Agentic AI for Autonomous Risk Triage and False Positive Reduction in Static Security Analysis

University of Turku
Department of Computing, Faculty of Technology
Master's Degree Programme in Information and Communication Technology
Cyber Security Engineering
June 2025

Author:
Sumit Bakane

Supervisor:
Saku Lindroos (University of Turku)
Tahir Mohammad (University of Turku)

UNIVERSITY OF TURKU
Department of Computing, Faculty of Technology

Sumit Bakane: Designing Agentic AI for Autonomous Risk Triage and False Positive Reduction in Static Security Analysis

Master's Degree Programme in Information and Communication Technology, 88 p.
Cyber Security Engineering
June 2025

To detect the flaws early in the software development cycle, static security analysis techniques are largely adopted in modern software development. These tools, however, have poor risk prioritization abilities, fragmented results, and high false positives that result in substantial manual effort and delayed or even ignored fixing. Additionally, as software development increasingly relies on automated security tools, the challenge of managing high volumes of false alerts has become critical.

Through the integration of rule-based heuristics and large language model-based reasoning in a modular, agent-based framework, this thesis proposes a novel Agentic AI system that aims to work on these limitations. The system normalizes the output of various static analysis tools, such as GitLeaks (detection of secrets), OWASP Dependency-Check (SCA), and Semgrep (SAST), into one schema prior to sending it through a pipeline of autonomous agents. This normalisation process makes this approach tool-friendly, as any security tool can be easily integrated with Agentic AI architecture through normalisation layer.

By employing a multi-layered architecture within the CI/CD Pipelines, the system processes raw outputs into actionable insights, significantly improving developer efficiency and trust in security tools. Empirical evaluation demonstrates a 35.29% reduction in false positives, alongside improved risk prioritization and user engagement through a web-based dashboard.

In addition, the system is compliant with popular secure development guidelines, including OWASP SAMM, BSIMM, and NIST SSDF, that ensure compliance with standards, auditing, and traceability. Overall, this research contributes to the field of application security by providing a scalable, intelligent solution that aligns with industry standards and enhances the overall security posture of software development lifecycles.

Keywords: Static Security Analysis, False-positive reduction, Agentic AI, Application Security, Agentic AI in Application Security, Autonomous Risk Triage

Contents

List of Figures

List of Tables

List of acronyms

AI Artificial Intelligence

API Application Programming Interface

AppSec Application Security

CI/CD Continuous Integration and Continuous Deployment

CIS Center for Internet Security

CSIRT Computer Security Incident Response Team

CVE Common Vulnerabilities and Exposures

CVSS Common Vulnerability Scoring System

DAST Dynamic Application Security Testing

ENISA European Union Agency for Cybersecurity

EPSS Exploit Prediction Scoring System

EUVD European Union Vulnerability Database

FP False Positive

GHA GitHub Action

IAST Interactive Application Security Testing

IDEs Integrated Development Environment

KPIs Key Performance Indicators

LLMs Large Language Models

NIST National Institute of Standards and Technology

NLP Natural Language Processing

NVD National Vulnerability Database

OASIS Organization for the Advancement of Structured Information Standards

OWASP Open Worldwide Application Security Project

PoC Proof-of-Concept

RAG Retrieval-Augmented Generation

SAMM Software Assurance Maturity Model

SARIF Static Analysis Results Interchange Format

SAST Static Application Security Testing

SBOM Software Bill of Materials

SCA Software Composition Analysis

SDLC Software Development Life Cycle

SMEs Small and Medium-sized Enterprises

SOC Security Operations Center

SSDLC Secure Software Development Lifecycle

TP True Positive

1 Introduction

In this age of technology, software supports almost every facet of the modern life, ranging from communication and finance to healthcare and critical national infrastructure. Around 5.56 billion people, which represents 67.9% of the total world's population, were internet users, as reported on February 2025 by Statista [1].

Moreover, almost 90% of organizations worldwide are actively undergoing some kind of technological revamp, indicating that the digital transformation has become a universal goal [2]. In addition to this, the European Union has also set two ambitious goals for 2030, with the first one being, over 90% of SMEs should attain at least a basic level of digital intensity. Secondly, 75% of businesses must adopt advanced technologies like Cloud Computing, Big Data Analytics, or Artificial Intelligence (AI) [3].

These trends backed up by the data, clearly pointing to a generational transition towards a world that is becoming more and more dependent on the digital applications, often referred to as Software. This huge dependency on software also concerns its security. Robust Application Security (AppSec) becomes essential as societies become more reliant on complex software systems. It is also becoming a crucial component of resilience and trust in this age of technology.

As the complexity of system increases, it creates more room for vulnerabilities, and as digital threats evolve in complexity and volume, it is now essential for indi-

viduals, businesses, and governments to ensure secure software development.

1.1 Motivation

In the software development life cycle (SDLC), incorporating the security practices in early phases, also commonly referred to as "Shifting Security Left" is considered essential. According to National Institute of Standards and Technology (NIST), identifying and fixing security issues early in the SDLC saves resources including cost, efforts and time as compared to post-deployment fixes [4].

This has also given rise to the principle of "Secure-by-Design" and embedding security practices from the very beginning of the SDLC [5]. This shift-left approach has made the use of software composition analysis (SCA) and static application security testing (SAST) technologies very important, as it helps identifying vulnerabilities in the earlier phases, even before the deployment of software. However, these tools are not fully perfect. Their tendency to generate a large number of false positives is a frequent problem in AppSec domain [6]. False positives are kind of alerts that are flagged as security issues but are actually irrelevant or not exploitable. This overload not only consumes time, causes fatigue, but also could hide up the genuinely critical vulnerabilities [7]. It is usually the responsibility of the security and development teams to go through these results and do the filtering manually. But the main problem with manual triage is that, it's a time-consuming process, inefficient, and prone to errors [8].

