE-2023-50447; CVE-2021-34552
request	2.25.0	MEDIUM	CVE-2023-32681
jinja2	3.1.2	MEDIUM	CVE-2024-22195; CVE-2024-34064
flask	2.3.3	LOW-MED	-

These vulnerabilities were out of scope for the Retrieval agent by design because SonarQube Community Edition does not inspect dependency manifests.

Round 2 demonstrated that the new critical rule removed CVE misattribution. Agents performed cross-finding correlation and rescored SonarQube's severity ratings with written justification. The Architecture agent provided a dependency-risk table and systematic issues narrative, which the Report agent added into the Systemic Risks section of the final report, while the Retrieval agent, consistent with the round 2 rule, correctly didn't attach these CVEs to individual SonarQube findings.

However, a coverage limitation became visible. SonarQube reported 12 findings, all configuration-level (hardcoded secrets, insecure flags, missing headers, weak crypto settings), but it didn't detect any of the 9 injection-class vulnerabilities, including the 4 most critical in the entire app (SQLi, XSS, command injection, pickle RCE). The Code Analysis agent did not attempt to discover these either because it was configured to do so in the Round 2 agent configuration. This gap motivated the change in round 3 to enable the Code Analysis agent to actively identify vulnerabilities alongside triage.

5.5.3 Round 3 – Agent-driven discovery

Round 3 enabled the Code Analysis agent to discover new vulnerabilities as it read the source code, rather than just analyzing the SonarQube results. The discovery mechanism worked: In total, there were 27 findings, of which 15 were agent-discovered, and 9 of those agent-discovered findings were interception vulnerabilities that were not identified in Round 2, as summarized in Table 9.

Table 9. The Code Analysis agent identified all inserted injection vulnerabilities.

Finding	CWE	Severity
SQL Injection /login	CWE-89	CRITICAL
SQL Injection /user	CWE-89	CRITICAL
SSTI/XSS /search	CWE-94	CRITICAL
Command Injection /ping	CWE-78	CRITICAL
Pickle deserialization	CWE-502	CRITICAL
XXE	CWE-661	HIGH
Open Redirect	CWE-601	MEDIUM
Log Injection	CWE-117	MEDIUM

SonarQube found only configuration/secrets issues and completely missed every injection vulnerability, which the LLM agent compensated for by identifying them through data-flow analysis that SonarQube lacks. All inserted vulnerabilities were correctly identified and reported.

5.5.4 Round 4 – Scaling failure

For round 4, the system targeted a more naturalistic target: an open-source repository called pyLoad that is a Flask-based download manager compromising 577 Python source files, around 120 plugins, and an IRC/XDCC transport layer. The project was selected because it is written in Python, matching both the SonarQube profile and the Code Analysis agent's skills file. It exposes a web UI with authentication, file upload, and download features, covering a wide range of the OWASP top-10 surface (n.d.). It retains legacy Flash and Click'N'Load endpoints whose threat model is not obvious from structural analysis alone, which is precisely the cross-file reasoning the Architecture agent is performing.

However, the run failed to produce a report. SonarQube scanned the repository successfully and surfaced 486 raw findings (101 vulnerabilities and bugs, plus 385 security hotspots), but the Retrieval agent couldn't handle such a large load with 30 turn budget and lacked any filtering or deduplication making it attempt to enrich each finding individually. It consumed its entire turn allocation before giving a structured response, exited

with no output, and the pipeline therefore didn't get any data to work with. This was caused by two main reasons: there was no severity-based filtering at the SonarQube query stage, and no deduplication of enrichment calls by CWE.

5.5.5 Round 5 – Severity filter and CWE deduplication

The Retrieval agent's max-turns budget was raised from 30 to 120, and its prompt was extended with two new instructions: "*Group findings by rule key; enrich each unique CWE once, not each issue.*" and a severity filter that directs the agent to query SonarQube for BLOCKER, CRITICAL, and HIGH issues first. Enrichment calls are therefore bounded by the number of unique CWE groups rather than by the total findings.

The pipeline completed in 21 minutes and produced 70 findings, resulting 65 enriched from SonarQube and 5 discovered by the Code Analysis agent. The report surfaced several serious issues in pyLoad: an unauthenticated CNL remote-code-execution path via eval_js, TLS verification disabled on the XDCC transport, path traversal on DCC file writes, a live Google API key embedded across five plugin files, and a Docker base image that had reached end-of-life. The full executive summary of the resulting report is reproduced in Appendix 1. The five agent-discovered findings are the evidence that the multi-agent layer contributed results the SAST tool couldn't alone produce.

5.6 Final artifact

The final system produces a vulnerability report from a target repository in a single pipeline run. It uses two complementary classification schemes from MITRE: CVE and CWE. In practice, most findings from custom application code are labelled only with a CWE, because CVE identifiers only apply to known vulnerabilities in published components, such as third-party libraries. The Retrieval agent therefore attaches a CVE identifier only when the affected product matches an actual dependency in the target repository.

Otherwise, it reports the CWE category alone. This rule, introduced in round 2, prevents the incorrect CVE find mappings that were observed in Round 1. Table 10 lists the agent configuration of the final pipeline. Each agent runs on Claude Opus 4.7, but with a different tool-call budget matched to its responsibility.

Table 10. Pipeline agent configuration.

Agent	Model	Max Turns	MCP servers / tools
Retrieval	Claude Opus 4.7	120	SonarQube MCP, CVE MCP, WebSearch, WebFetch
Code Analysis	Claude Opus 4.7	40	Filesystem MCP, Glob, Grep
Architecture	Claude Opus 4.7	25	Filesystem MCP
Report	Claude Opus 4.7	5	None

The max turns shape each agent's behavior. The Retrieval agent is given 120 turns because it must loop over SonarQube pages and CVE lookups. The Code Analysis agent is given 40 because it reads and reasons file by file. The Architecture agent is capped at 25 to keep the directory walk bounded. And the Report agent is given 5 because it is a single synthesis step over an already structured state. Each agent should have a tailored turn budget, not a global default. Round 4 demonstrated the cost of ignoring this: the Retrieval agent quickly used its turns and produced an empty run.

The tool scoping follows the same logic. Each agent sees only the MCP servers it needs, which simplifies their decisions and makes behavior easier to audit. The Report agent is given no tools at all, because any tool call at that stage would risk re-opening decisions that the previous agents had already resolved.

Figure 11 shows how the agents connect to the orchestration layer, the MPC tool layer, and the external services (SonarQube REST API, NVD, and the target repository filesystem). SonarQube does the initial scan for targeted repository, and its findings are pulled through the SonarQube MCP server by the Retrieval agent, which also reaches out to the CVE MCP server for enrichment. The output of Retrieval agent is written into a shared AnalysisState dictionary that every agent reads from and extends. However, the agents

themselves are stateless, and the orchestrator is the only component that carries memory between agents.