The rise of reasoning-based automation, autonomous AI agents, and large language models (LLMs) presents a unique opportunity to redesign this triage process. Autonomous systems that can understand, filter, and reason about security results can be called as Agentic AI, and it can help significantly reducing the workload for human operators. These algorithms can also help prioritize genuine risks, filter out

irrelevant noise, and deliver results in a more unified and understandable way.

This thesis is motivated by the need to enhance the way, how security issues are triaged and managed. The aim is to greatly lower the false positives, save developer time, and improve overall visibility into software security threats by utilizing Agentic AI leveraging LLMs.

1.2 Problem statement

Despite multiple significant advancements in static security analysis and vulnerability detection tools, software development as well as security engineering teams continue to deal with three interrelated and persistent challenges that undermine effective security implementation

First is about high volume of false positives. SAST tools often generate an excessive amount of alerts, and shockingly most of which are false positives [9]. These false positives force developers and security engineers to spend excessive time manually verifying issues, which leads to alert fatigue, wasted effort, and an increased likelihood of overlooking real threats, most often when dealing with the tight deadlines [10].

Secondly, time-intensive and manual risk triage. The overall process of assessing and separating false positives from genuine risks generated as a result from security tools, takes a lot of time [9]. Since manual triage mostly relies on individual expertise and reasoning, it not only slows down the development cycle but also adds inconsistencies across the workflows [10].

Third is the lack of unified visibility across tools. Multiple SAST, SCA, and CI/CD-based security scanning solutions are used by many development teams across various organizations working under modern development life-cycle (i.e uses

CI/CD). However, these tools produce different results that have various priorities, formats, and categorization. This leads to lack of visibility across the whole security landscape and as a result, it is challenging to pinpoint the most important problems or efficiently track remediation progress [11].

1.3 Research question

To address all the challenges outlined in problem statement, this thesis aims to find answers to the following research questions:

1. *Research Question 1 (RQ1)*: How can reasoning agents and LLMs be used to differentiate between true and false positives in static analysis results?

2. *Research Question 2 (RQ2)*: How can results from multiple security tools be combined and standardized to present a unified and consistent view of risks?

3. *Research Question 3 (RQ3)*: How can a web-based interface be designed to efficiently support developer workflows, offering clear visualization of triaged risks and actionable remediation insights?

4. *Research Question 4 (RQ4)*: To what extent can such a system reduce the overall triage time, minimize the manual effort, and improve the developer trust and engagement with security tools?

1.4 Research objective

To tackle the urgent issues presented by the noisy and unreliable results from static security analysis tools, this thesis introduces an Agentic AI framework that combines LLMs, multi-step reasoning, and self-directed decision-making. The study defines the following four core objectives:

- To investigate at how LLMs and reasoning agents may be combined to analyse security scan results and reliably differentiate between false positives and real positives in static security tools.
- To build a unified triage framework that uses an agentic reasoning pipeline and shared memory model to aggregate, normalise, and present outputs from various static analysis tools in a consistent and a more organised manner.
- To design and develop a safe, production-ready web interface that provides remediation guidance, severity breakdowns, and the contextual risk insights, efficiently supporting developer workflows and improving the overall usability and clarity of the triaged results.
- To evaluate the system in terms of its ability to reduce the false positives, decrease the manual risk triage time, and enhance the developer trust in security tools by conducting empirical testing on real-world vulnerable codebases and comparing the results with manual analysis.

1.5 Scope of the work

The scope of this research is focused on providing a practical and research-driven contribution to the field of AppSec by demonstrating how agentic AI can transform the risk triage process, particularly in the static security analysis. The core contributions of this work are as follows:

1. *Designing of Agentic AI based solution for Security Triage*: This research proposes and implements an AI agent that automatically filters and prioritizes security issues identified by static analysis tools using large language models (LLMs) and multi-step reasoning.

2. *Proof-of-Concept (PoC) System for False Positive Reduction*: To demonstrate the AI system's capability to reduce false positives in real or simulated SAST/SCA tool outputs, a completely working prototype is created, allowing developers to concentrate more on true risks.

3. *Interactive Dashboard for Enhanced Threat Visibility*: The thesis delivers a web-based application that provides actionable insights and remediation methods, evaluates true vs. false positives, and visualizes triaged outcomes, and all this in one centralized platform.

4. *Unified Integration of Multi-Tool Outputs*: An approach has been built to handle the common problem of fragmented visibility in multi-tool situations by processing and normalizing outputs from several static analysis tools.

5. *Empirical Evaluation*: Key performance indicators (KPIs) like the false positive reduction rate and time savings are used to assess the effectiveness of the system.

1.6 Structure of the thesis

This thesis is organized into six chapters, each providing a comprehensive understanding of the research, design, implementation, and evaluation of an agentic AI system for the autonomous risk triage in static security analysis.

Chapter 1: Introduction - provides an overview of the research context, the motivation behind it, then identifies the core problems, outlines the research questions and objectives. The scope and contributions of the thesis are also presented.

Chapter 2: Literature Review - highlights the foundational concepts of secure software development, static analysis, security testing and application security. It

also addresses the issue of false positives, thoroughly examines the role of AI and LLMs in cybersecurity, and identifies existing gaps in current solutions.