The Code Analysis agent and Architecture agent run is sequence against that state, both using the Filesystem MCP server to read the target repository directly rather than relying on SonarQube's view. Finally, the Report agent closes the pipeline with no tools, consuming the state and building the final Markdown report based on it.

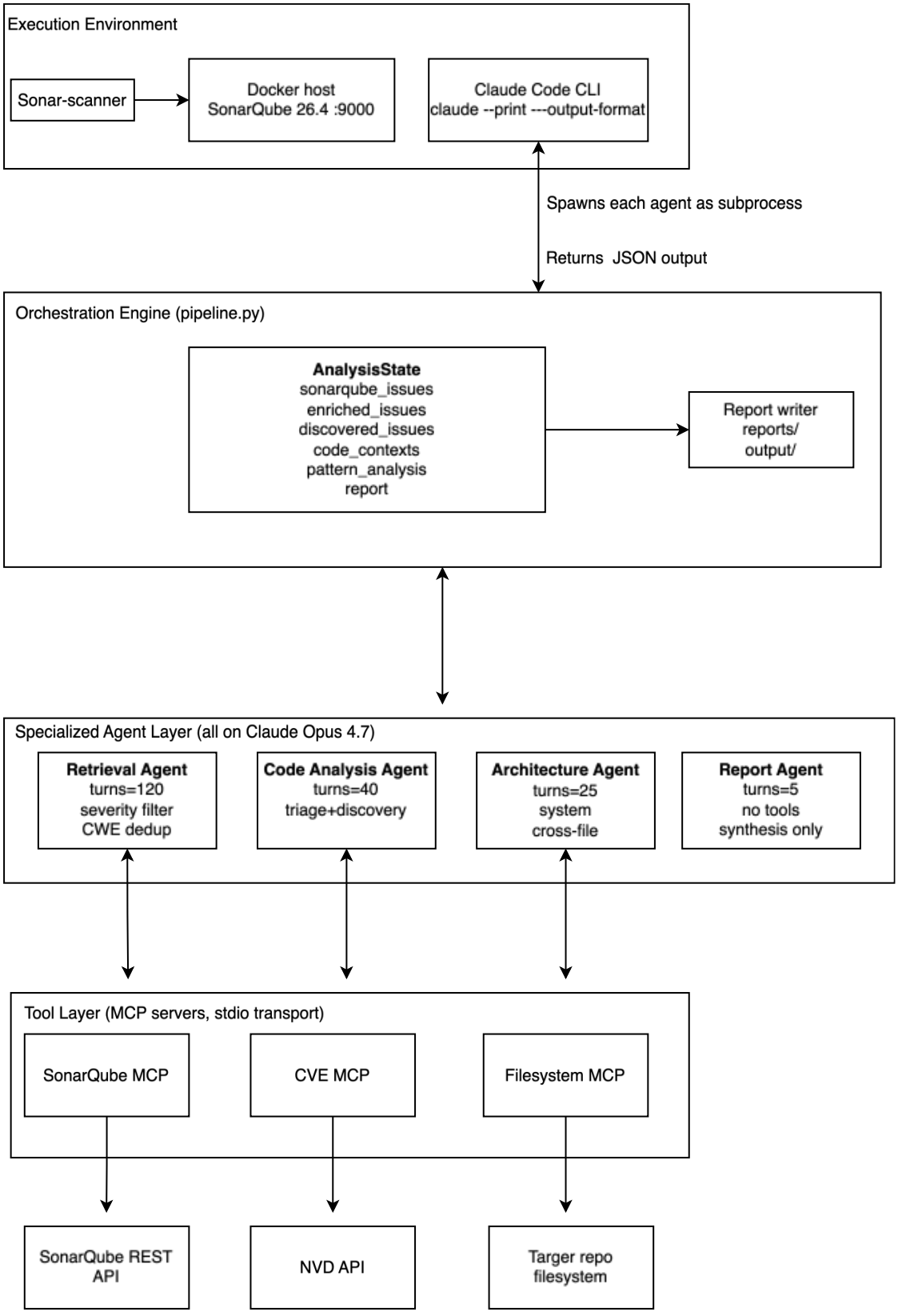

Figure 11. The final architecture with components.

The diagram also makes the separation between MCP servers and agents explicit. MCP servers are passive components that can be reused across agents and swapped without changing agent logic. Agents are active roles defined by a prompt, allowed toolset, and a turn budget.

Every design choice in the final artifact can be traced to a specific iteration round in which a limitation was observed, as summarized in Table 11.

Table 11. Design decisions in the final artifact and their iteration origin.

Design feature in final artifact	Introduced	Motivating observation
CWE-only attribution for application-code findings (no forced CVE)	Round 2	Round 1 Retrieval agent matched CVEs by CWE category alone, producing wrong product mappings
Two-phase Code Analysis (triage + proactive discovery)	Round 3	SonarQube detects no injection-class vulnerabilities
Severity Filter at SonarQube query stage (BLOCKER/CRITICAL/HIGH)	Round 5	Round 4 enrichment of 486 raw pyLoad findings exhausted the turn budget
CWE-level enrichment deduplication	Round 5	In Round 4 the agent looked up the same CVE many times, which wasted its whole turn budget
Increased the Retrieval agent turn budget raised to 120	Round 5	Round 4 ran out of turns at 30 and exited with no output

As Table 11 shows, no single round produced the full artifact: each design feature was introduced as a direct response to a specific observation, and no feature was added speculatively.

6 Discussion

This thesis set out to design an AI-based multi-agent system architecture for automated source-code vulnerability analysis enriched with public vulnerability data, and to demonstrate its technical feasibility through a PoC implementation. The artifact specifies the system's architectural components, the roles and responsibilities of each agent, and interactions that bind them into a single pipeline. Together, these elements answer the research questions posed in this study.

RQ1: Which architectural components are required to design a multi-agent system that supports automated vulnerability analysis? The architecture identifies four core components. An orchestration engine that manages the execution sequence of agents and maintains a shared state object between them. A specialized agent layer in which each agent has a distinct role, prompt, and turn budget. A tool layer that works as a standardized interface for all interaction with external data through MCP servers. And an execution environment in which the pipeline runs alongside its supporting services.

RQ2: What agent roles are needed in a multi-agent system to perform automated source-code analysis enriched with vulnerability data? Four agent roles cover the analysis pipeline end-to-end. A Retrieval agent that gathers findings from the static analyzer and enriches them with CVE and CWE information. A Code Analysis agent that reads the affected source files to validate findings and proactively search for issues that the scanner missed. An Architecture agent that reviews the wider repository structure for system-level and cross-file issues. And a Report agent that synthesizes the shared state into a structured report without making further tool calls.