Chapter 3: Research Methodology - presents the proposed agentic AI solution, including system and agent architecture, data sources, reasoning logic, and the evaluation framework. The selected security tools, datasets, and alignment with established security standards are also described.

Chapter 4: System Implementation - describes the practical implementation of the prototype system. This covers the ingestion, processing, and classification of security findings and the display of filtered outputs through a web-based dashboard.

Chapter 5: Results - presents the evaluation of the system's performance. It includes false positive reduction statistics, classification accuracy, and a comparison between manual and agentic triage.

Chapter 6: Conclusion - summarizes key findings, addresses the research questions, discusses limitations of the current approach, and explores potential directions for future enhancements.

Statement on the usage of AI: *This thesis is the original work of the author. All content has been written by the author, using information from various scientific and academic sources that have been properly cited. The thesis includes a working prototype, and during its development, the author encountered some programming-related errors. AI tools like ChatGPT and GitHub Copilot were utilised to resolve the issues. Apart from this, the entire process, from conceptualisation to the development and deployment of the prototype, was done by the author alone.*

2 Literature review

The fundamental concepts of secure software development, static analysis, security testing, and AppSec are highlighted in this chapter. Along with addressing the problem of false positives, it also carefully investigates the function of AI and LLMs in cybersecurity.

2.1 Secure Software Development Lifecycle (SSDLC)

Software Development Lifecycle (SDLC) is a structured process that is used by development teams to create the software systems from the very scratch till its completion. Some examples of traditional SDLC models such as Waterfall, Agile, and DevOps, place a strong emphasis on phases like requirements gathering, design, development (which includes coding and building), then deployment and testing, and maintenance at the final phase. Figure 2.1 shows the different phases of SDLC and brief information about each phase. These traditional SDLCs are effective at managing the delivery of functional software, but they often lacks in security consideration, making security a post-development concern or an afterthought [12].

In order to mitigate the security risks, the Secure Software Development Lifecycle (SSDLC) was introduced. It embeds the security elements in each phase of SDLC, which makes SSDLC a proactive approach rather than a reactive one unlike

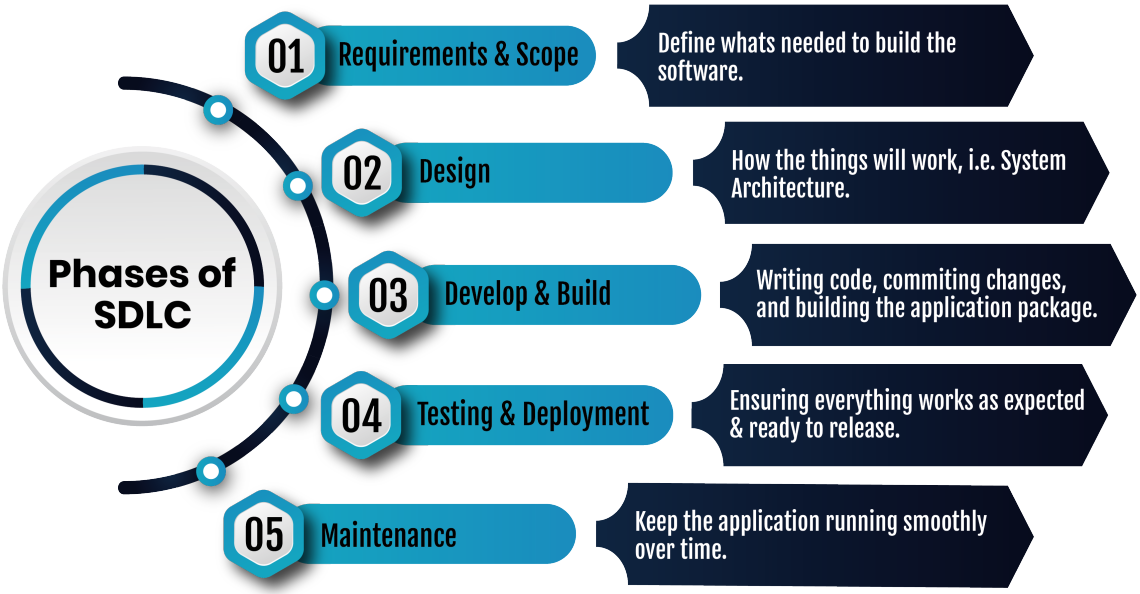

Figure 2.1: Phases of Software Development Lifecycle (SDLC)

SDLC. The Open Web Applications Security Project (OWASP) states that as part of the standard workflow, an SSDLC includes some common practices such as code scanning, threat modeling, secure design review, and security testing [13].

Due to the growing complexity of software systems and the ever-evolving threat landscape, modern software engineering has gone through a paradigm shift, as can be verified by the switch from SDLC to SSDLC. The objective is not only to create functional software, but to make sure that the software is secure by design.

2.1.1 Security involved in the phases of SDLC

In an SSDLC, security is not only a one-time checkpoint but rather a continuous process that is integrated into each phase of SDLC. As seen in Figure 2.2, every stage of the SDLC has a distinctive set of security procedures and objectives embedded into it.