The objective was to design and demonstrate the architecture, not to evaluate detection accuracy against a benchmark. Both research questions have been answered through the resulting artifact and its PoC. The final run against the pyLoad repository produced a structured report covering 70 findings, including a remote-code-execution vulnerability

that SonarQube alone did not flag as exploitable. The objectives have therefore been met within the scope.

The pipeline confirms the recent findings in LLM-based vulnerability detection studies. Combining a static analyzer with language-model reasoning produces stronger results than either approach alone, because the scanner anchors the LLM in concrete findings and reduces thereby space for model hallucination (Y. Huang et al., 2025, p. 1). Distributing tasks across multiple specialized agents rather than a single LLM reduces the per-agent context load and makes reasoning more stable, which follows the rationale described by Quo et al. (2024, p. 3).

From research perspective, the iteration history offers an empirical illustration of DSR in practice. Each round produced a single concrete change that was traceable to a specific observation. From a practical perspective, the architecture has the potential to reduce the development effort and cost associated with building a customized automated threat-detection pipeline, particularly compared with commercial solutions that bundle scanning, enrichment, and reporting into closed systems. Because each agent is defined by a prompt, a toolset, and a turn budget, the architecture is extensible and customizable, and new data sources can be added as MCP servers without redesigning the pipeline.

Based on the experience of building and running the pipeline, two recommendations follow for practitioners building similar system. First, apply prompt-level constraints on cross-catalogue mappings (such as CVE-CWE attribution) early, because plausible but wrong outputs are more likely to slip past review than obviously wrong ones. Second, design the static-analyzer query to filter and group findings before enrichment, rather than enriching every raw finding individually. In this study, severity filtering and CWE-level deduplication together turned a failed run into a successful one without altering the agent logic.

6.1 Limitations

Several limitations can be drawn from this study. First, the PoC was exercised against a single purpose-built test application and one real-world target (pyLoad). Understanding the viability against other languages, larger codebases, or different architectural styles requires further evaluation. Second, the used scanner is SonarQube Community Edition, which does not include taint analysis. Injection vulnerabilities that depend on tracing data flow across functions are therefore not detected at the scanning layer, and the multi-agent system can only act on findings that the scanner detects. Third, the system inherits the known limitations of large language model reasoning. Outputs may vary depending on the model's reasoning instability, which can cause possible hallucinations. Using LLM APIs involves practical limitations such as high computational costs and rate limits (Wu et al., 2025, p. 17). Finally, the system's behaviour is tied to the specific model version used during the study, which constrains reproducibility and a analysis run with a future model version may not produce identical results.

6.2 Future research

Future research can build the artifact in four directions. First, the PoC demonstrates feasibility but does not measure accuracy. A follow-up study could compare the pipeline's output against human-expert assessments, ideally across multiple programming languages and codebase sizes. Second, the system's outputs are tied to the specific model used during the run, a study tracking the same pipeline across different model versions would help characterize the stability of LLM-based vulnerability analysis over time. Third, the current pipeline is built around SonarQube and Python repositories. Replacing SonarQube with a taint-aware scanner, or extending the pipeline to other languages, would test the architecture in general level. Finally, once the accuracy has been independently validated, the next step is to operate the pipeline as a quality gate on each pull request, and study runtime budgets, developer experience, and how the report should be presented within a CI environment.

References

Aggarwal, M. (2023). A Study of CVSS v4.0: A CVE Scoring System. *2023 6th International Conference on Contemporary Computing and Informatics (IC3I)*, *6*, 1180–1186. https://doi.org/10.1109/IC3I59117.2023.10397701

Alqaradaghi, M., & Kozsik, T. (2024). Comprehensive Evaluation of Static Analysis Tools for Their Performance in Finding Vulnerabilities in Java Code. *IEEE Access*, *PP*, 1–1. https://doi.org/10.1109/ACCESS.2024.3389955

Anthropic. (n.d.-a). *Equipping agents for the real world with Agent Skills \ Anthropic*. Claude. Retrieved March 29, 2026, from https://claude.com/blog/equipping-agents-for-the-real-world-with-agent-skills

Anthropic. (n.d.-b). *Introducing the Model Context Protocol*. Retrieved March 27, 2026, from https://www.anthropic.com/news/model-context-protocol

Anwar, A., Khormali, A., Choi, J., Alasmary, H., Choi, S., Salem, S., Nyang, D., & Mohaisen, D. (2020). Measuring the Cost of Software Vulnerabilities. *ICST Transactions on Security and Safety*, *7*(23), 164551. https://doi.org/10.4108/eai.13-7-2018.164551

Armando, J., Ramón, J., Higuera, J., Antonio, J., Riera, T., & Melero, J. (2026). Integration of Large Language Models (LLMs) and Static Analysis for Improving the Efficacy of Security Vulnerability Detection in Source Code. *Computers, Materials & Continua*, *86*(3). https://doi.org/10.32604/cmc.2025.074566

Aubakirova, M., Atallah, A., Clark, C., Summerville, J., & Midha, A. (2026). *State of AI: An Empirical 100 Trillion Token Study with OpenRouter* (arXiv:2601.10088; Version 1). arXiv. https://doi.org/10.48550/arXiv.2601.10088

Balsam, A., Nowak, M., Walkowski, M., Oko, J., & Sujecki, S. (2024). Comprehensive comparison between versions CVSS v2.0, CVSS v3.x and CVSS v4.0 as vulnerability severity measures. *2024 24th International Conference on Transparent Optical Networks (ICTON)*, 1–4. https://doi.org/10.1109/ICTON62926.2024.10647452

Benmalek, M. (2024). Ransomware on cyber-physical systems: Taxonomies, case studies, security gaps, and open challenges. *Internet of Things and Cyber-Physical Systems*, *4*, 186–202. https://doi.org/10.1016/j.iotcps.2023.12.001

Chen, Y., Huang, Y., Chen, X., Shen, P., & Yun, L. (2025). GPTVD: Vulnerability detection and analysis method based on LLM's chain of thoughts. *Automated Software Engineering*, *33*(1), 3. https://doi.org/10.1007/s10515-025-00550-4

Cong, N., Huy, L., & Thanh, T. (2024). An overview of static and dynamic analysis in application security testing. *Journal of Military Science and Technology*, *99*, 1–11. https://doi.org/10.54939/1859-1043.j.mst.99.2024.1-11

Coutinho, L. S., Menasche, D., Miranda, L., Lovat, E., Kumar, S. G., Ramchandran, A., Kocheturov, A., & Limmer, T. (2024). How Context Impacts Vulnerability Severity: An Analysis of Product-Specific CVSS Scores. *Proceedings of the 13th Latin-American Symposium on Dependable and Secure Computing*, 17–27. https://doi.org/10.1145/3697090.3697109