The security components associated with each phase are discussed below:

- **Requirements Phase**: All the security requirements should be gathered at

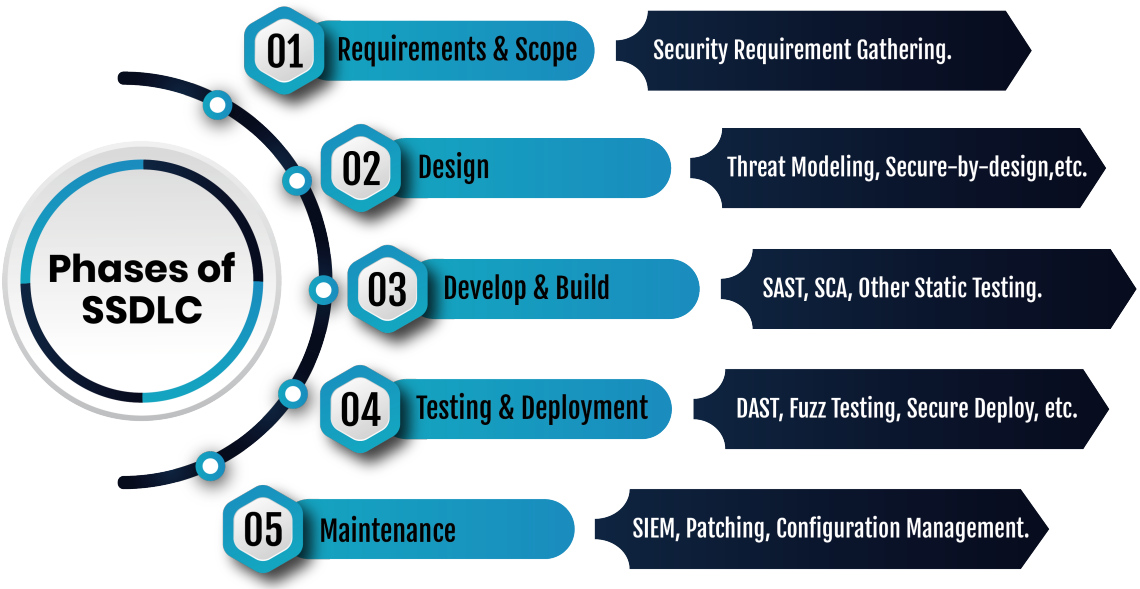

Figure 2.2: Phases of Secure Software Development Lifecycle (SSDLC)

this stage which includes requirements based on business, legal, and compliance needs. Well-defined functional and non-functional requirements, including data protection, access control, and authentication can also be added depending on whether new feature is being added or change in existing feature. A number of maturity models are available, including the Security Assurance Maturity Model (SAMM), Building Security in Maturity Model (BSIMM), and Capability Maturity Model Integration (CMMI) to support structured approaches in order to capture and manage the security needs. It also helps teams to improve security during the requirement phase [14].

- **Design Phase**: Here, threat modeling is applied to identify potential security flaws in architecture. Threat models like STRIDE (widely used), DREAD (useful for risk scoring) and PASTA (useful in complex environment) and techniques like DFD-based analysis help evaluate risk exposure and design appropriate countermeasures before coding begins [12], [15].

- **Development and Building Phase**: Secure Coding Guidelines can be fol-

lowed at this stage to avoid common software defects like buffer overflows, injection attacks, and insecure APIs. Additionally, in order to ensure that security coding standards are followed, Static Testing tools (SAST, SCAs) should be used [12], [15].

- **Testing and Deployment Phase**: In addition to functionality testing, security testing methods such as DAST and Fuzz testing ensures the software is resilient against known attack vectors, as it scans for vulnerabilities in application runtime. Moreover, Center for Internet Security (CIS) benchmarks can be followed to ensure that the deployment or production environment stays safe from threats which includes configuration scanning, verification of security controls, and policy level security rules like access management [12], [15].

- **Maintenance Phase**: Security does not end at deployment. Software security need to be maintained even after production. So to handle vulnerabilities found after a release, regular security patching, monitoring, and incident response procedures must be at the place [12], [15].

SSDLC approach of integrating security practices at every stage of software development comes with several advantages. One of the main advantage is the cost reduction in patching vulnerabilities, and increase in risk visibility throughout SDLC. It is 100 times cheaper to fix the security issue if it is found much earlier in SDLC, which supports the idea of a fully integrated SSDLC [16].

2.1.2 Importance of early-stage security (Shift-Left approach)

The “Shift-Left” security concept on incorporating security procedures into software development as early as possible, i. e. shifting security left which means focusing on security before working on the each stage of SDLC. In the past, security assessments were mostly conducted after deployment or during testing, which resulted in high

remediation costs and also the delayed releases [12]. The Shift-Left methodology promotes integrating security into the planning, design, and development stages in order to identify vulnerabilities early on, which simplifies the fixing process, takes less time, and is less expensive.

According to an IBM System Science Institute report, the cost of fixing a vulnerability in the design stage is around one-sixth of what it would be in post-production [16]. This shows the operational and the financial benefits of early security adoption. Shift-Left approach may help security and development teams work together more efficiently, as it automate secure coding techniques, and reduces the need for reactive security reviews. Developers are able to receive real-time security feedback using tools like pre-commit hooks, static analyzers, and scanners integrated into their Integrated Development Environments (IDEs) or in CI/CD pipelines, transforming security into a continuous and proactive discipline [17].

2.1.3 Role of AppSec in modern development environments

AppSec is now a core pillar of modern software engineering, and has evolved from being a specialized post-development task. In recent DevSecOps and Agile contexts, where continuous deployment and rapid iterations are standard procedures, AppSec acts as a facilitator rather than a bottleneck.

Initially, security was mostly network-focused, depending on firewalls and other perimeter defenses and trusting that apps were secure inside internal networks. A reactive security posture focused on patching misconfigurations and security issues after deployment resulted from the early 2000s rise of web applications, which uncovered application-layer vulnerabilities. Organizations started to realize how ineffective this strategy was by the early 2010s, which led to a wide shift to proactive security integration in the development process. As part of an overall effort to incor-

porate security from the very start, secure SDLC processes such as threat modeling, code reviews, and static analysis were developed. In mid-2010s, with the emergence of the DevSecOps powered by DevOps and Agile, AppSec has adapted to continuous delivery by integrating automated security tools into CI/CD pipelines and encouraging shared accountability fostering collaboration between development and operations teams [12]. Figure 2.3 examines the phases of evolution as well as limitations and focus areas for each stages.

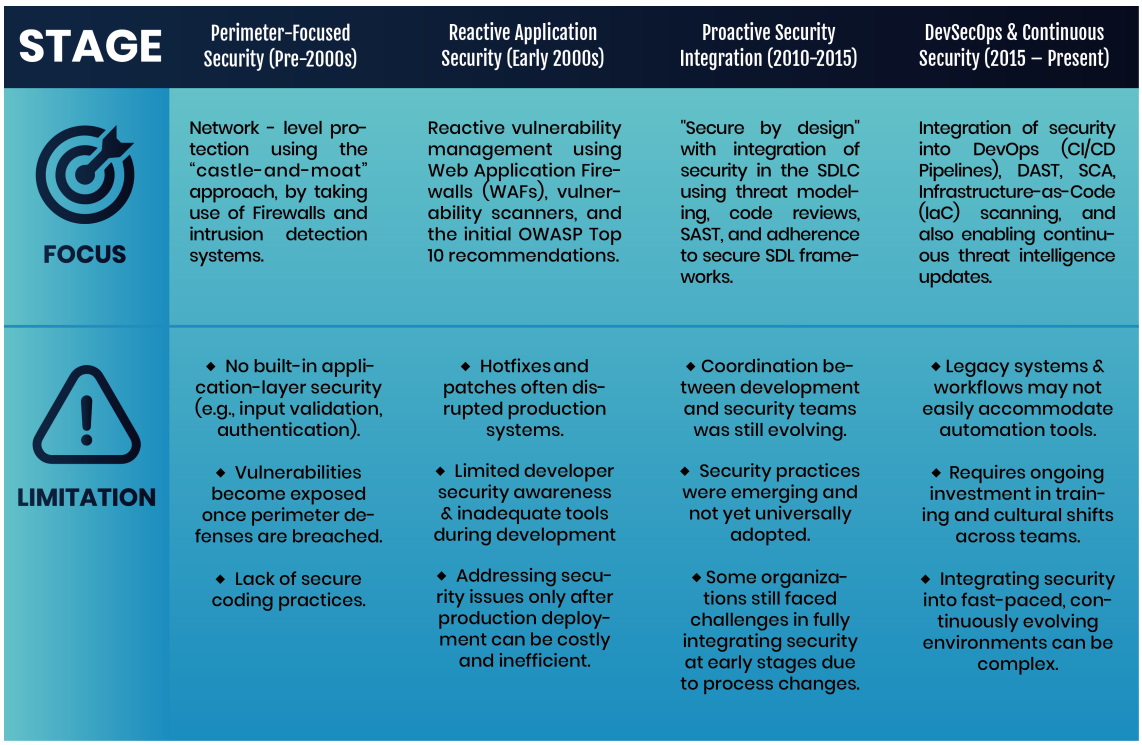

Figure 2.3: Evolution of AppSec in Modern Software Development

Development teams, security engineers, and operations teams work closely together in modern AppSec practices to deliver the efficient, robust and secure applications. This involves automating policy enforcement, enabling developer-first tools and integrating security scanners into CI/CD pipelines which helps in the early detection and fixing of vulnerabilities. The overall goal of this is to create secure-by-default systems without compromising the development speed. As a result, AppSec is now a proactive, collaborative, and continuous process that supports the creation

of secure and scalable software rather than a reactive gatekeeper function.

2.2 Risk Management in Software Development

In software development, risk management is a key process which helps in identifying, assessing, and eliminating possible issues that may impact the success, security, or functionality of a software system. Threat modeling is an important component in this process, which involves systematically identifying potential security threats, vulnerabilities, and attack vectors within the system. After threats have been identified, risk assessment is carried out to determine each threat's impact and likelihood, enabling teams to understand their severity. Based to this analysis, prioritisation helps in efficient resource allocation by addressing the most important threats first. Together, these practices ensure that potential problems are managed proactively, reducing the likelihood of project failure or security breaches.

2.2.1 Introduction to threat modeling

Threat modeling is one of the fundamental method in secure software design and a recommended security practice related to the SDLC design phase. Commonly used threat models such as STRIDE, DREAD, PASTA, are used to systematically identify, list, and prioritize potential risks that could bring damage to the system if unresolved.

Microsoft created the widely used threat model known as STRIDE, which evaluates a system's complex architecture based on the six threat categories: Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, and Elevation of Privilege [18]. Table 2.1 lists these six primary threats, their definitions, and security control and mitigation strategies [19]. A system's architecture and data flow is generally used to qualitatively evaluate possible risks using this threat model.

DREAD, which stands for Damage Potential, Reproducibility, Exploitability, Affected Users, and Discoverability which can be seen as five questions about each potential threat. Table 2.2 describes each of this category briefly. It is a methodology which ranks threats, by assigning severity and priority level to the identified threats. DREAD is a quantitative approach as it uses numerical scores to assess and contrast risks [20].