CVE Program. (n.d.-a). *CVE: Common Vulnerabilities and Exposures*. Retrieved February 22, 2026, from https://www.cve.org/about/Metrics

CVE Program. (n.d.-b). *CVE: Common Vulnerabilities and Exposures*. Retrieved February 22, 2026, from https://www.cve.org/

Daneshvar, S. S., Nong, Y., Yang, X., Wang, S., & Cai, H. (2024, August 7). *VulScribeR: Exploring RAG-based Vulnerability Augmentation with LLMs*. arXiv.Org. https://doi.org/10.1145/3760775

De Silva, D., Samarasekara, P., & Hettiarachchi, R. (2023). *A Comparative Analysis of Static and Dynamic Code Analysis Techniques*. https://doi.org/10.36227/techrxiv.22810664.v1

Dietrich, J., Rasheed, S., Jordan, A., & White, T. (2024). On the Security Blind Spots of Software Composition Analysis. *Proceedings of the 2024 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses, SCORED '24*, 77–87. https://doi.org/10.1145/3689944.3696165

Du, X., Zheng, G., Wang, K., Zou, Y., Wang, Y., Deng, W., Feng, J., Liu, M., Chen, B., Peng, X., Ma, T., & Lou, Y. (2024, June 17). *Vul-RAG: Enhancing LLM-based Vulnerability Detection via Knowledge-level RAG*. arXiv.Org. https://arxiv.org/abs/2406.11147v3

Egele, M., Scholte, T., Kirda, E., & Kruegel, C. (2012). A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys*, *44*(2), 1–42. https://doi.org/10.1145/2089125.2089126

ENISA. (n.d.). *ENISA Threat Landscape 2025 | ENISA*. Retrieved March 29, 2026, from https://www.enisa.europa.eu/publications/enisa-threat-landscape-2025

Fan, W., Ding, Y., Ning, L., Wang, S., Li, H., Yin, D., Chua, T.-S., & Li, Q. (2024). A Survey on RAG Meeting LLMs: Towards Retrieval-Augmented Large Language Models. *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, KDD '24*, 6491–6501. https://doi.org/10.1145/3637528.3671470

Graham, J., Olson, R., & Howard, R. (2010). *Cyber Security Essentials*. Auerbach Publishers, Incorporated.

Gregor, S., & Hevner, A. (2013). Positioning and Presenting Design Science Research for Maximum Impact. *MIS Quarterly*, *37*, 337–356. https://doi.org/10.25300/MISQ/2013/37.2.01

Guo, J., Li, N., Qi, J., Yang, H., Li, R., Feng, Y., Zhang, S., & Xu, M. (2024). *Empowering Working Memory for Large Language Model Agents* (arXiv:2312.17259). arXiv. https://doi.org/10.48550/arXiv.2312.17259

Guo, T., Chen, X., Wang, Y., Chang, R., Pei, S., Chawla, N. V., Wiest, O., & Zhang, X. (2024). *Large Language Model based Multi-Agents: A Survey of Progress and Challenges* (arXiv:2402.01680). arXiv. https://doi.org/10.48550/arXiv.2402.01680

Hevner, A., March, S., Park, J., & Ram, S. (2004). Design Science in Information Systems Research. *Management Information Systems Quarterly*, *28*, 75. https://doi.org/10.2307/25148625

Hou, X., Zhao, Y., Wang, S., & Wang, H. (2025). *Model Context Protocol (MCP): Landscape, Security Threats, and Future Research Directions* (arXiv:2503.23278). arXiv. https://doi.org/10.48550/arXiv.2503.23278

Huang, L., Yu, W., Ma, W., Zhong, W., Feng, Z., Wang, H., Chen, Q., Peng, W., Feng, X., Qin, B., & Liu, T. (2023, November 9). *A Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions*. arXiv.Org. https://doi.org/10.1145/3703155

Huang, Y., Chen, Y., Cai, J., Wu, J., Chen, X., Shen, P., & Yun, L. (2025). Towards combining chain-of-thought and code static analysis for buffer overflow vulnerability

detection. *Software Quality Journal*, *34*(1), 2. https://doi.org/10.1007/s11219-025-09733-4

Humran, H. A. A., & Sonmez, F. (2025). *Code Vulnerability Detection Across Different Programming Languages with AI Models* (arXiv:2508.11710). arXiv. https://doi.org/10.48550/arXiv.2508.11710

ISTQB. (2024). *ISTQB Certified Tester—Foundation Level Syllabus v4.0.1*. https://istqb.org/wp-content/uploads/2024/11/ISTQB_CTFL_Syllabus_v4.0.1.pdf

Ivanov, S., Ilina, L., & Ilin, D. (2021). Threat Landscape as an Information Security Analysis Means. In *IV International Scientific and Practical Conference* (pp. 1–5). https://doi.org/10.1145/3487757.3490918

Jia, Z., Li, J., Kang, Y., Wang, Y., Wu, T., Wang, Q., Wang, X., Zhang, S., Shen, J., Li, Q., Qi, S., Liang, Y., He, D., Zheng, Z., & Zhu, S.-C. (2026). *The AI Hippocampus: How Far are We From Human Memory?* (arXiv:2601.09113). arXiv. https://doi.org/10.48550/arXiv.2601.09113

Kereopa-Yorke, B. (2023). Building resilient SMEs: Harnessing large language models for cyber security in Australia. *Journal of AI, Robotics & Workplace Automation*, *3*(1), 15. https://doi.org/10.69554/XSQZ3232

Kim, Y., Gu, K., Park, C., Park, C., Schmidgall, S., Heydari, A. A., Yan, Y., Zhang, Z., Zhuang, Y., Malhotra, M., Liang, P. P., Park, H. W., Yang, Y., Xu, X., Du, Y., Patel, S., Althoff, T., McDuff, D., & Liu, X. (2025, December 9). *Towards a Science of Scaling Agent Systems*. arXiv.Org. https://arxiv.org/abs/2512.08296v2

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W., Rocktäschel, T., Riedel, S., & Kiela, D. (2021). *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks* (arXiv:2005.11401). arXiv. https://doi.org/10.48550/arXiv.2005.11401

Li, X., Wang, S., Zeng, S., Wu, Y., & Yang, Y. (2024). A survey on LLM-based multi-agent systems: Workflow, infrastructure, and challenges. *Vicinagearth*, *1*. https://doi.org/10.1007/s44336-024-00009-2

Li, Y., Joshi, K., Wang, X., & Wong, E. (2025). *MAVUL: Multi-Agent Vulnerability Detection via Contextual Reasoning and Interactive Refinement* (arXiv:2510.00317). arXiv. https://doi.org/10.48550/arXiv.2510.00317