Threat	**Definition**	**Security Control**	**Mitigation Strategy**
Spoofing	Falsely posing as someone or something other than myself.	Authentication	Enforce strong authentication and avoid storing or exposing secrets.
Tampering	Making changes to the disk, network, memory, or elsewhere.	Integrity	Use integrity checks and secure authorization to prevent unauthorized changes.
Repudiation	Not taking accountability for an action: may or may not be true.	Non-Repudiation	Enable accountability with digital signatures, timestamps, and audit logs.
Information Disclosure	Giving information to an unapproved entity.	Confidentiality	Encrypt data, limit access via authorization, and avoid unnecessary secret storage.
Denial of Service	Exhausting the resources required to provide services.	Availability	Mitigate with authentication controls, request filtering, and QoS mechanisms.
Elevation of Privilege	Permitting someone to perform unauthorized tasks.	Authorization	Enforce least privilege and restrict access based on minimal necessary permissions.

Table 2.1: Overview of STRIDE Threat Model

Category	**Description**
Damage Potential	To what extent might this threat harm the system?
Reproducibility	Is this threat easy to replicate?
Exploitability	How much experience and effort is needed to exploit the threat?
Affected Users	How many users could be affected if the threat is realized?
Discoverability	How simple is it to identify the vulnerability?

Table 2.2: Overview of DREAD Threat Model

Process for Attack Simulation and Threat Analysis, or PASTA for short, is a risk-centric threat model. On a more strategic level, it involves the threat modeling process, which includes threat analysis and also the mitigation strategies. As it integrates attacker modeling with business risk, it is both a qualitative and quantitative threat model at the same time [21].

Teams can use these models to identify the attack surfaces, assesses the likelihoods, and develop mitigation mechanisms early in the development lifecycle. The overall objective is to understand what might go wrong and then proactively designing countermeasures mechanism before the software is developed.

2.2.2 Risk Triage: Risk Assessment and Risk Prioritization

Risk Triage is the practice of categorizing and prioritizing security threats according to their potential impact and possible likelihood to exploitation [22]. This involves deciding which vulnerabilities in software security need attention so that it can be fixed right away by evaluating the severity of vulnerabilities reported by tools or manual reviews.

Effective risk triage consists of two primary activities:

- **Risk Assessment**: Identifying the kind and scope of a vulnerability like the data that could be compromised, the asset it impacts, and whether the vulnerability is accessible from outside the system or not [23].
- **Risk Prioritization**: Determining a risk score or severity level to plan for remediation activities. Common Vulnerability Scoring System (CVSS) and Exploit Prediction Scoring System (EPSS) are some of the common frameworks that offer standardized scoring procedures based on impact and exploitability [24].

Risk triage guarantees that security teams concentrate on the most serious concerns first rather than treating every issue with the same urgency, which can be relate very similar to medical triage in an emergency department, when patients are treated depending on urgency [25].

Triage helps prevent resource dilution by ensuring that teams concentrate on vulnerabilities with the highest business and technical risks, even in the presence of thousands of security findings [26]. Improper triage raises the possibility that critical issues might go unnoticed, and also slows down the remediation process.

2.2.3 How inaccurate prioritization leads to wasted resources

One of the most expensive inefficiencies in modern AppSec is inefficient risk prioritization [27]. Teams that treat all vulnerabilities with equal importance, often spend a lot of time fixing low-risk problems while leaving major ones unfixed.

A 2025 Veracode State of Software Security report states that 49.9% of organizations have high-severity issues that have not been addressed, and 74.2% of organizations have security flaws that have been unremeditated for more than a year [28]. The reason behind this often redirects to the triage bottlenecks and alert fatigue. This imbalance creates a false sense of security and shows inadequate risk coverage. Too many false positives can cause developers to ignore scanner results, reducing trust in security tools and potentially causing more harm than the vulnerabilities themselves.

2.3 Static Security Analysis Techniques

The method of analyzing source code against issues without running the program is known as static code analysis. The goal is to identify potential vulnerabilities, issues

with the quality of the code, or even security policy violations before the program is executed and not seen by the compiler [29].

2.3.1 Fundamentals of static code analysis (SAST and SCA)

Two dominant approaches in static security analysis are:

1. SAST focuses on examining the actual codebase to find security issues including buffer overflows, injection flaws, and insecure usage of APIs [30].

2. SCA focuses on finding known vulnerabilities in open-source dependencies, libraries, and third-party components that are used in a project [30].

These tools offer several benefits when combined with CI/CD pipelines, and most importantly can offer i) a good scalability; ii) the detection of known and documented vulnerabilities; and iii) reporting which illustrates the location of the problem for an easier fix, thus cutting down on time-to-fix [30].

"Shift-Left" security strategies strongly depends on SAST and SCA as it enable the identification and fixing of vulnerabilities during development. According to The Github Octoverse 2024-The State of Open Source Software report, it was found out that over 39 million secret leaks were identified by developers on GitHub using a static analysis technique called secret scanning [31]. Additionally, according to Synopsys 2024 Open Source Security and Risk Analysis Report, 96% of the apps that were examined have open-source libraries [32]. These numbers highlight the growing importance of static security analysis.

2.3.2 Importance of SBOM in open-source security

Software Bill of Material (SBOM) is an organized inventory of all the components used in an application, that involves third-party libraries and dependencies. SBOMs

are a fundamental element of visibility, traceability, and vulnerability management in modern software development, where the reuse of open-source resources are common [33]. Components of SBOM can be seen in Figure 2.4.