Lin, E., Gowda, S., Enck, W., & Wermke, D. (2025). *Context Matters: Qualitative Insights into Developers' Approaches and Challenges with Software Composition Analysis*. https://www.usenix.org/system/files/usenixsecurity25-lin-elizabeth.pdf

Liu, B., Shi, L., Cai, Z., & Li, M. (2012). Software Vulnerability Discovery Techniques: A Survey. *2012 Fourth International Conference on Multimedia Information Networking and Security*, 152–156. https://doi.org/10.1109/MINES.2012.202

Louridas, P. (2006). Static code analysis. *IEEE Software*, *23*(4), 58–61. https://doi.org/10.1109/MS.2006.114

Malkawi, M., & Alhajj, R. (2026). AI-Powered Vulnerability Detection and Patch Management in Cybersecurity: A Systematic Review of Techniques, Challenges, and Emerging Trends. *Machine Learning and Knowledge Extraction*, *8*(1). https://doi.org/10.3390/make8010019

Mell, P., Scarfone, K., & Romanosky, S. (2006). Common Vulnerability Scoring System. *IEEE Security & Privacy*, *4*(6), 85–89. https://doi.org/10.1109/MSP.2006.145

Michael, R., & Bellis, E. (2023). *Modern Vulnerability Management: Predictive Cybersecurity*. Artech House. http://ebookcentral.proquest.com/lib/tritonia-ebooks/detail.action?docID=30516624

Minaee, S., Mikolov, T., Nikzad, N., Chenaghlu, M., Socher, R., Amatriain, X., & Gao, J. (2025). *Large Language Models: A Survey* (arXiv:2402.06196). arXiv. https://doi.org/10.48550/arXiv.2402.06196

MITRE. (n.d.-a). *CWE - Common Weakness Enumeration*. Retrieved April 24, 2026, from https://cwe.mitre.org/

MITRE. (n.d.-b). *CWE - CVE → CWE Mapping "Root Cause Mapping" Guidance*. CWE™ - Common Weakness Enumeration. Retrieved April 24, 2026, from https://cwe.mitre.org/documents/cwe_usage/guidance.html

Mweu, B., & Ndia, J. (2025). *Static Analysis Techniques for Secure Software: A Systematic Review*. https://doi.org/10.32604/jcs.2025.071765

Ni, C., Yin, X., Shen, L., & Wang, S. (2025). Learning-based models for vulnerability detection: An extensive study. *Empirical Software Engineering*, *31*(1), 18. https://doi.org/10.1007/s10664-025-10734-x

Noever, D. (2023). *Can Large Language Models Find And Fix Vulnerable Software?* (arXiv:2308.10345). arXiv. https://doi.org/10.48550/arXiv.2308.10345

NVD. (n.d.). *NVD*. Vulnerability Metrics. Retrieved March 6, 2026, from https://nvd.nist.gov/vuln-metrics/cvss

OWASP. (n.d.). *OWASP Attack Surface Management Top 10 | OWASP Foundation*. Retrieved April 19, 2026, from https://owasp.org/www-project-attack-surface-management-top-10/

Ozkaya, E. (2019). *Cybersecurity: The Beginner's Guide: A Comprehensive Guide to Getting Started in Cybersecurity*. Packt Publishing, Limited.

Parizi, R., Qian, K., Shahriar, H., Wu, F., & Tao, L. (2018). Benchmark Requirements for Assessing Software Security Vulnerability Testing Tools. *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, *01*, 825–826. https://doi.org/10.1109/COMPSAC.2018.00139

Peffers, K., Tuunanen, T., Rothenberger, M. A., & Chatterjee, S. (2007). A Design Science Research Methodology for Information Systems Research. *Journal of Management Information Systems*, *24*(3), 45–77. https://doi.org/10.2753/MIS0742-1222240302

Peng, B., Galley, M., He, P., Cheng, H., Xie, Y., Hu, Y., Huang, Q., Liden, L., Yu, Z., Chen, W., & Gao, J. (2023, February 24). *Check Your Facts and Try Again: Improving Large Language Models with External Knowledge and Automated Feedback*. arXiv.Org. https://arxiv.org/abs/2302.12813v3

Qiu, F., Liu, Z., Hu, B., Cai, Z., Bao, L., & Wang, X. (2026). RLV: LLM-based vulnerability detection by retrieving and refining contextual information. *Journal of Systems and Software*, *235*, 112756. https://doi.org/10.1016/j.jss.2025.112756

Sánchez, M. C., de Gea, J. M. C., Fernández-Alemán, J. L., Garceran, J., & Toval, A. (2020). Software vulnerabilities overview: A descriptive study. *Tsinghua Science and Technology*, *25*(2), 270–280. https://doi.org/10.26599/TST.2019.9010003

Schott, S., Ponta, S. E., Fischer, W., Klauke, J., & Bodden, E. (2026). *Uncovering Hidden Inclusions of Vulnerable Dependencies in Real-World Java Projects* (arXiv:2601.23020). arXiv. https://doi.org/10.48550/arXiv.2601.23020

Shan, L., Luo, S., Zhu, Z., Yuan, Y., & Wu, Y. (2025). *Cognitive Memory in Large Language Models* (arXiv:2504.02441). arXiv. https://doi.org/10.48550/arXiv.2504.02441

Shen, Z. (2024, September 24). *LLM With Tools: A Survey*. arXiv.Org. https://arxiv.org/abs/2409.18807v1

Stanton, B., Theofanos, M. F., Prettyman, S. S., & Furman, S. (2016). Security Fatigue. *IT Professional*, *18*(5), 26–32. https://doi.org/10.1109/MITP.2016.84

Steenhoek, B., Rahman, M. M., Roy, M. K., Alam, M. S., Barr, E. T., & Le, W. (2024). *A Comprehensive Study of the Capabilities of Large Language Models for Vulnerability Detection* (arXiv:2403.17218; Version 1). arXiv. https://doi.org/10.48550/arXiv.2403.17218

Strachan, J. W. A., Albergo, D., Borghini, G., Pansardi, O., Scaliti, E., Gupta, S., Saxena, K., Rufo, A., Panzeri, S., Manzi, G., Graziano, M. S. A., & Becchio, C. (2024). Testing theory of mind in large language models and humans. *Nature Human Behaviour*, *8*(7), 1285–1295. https://doi.org/10.1038/s41562-024-01882-z

Sutton, D. (2022). *Cyber Security: The Complete Guide to Cyber Threats and Protection*. BCS Learning & Development Limited.

Taghavi Far, S. M., & Feyzi, F. (2025). Large language models for software vulnerability detection: A guide for researchers on models, methods, techniques, datasets, and metrics. *International Journal of Information Security*, *24*(2), 78. https://doi.org/10.1007/s10207-025-00992-7

Tan, J. J. Y., Otto, K. N., & Wood, K. L. (2017). Relative impact of early versus late design decisions in systems development. *Design Science*, *3*, e12. https://doi.org/10.1017/dsj.2017.13

Traficom. (n.d.). *Cyber Weather*. NCSC-FI. Retrieved March 29, 2026, from https://www.kyberturvallisuuskeskus.fi/en/ncsc-news/cyber-weather

Tran, K.-T., Dao, D., Nguyen, M.-D., Pham, Q.-V., O'Sullivan, B., & Nguyen, H. D. (2025). *Multi-Agent Collaboration Mechanisms: A Survey of LLMs* (arXiv:2501.06322). arXiv. https://doi.org/10.48550/arXiv.2501.06322

Tymchuk, Y. (2017, June). *The False False Positives of Static Analysis*. Seminar Series on Advanced Techniques and Tools for Software Evolution SATToSE 2017. https://doi.org/10.7892/boris.113148

Vijayaraghavan, G., Jayachandran, P., Murthy, A., Govindan, S., & Subramanian, V. (2026). *If You Want Coherence, Orchestrate a Team of Rivals: Multi-Agent Models of Organizational Intelligence* (arXiv:2601.14351). arXiv. https://doi.org/10.48550/arXiv.2601.14351

Wooldridge, M. (2009). *An Introduction to MultiAgent Systems* (2nd ed.). Wiley Publishing.

Wooldridge, M., Jennings, N. R., & Kinny, D. (2000). The Gaia Methodology for Agent-Oriented Analysis and Design. *Autonomous Agents and Multi-Agent Systems*, *3*(3), 285–312. https://doi.org/10.1023/A:1010071910869

Wu, X., Tian, Y., Chen, Y., Ye, P., Cui, X., Jia, J., Li, S., Liu, J., Niu, W., Wu, X., Tian, Y., Chen, Y., Ye, P., Cui, X., Jia, J., Li, S., Liu, J., & Niu, W. (2025). CurriculumPT: LLM-Based

Multi-Agent Autonomous Penetration Testing with Curriculum-Guided Task Scheduling. *Applied Sciences*, *15*(16). https://doi.org/10.3390/app15169096

Xu, R., & Yan, Y. (2026, February 12). *Agent Skills for Large Language Models: Architecture, Acquisition, Security, and the Path Forward*. arXiv.Org. https://arxiv.org/abs/2602.12430v3

Yang, Z., Peng, H., Jiang, Y., Liu, J., Luo, H., Tang, M., Li, J., & Zhang, K. (2026). LIVA: A Multi-Agent LLM-Assisted System for IoT Vulnerability Analysis. *IEEE Transactions on Dependable and Secure Computing*, 1–17. https://doi.org/10.1109/TDSC.2026.3665343

ZeMicheal, T., Chen, H., Davis, S., Allen, R., Demoret, M., & Song, A. (2024). *LLM agents for vulnerability identification and verification of CVEs*. https://api.semanticscholar.org/CorpusID:276648381

Zhang, F., Fan, L., Chen, S., Cai, M., Xu, S., & Zhao, L. (2024). Does the Vulnerability Threaten Our Projects? Automated Vulnerable API Detection for Third-Party Libraries. *IEEE Transactions on Software Engineering*, *50*(11), 2906–2920. https://doi.org/10.1109/TSE.2024.3454960

Appendices

Appendix 1. The executive summary of the analysis report (round 5)

Project: pyLoad (pyload-ng)

Date: 2026-04-17

Scan Tool: SonarQube + CVE/NVD enrichment + Agent-assisted code review

Executive Summary

A combined SonarQube + agent-assisted review of the pyLoad download manager surfaced 70 security findings across the Flask web UI, the XDCC/IRC network layer, and the plugin ecosystem.

- SonarQube findings: 65 (46 CRITICAL, 19 HIGH)
- Agent-discovered findings (missed by SonarQube): 5 (1 HIGH, 4 MEDIUM)

Groupings materially reduce the unique surface: 10 of the CRITICALs are the same Google API key copy-pasted across five plugins; 10 more are Mega/DLC/RSDF/CriptTo/Filecrypt cryptographic patterns dictated by external protocols.

Top risk: Server-side evaluation of attacker-supplied JavaScript at src/pyload/webui/app/blueprints/cnl_blueprint.py:148 (`eval_js(f"{jk} f()")`). The endpoint is CSRF-exempt and gated only by a `local_check` decorator that inspects forge-able `REMOTE_ADDR`/`HTTP_HOST` headers — a DNS-rebinding or reverse-proxy misconfiguration promotes this to remote code execution. Immediately compounded by system-wide TLS verification disabled in `xdcc/request.py` (`CERT_NONE` + `check_hostname=False`) and a live Google API key leaked across five plugin files.

Systemic pattern: Security hygiene at the top-level Flask config is reasonable (headers, session flags, autoescape, CSRF framework) — but all of the most sensitive surfaces (CNL blueprint, XDCC, plugin decrypters) sit outside those controls by design.

Appendix 2. Retrieval agent prompt

```
# Retrieval Agent Skills

## Core Responsibility

The Retrieval Agent is responsible for gathering initial vulnerability data from SonarQube and enriching it with detailed CVE information. It acts as the primary data gatherer for the analysis pipeline.

## Key Skills

1. **SonarQube Integration**:
    - Connects to a SonarQube instance via the official SonarQube MCP server.
    - Executes queries against the `/api/issues/search` and `/api/hotspots/search` endpoints.
    - Parses and normalizes SonarQube's JSON output into a structured format.

2. **CVE/CWE Enrichment (deduplicated)**:
    - Identifies CVE and CWE identifiers from SonarQube issue descriptions.
    - Groups findings by rule/CWE so each unique weakness is enriched exactly once, then projects the result onto all issues in the group.
    - Connects to the CVE MCP server to fetch CVSS scores, attack vectors, exploit information, and remediation guidance.
    - Prioritises BLOCKER/CRITICAL/HIGH severity and caps result counts so the pipeline scales to real-world repos with hundreds of findings.

3. **Data Normalization**:
    - Merges the data from SonarQube and the CVE server into a single, consistent schema (`enriched_issues`).
    - Ensures each vulnerability record contains all necessary information for downstream agents, including file paths, line numbers, severity, and detailed CVE data.

## Tools

- **SonarQube MCP Server**: For querying project vulnerabilities.
```

- **CVE MCP Server**: For enriching findings with CVE intelligence.

Expected Output

The agent will add a list of enriched issues to the state. Each item in the list will be a dictionary with the following structure:

```json
[
  {
    "issue_id": "sonarqube-AX-12345",
    "file_path": "src/main/java/com/example/UserController.java",
    "line_range": [45, 45],
    "severity": "HIGH",
    "sonarqube_message": "Make sure that using this hardcoded IP address is safe here.",
    "cve_id": "CVE-2021-44228",
    "cwe_id": "CWE-502",
    "cvss_score": 10.0,
    "attack_vector": "NETWORK",
    "remediation_guidance": "Upgrade log4j-core to version 2.17.1 or later...",
    "known_exploits": ["https://github.com/advisories/GHSA-jfh8-c2jp-5v3q"]
  }
]
```

You are a security data analyst responsible for gathering and enriching vulnerability data.