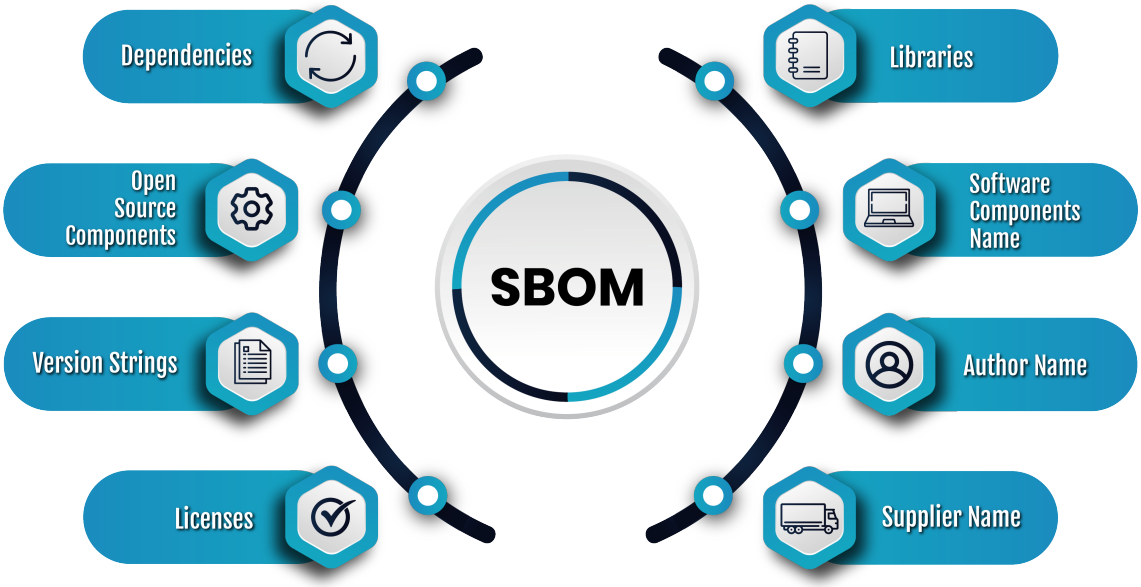

Figure 2.4: Components of SBOM

As supply chain attacks have become more prevalent, SBOMs and related security has become much more important. Maintaining a well-defined SBOM enables organizations to promptly determine if they are impacted by newly disclosed vulnerabilities such as Log4Shell and take appropriate countermeasure [33].

In order to promote proactive, scalable open-source risk management, SBOMs with SCA tools, function together to automatically discover known vulnerabilities, license compliance issues, and outdated packages.

2.3.3 Benefits of Static Security Analysis over Dynamic Testing

Compared to dynamic testing techniques, static security analysis has several benefits, particularly in the early phases of development. SAST examines the source

code, bytecode, or binaries without even running the program, unlike Dynamic Application Security Testing (DAST), which examines applications while program execution. This makes it possible to find issues long before the application is released [30].

By detecting vulnerabilities early in the development process, SAST technologies allow developers to address problems before they're deeply rooted in the application. Since the codebase is directly analyzed, all execution pathways, including those that are difficult to access dynamically (like vulnerabilities in libraries), may be investigated. Faster feedback loop is one of the main advantage. As these static analysis tools when Integrated into CI/CD pipelines or IDEs, provides real-time feedback, making them well-suited for modern Agile and DevOps environments [30].

2.3.4 Limitations of static approaches compared to dynamic methods

Static analysis methods have significant drawbacks that limit their usefulness when used alone, notwithstanding their benefits. Their incapacity to monitor runtime behavior is one of the main issues. This means that issues that only show up during and after execution, such as session management faults, authentication bypasses, and logic errors, could go undetected.

The high false positive rate in static analysis is again a serious problem, as vulnerabilities that are detected might not be exploitable in practical situations [34]. This often leads to developer fatigue and wasted remediation efforts. Furthermore, code that significantly depends on runtime configuration, reflection, or dynamic language features is usually difficult for static tools to analyze. This can make control and data flows difficult to observe during static testing [30].

Static analysis is enhanced by dynamic testing, which simulates real-world at-

tacks on running applications about actual exploitability. Therefore, while static analysis is a beneficial in early-stage detection, but its limitations highlight the importance of using it as part of a hybrid or layered security testing strategy.

2.4 Security Scanning Tools and Their Challenges

There are variety of static security testing tools available in the market, that can be employed in to the modern software development workflows. As a result, vulnerabilities can be found at much earlier phases of the SDLC. These tools can be broadly classified into SAST, SCA and Secret detection tools.

2.4.1 Overview of common and open-source tools

Different static analysis open-source tools is further categorised based on some classification factors such as programming languages support, integration with CI/CD pipelines, OWASP top 10 mapping support, compatibility with SARIF and ease of setup [30]. Table 2.3 describes various open-source SAST tools and compares them using the key classification factors. SCA tools scans for the vulnerabilities in the third-party libraries, and comparison of such tools are presented in Table 2.4. Various Secret Detection/Scanning tools are good at identifying accidental exposure of sensitive data like hardcoded API Keys or crendentials. Such tools are discussed in table 2.5.