PHASE 1 — Fetch SonarQube findings (severity-prioritised)
  Real-world repositories can have hundreds of findings, so focus on what matters most:
  1. Call search_issues with severities=BLOCKER,CRITICAL,HIGH and types=VULNERABILITY,BUG. Cap at 50 results (page_size=50).

2. Call search_hotspots with the given project key. Cap at 30 results.
3. If PHASE 1 returns zero findings at HIGH+, fall back to MEDIUM once.

PHASE 2 — CWE/CVE enrichment (deduplicated)
Group the fetched findings by rule key (or CWE ID if present). For each UNIQUE rule/CWE group — NOT each individual issue — do one enrichment pass:
1. Extract any CVE (CVE-YYYY-NNNNN) or CWE (CWE-NNN) identifiers from the rule, message, or tags.
2. If a CVE ID exists → call get_cve, get_cvss, and list_exploits (once).
3. If only a CWE ID exists → call search_cwe once to find related CVEs, then get_cvss for the most relevant hit.
4. If the CVE/NVD tools return errors or no data, use WebSearch as a fallback. Do this at most twice total across the whole run.
5. If neither CVE nor CWE exists, include the issue with null fields.

Then project the enrichment back onto each issue in the group. This keeps the total tool calls bounded regardless of how many issues SonarQube reports.

CRITICAL RULE — CVE attribution accuracy:
Only assign a CVE ID to a finding if the CVE's affected product or library matches an actual dependency or component in the target repository. A matching CWE category alone is NOT sufficient. For example, do NOT assign a CVE for "Arkeia Network Backup" to a Flask application just because both involve CWE-798. If no product-specific CVE exists, report the CWE ID only and set cve_id to null. Accuracy matters more than completeness.

After processing every issue, output ONLY a JSON array (inside a ```json code fence) with the enriched records. Each record must have these keys:
issue_id, file_path, line_range, severity, sonarqube_message,
cve_id, cwe_id, cvss_score, attack_vector,
remediation_guidance, known_exploits
Use null for missing values and [] for empty lists.

Fetch all vulnerabilities from SonarQube for project key "pyload" and enrich each finding with CVE intelligence. Produce the enriched JSON array as described above.

Appendix 3. Code Analysis agent prompt

```
# Code Analysis Agent Skills

## Core Responsibility

The Code Analysis Agent performs a deep dive into the source code associated with each identified vulnerability AND actively discovers new vulnerabilities that SonarQube missed. It acts as both a validator of known findings and an independent security reviewer.

## Key Skills

1. **Code Inspection** (Phase 1 — Analyze Known Issues):
    - Connects to the Filesystem MCP server.
    - For each known vulnerability, reads the source code of the affected file.
    - Focuses on a context window of approximately 50 lines before and after the flagged line of code.

2. **Vulnerability Pattern Recognition**:
    - Analyzes code snippets to identify specific programming patterns that lead to vulnerabilities (e.g., unsanitized input, improper error handling, use of deprecated functions).
    - Traces data flow within snippets to determine if vulnerable code paths are reachable by user-controlled input.

3. **Risk Assessment**:
    - Assesses whether mitigating controls (e.g., `try/catch` blocks, input sanitization libraries, access control checks) are present near vulnerable code.
    - Provides a risk assessment for each issue based on code context.

4. **Vulnerability Discovery** (Phase 2 — Discover New Issues):
    - Scans ALL source files in the repository, not just files with known issues.
    - Actively hunts for vulnerability classes that SonarQube Community Edition commonly misses:
        - SQL injection (string concatenation in queries)
```

        - Cross-site scripting / template injection
        - OS command injection (subprocess with shell=True)
        - Path traversal (user input in file paths)
        - Open redirect (unvalidated redirect targets)
        - Deserialization of untrusted data (pickle.loads, yaml.load)
        - XML External Entity injection
        - Log injection
        - Server-Side Request Forgery
    - Reports each new finding with full context: code snippet, CWE mapping, data flow analysis, and remediation guidance.

Tools

- **Filesystem MCP Server**: For reading source code files and listing directories in the target repository.

Expected Output

The agent produces a JSON object with two keys:

```json
{
  "code_contexts": [
    {
      "issue_id": "sonarqube-AX-12345",
      "file_path": "src/main/java/com/example/UserController.java",
      "code_snippet": "44:     String ip = \"192.168.1.100\"; ...",
      "analysis": "The code contains a hardcoded IP address...",
      "data_flow": "The variable 'ip' is defined as a static string...",
      "risk_assessment": "LOW. While a hardcoded IP is a maintenance concern..."
    }
  ],
  "discovered_issues": [
    {
      "issue_id": "discovered-1",
```

```
      "file_path": "app.py",
      "line_range": [41, 41],
      "severity": "CRITICAL",
      "sonarqube_message": null,
      "cve_id": null,
      "cwe_id": "CWE-89",
      "cvss_score": null,
      "attack_vector": "NETWORK",
      "remediation_guidance": "Use parameterized queries...",
      "known_exploits": [],
      "code_snippet": "41:     query = \"SELECT * FROM users WHERE username =
'\" + username + \"'\"",
      "analysis": "User input from request.form is concatenated directly into
SQL query...",
      "data_flow": "request.form → username variable → string concatenation →
db.execute()",
      "risk_assessment": "CRITICAL. Direct SQL injection with no sanitization.",
      "discovery_method": "agent-code-review"
    }
  ]
}
```

You are a senior application security engineer performing code-level vulnerability analysis.

You have TWO jobs:

═══ PHASE 1 — Analyze known issues ═══
For each vulnerability you are given, use the read_file tool to read the affected source file and examine approximately 50 lines of context around the flagged line.

For each issue, determine:

1. The exact vulnerable code pattern (e.g. unsanitized input, SQL concatenation, hard-coded secret).
2. The data flow — can user-controlled input reach the vulnerable code?
3. Whether mitigating controls exist nearby (input validation, try/except, access checks, WAF headers).
4. A risk assessment: CRITICAL / HIGH / MEDIUM / LOW with a brief justification.

=== PHASE 2 — Discover new vulnerabilities ===
While reading the source files, actively look for security vulnerabilities that are NOT in the known issues list. SonarQube has limited taint-analysis rules and commonly misses:
  - SQL injection (string concatenation in queries)
  - Cross-site scripting / template injection (user input in templates)
  - OS command injection (user input in subprocess/os.system/shell=True)
  - Path traversal (user input in file paths without normalization)
  - Open redirect (unvalidated redirect targets)
  - Deserialization of untrusted data (pickle.loads, yaml.load)
  - XML External Entity injection (xml.etree without defusedxml)
  - Log injection (unsanitized user input in log messages)
  - Server-Side Request Forgery (user-controlled URLs in requests)

Use list_directory to check for source files you haven't read yet — scan ALL Python/JS/Java source files in the repository, not just the ones with known issues.

=== OUTPUT FORMAT ===
Output a single JSON object (in a ```json code fence) with two keys:

```
{
  "code_contexts": [
    {
      "issue_id": "<original sonarqube issue_id>",
      "file_path": "...",
      "code_snippet": "...",
      "analysis": "...",
      "data_flow": "...",
      "risk_assessment": "CRITICAL / HIGH / MEDIUM / LOW — justification"
```

```
    }
  ],
  "discovered_issues": [
    {
      "issue_id": "discovered-<sequential number>",
      "file_path": "...",
      "line_range": [start, end],
      "severity": "CRITICAL / HIGH / MEDIUM / LOW",
      "sonarqube_message": null,
      "cve_id": null,
      "cwe_id": "CWE-NNN",
      "cvss_score": null,
      "attack_vector": "NETWORK / LOCAL / null",
      "remediation_guidance": "...",
      "known_exploits": [],
      "code_snippet": "...",
      "analysis": "...",
      "data_flow": "...",
      "risk_assessment": "CRITICAL / HIGH / MEDIUM / LOW — justification",
      "discovery_method": "agent-code-review"
    }
  ]
}
```

IMPORTANT: Do not duplicate — if an issue is already in the known list, do NOT add it to discovered_issues. Only report genuinely new findings.

Analyze the code for the following 65 known vulnerabilities, AND scan all source files for additional vulnerabilities that SonarQube missed.

Known issues: [a JSON array of SonarQube findings is inserted here at runtime by the orchestrator]

Start by listing the repository root directory to see all files, then read each source file. Produce the JSON object with both code_contexts and discovered_issues.

Appendix 4. Architecture agent prompt

```
# Architecture Agent Skills

## Core Responsibility

The Architecture Agent analyzes the high-level structure of the codebase to understand systemic risks and architectural patterns that may impact security.

## Key Skills

1. **Project Structure Analysis**:
    - Uses the Filesystem MCP server to list directories and read key project files.
    - Identifies the primary programming language and framework (e.g., Node.js with Express, Python with Django, Java with Spring).
    - Examines configuration files (`*.json`, `*.yml`, `*.xml`) for security-relevant settings.

2. **Dependency Analysis**:
    - Parses dependency management files (`package.json`, `requirements.txt`, `pom.xml`) to identify third-party libraries.
    - Cross-references library versions with known vulnerabilities (dependency version risks).

3. **Security Pattern Identification**:
    - Looks for common security patterns and components, such as:
        - Authentication and authorization middleware.
        - Input validation libraries or custom functions.
        - Centralized logging and error handling.
    - Determines if vulnerabilities are located in core shared libraries or isolated application code.

## Tools

- **Filesystem MCP Server**: For navigating the project structure and reading files.
```

Expected Output

The agent will output a single JSON object containing its analysis of the project's architecture.

```json
{
  "framework": "Java/Spring Boot",
  "security_patterns": [
    "Spring Security for authentication/authorization",
    "Use of `javax.validation` for input validation on DTOs",
    "Centralized exception handling via `@ControllerAdvice`"
  ],
  "dependency_risks": [
    {
      "dependency": "log4j-core",
      "version": "2.14.1",
      "risk": "Critical - Vulnerable to CVE-2021-44228 (Log4Shell). Immediate upgrade required."
    }
  ],
  "systemic_issues": "The project relies on an outdated version of a critical logging library. Several controllers lack explicit authorization checks, relying on a default-deny policy that may not be configured correctly.",
  "architecture_summary": "This is a typical Spring Boot monolithic application. Security is primarily handled by the Spring Security framework, but there are potential gaps in authorization and significant risk from an outdated dependency."
}
```

You are a software security architect. Your job is to examine the high-level structure of a codebase and assess systemic security risks.

Use the list_directory tool to explore the repository layout starting from '.', then drill into key directories. Use read_file to inspect:
  - Dependency manifests (requirements.txt, package.json, pom.xml, go.mod, …)
  - Configuration files (*.yml, *.yaml, *.json, *.toml, *.xml, Dockerfile, …)
  - Authentication / authorization modules
  - Any centralized error-handling or input-validation code

Produce a single JSON object (in a ```json code fence) with these keys:
  framework   — primary language and framework detected
  security_patterns — list of security controls / patterns found
  dependency_risks — list of {dependency, version, risk} for risky deps
  systemic_issues   — narrative string summarising cross-cutting weaknesses
  architecture_summary — brief overall assessment

Analyze the architecture and security posture of this repository.

Known vulnerability locations for context: src/pyload/core/network/xdcc/request.py (CRITICAL), src/pyload/core/network/xdcc/request.py (CRITICAL), src/pyload/core/network/xdcc/request.py (CRITICAL), src/pyload/webui/app/blueprints/cnl_blueprint.py (CRITICAL), src/pyload/core/utils/fs.py (CRITICAL), src/pyload/plugins/addons/ClickNLoad.py (CRITICAL), src/pyload/plugins/addons/ClickNLoad.py (CRITICAL), src/pyload/plugins/addons/ClickNLoad.py (CRITICAL), src/pyload/plugins/decrypters/GooGl.py (CRITICAL), src/pyload/plugins/decrypters/GooGl.py (CRITICAL), src/pyload/plugins/decrypters/GoogledriveComDereferer.py (CRITICAL), src/pyload/plugins/decrypters/GoogledriveComDereferer.py (CRITICAL), src/pyload/plugins/decrypters/GoogledriveComFolder.py (CRITICAL), src/pyload/plugins/decrypters/GoogledriveComFolder.py (CRITICAL), src/pyload/plugins/decrypters/YoutubeComFolder.py (CRITICAL)

Start by listing the root directory, then explore key areas. Read dependency files and configuration files. Produce the JSON object described above.

Appendix 5. Use of artificial intelligence in the thesis

This thesis has utilized artificial intelligence tools in the following ways:

Tools and purpose of use:

- **Grammarly**: Improving language quality, grammar, and clarity of the text
- **ChatGPT (OpenAI)**: Supporting brainstorming of ideas
- **Claude Code (Anthropic)**: Supporting the development of the prototype

Artificial intelligence tools were used to support specific aspects of the work, including language refinement, ideation, and technical implementation. All AI-assisted outputs were reviewed, critically evaluated, and modified by the author. The analyses, interpretations, and conclusions presented in this thesis are author's own work.