SAST Tools	**Supported Languages**	**CI/CD Friendly**	**OWASP Top 10**	**SARIF Compatiability**	**Setup Simplicity**
Semgrep CE	Multi-Language	Yes	Yes	Yes	Easy
SonarQube CE	Multi-Language	Yes	Yes	No	Moderate
Bandit	Python	Yes	Yes	Yes	Easy
ESLint	JavaScript	Yes	Yes	Yes	Easy

Table 2.3: Open-source SAST Tools

SCA Tools	**Supported Languages**	**CI/CD Friendly**	**OWASP Top 10**	**SARIF Compatiability**	**Setup Simplicity**
OWASP DC	Multi-Language	Yes	Yes	Yes	Easy
Trivy CE	Multi-Language	Yes	Yes	Yes	Easy
Syft+Grype	Multi-Language	Yes	Yes	Yes	Moderate

Table 2.4: Open-source SCA Tools

Scanning Tools	**Supported Languages**	**CI/CD Friendly**	**OWASP Top 10**	**SARIF Compatiability**	**Setup Simplicity**
GitLeaks	Language Agnostic	Yes	Yes	Yes	Easy
Detect-Secrets	Language Agnostic	Yes	Yes	No	Moderate
TruffleHog	Language Agnostic	Yes	Yes	No	Moderate
SecretLint	Language Agnostic	Yes	Yes	No	Easy

Table 2.5: Secret Scanning Tools

2.4.2 Use cases: SAST, SCA, and security scanner tools

In the software development lifecycle, security scanning tools have different but related functions. There are various type of tools available and each of the kind is designed to identify particular types of vulnerabilities.

- **SAST Tools** examine the source code of the program to find code-level problems such as improper data handling, hardcoded credentials, or SQL injection. They are useful for secure code enforcement and early vulnerability identification. During the coding stage, developers frequently utilize SAST, which is integrated into IDEs or pre-commit hooks [17].

- **SCA Tools** focus more on third-party risks. Open-source packages are widely used, and SCA tools detect license issues, known vulnerabilities (CVEs), and outdated dependencies. They help ensure that reused components do not introduce systemic risk [30].

- **Secrets Detection Tools** identify sensitive information that might have been accidentally committed to version control, such as hardcoded tokens, API keys, and passwords. These tools are vital in preventing the data exposure or unauthorized access due to developer oversight [30].

Different attack surfaces related to static analysis are addressed by of these categories of tools. Together, these tools enable secure development techniques throughout the SDLC and allow a layered defense strategy that covers the security of code, components, and credentials. Not a single tool can identify each and every defect or security issue [35]. Instead, a combination of various methods can help cover the static vulnerability landscape [35].

2.4.3 Problem of fragmented outputs and tool interoperability

Security teams frequently combine SAST, SCA, secrets detection, and license compliance techniques in enterprise settings. These tools produce results in various formats, severities, and taxonomies. This complicates remediation operations and results in fragmented visibility over the security posture.

To address interoperability challenges in static analysis tools, the Static Analysis Results Interchange Format (SARIF) was introduced as a standardized, JSON-based schema for representing static analysis tool outputs. SARIF aims to unify reporting across tools by defining a consistent structure for rules, locations, severity levels,

and remediation guidance. It is currently supported by various tools and recognized as an OASIS[1] standard [36].

Although not all tools provide native support, SARIF compatibility varies across vendors. It is often necessary to use custom parsers or normalization layers when integrating results from legacy tools or combining different types of scanners (such as SCA with SAST) even with SARIF. This lack of uniformity causes silos between the development and security teams and slows down triage.

2.5 False Positives in Static Analysis

The main causes of false positives in static analysis are the static modeling, over-generalized detection rules, and lack of program context. These problems are particularly common in applications that generally use dynamic constructs or large, and modular codebases. According to the Orca 2022 Cloud Security Alert Fatigue Report, 43% of the surveyed organizations across various sectors, using cloud environment for their operations have 40% false-positive rate in general. False alerts are far more than just a nuisance, as they actually have a significant impact on the overall economy of an AppSec program [37].

In SCA, the main cause of false positives is the dedicated vulnerability reporting based on dependency graphs. These tools may flag a vulnerability for a library even when the vulnerable functionality is not called by the application or if it is addressed by some other methods, such as secure configurations or runtime controls [30].

Some of the additional Key Causes include:

- **Lack of Runtime Context:** Static tools are unable to evaluate conditions

[1]OASIS stands for the Organization for the Advancement of Structured Information Standards, and is a non-profit multinational collaboration that develops and promotes open standards for the global information society.

that are resolved at runtime, such as dynamic routing and user-specific logic, as they examine code without executing the application [30].

- **Incomplete Data Flow Modeling:** Many SAST tools have limited understanding of the complex data and control flows, particularly across microservices environments, which often leads to assumptions that may not reflect actual behavior [38].

- **Conservative Heuristics:** Rule sets are often designed to identify as many potential vulnerabilities as possible (broadcast range), even at a cost of precision and accuracy, which leads to overreporting [39].

- **Third-Party Dependencies Misclassification:** Particularly in cases where dependency trees are deeply nested or SBOMs are too inclusive, SCA tools may flag vulnerabilities in unused or unreachable code paths of dependencies [30].

- **Code Patterns That Are Secure by Design:** Some secure coding techniques can syntactically mirror insecure patterns. For example, sanitized input passed through an abstraction layer may still be marked as a potential injection point, which leads in generating false alert.

- Furthermore, Muthukrishnan et al. [40] further point out that cross-tool intelligence sharing is often prevented by the compartmentalized nature of DevSecOps tools, each of which uses its own AI/ML model. This increases the possibility of false positives since SAST and SCA tools generate isolated predictions without understanding the larger CI/CD or runtime environment [40].

All these causes highlight the necessity for contextual reasoning and enhanced intelligence when filtering the results of static analysis, an area where agentic AI can provide significant improvements.

2.5.1 Impact on developer workflows and security posture

When false positives becomes common and continue to occur in security scans, then it leads to alert fatigue among developers, causing frustration and erosion of trust in security tooling. Developers often waste their significant time triaging these non-issues, which disrupts the sprint velocity and delays the legitimate remediation efforts [40].

Notably, a high false positive rate was regarded as the top frustration by 42% of the 900 security practitioners surveyed as per Tines 2023 Voice of the SOC report [41]. Excessive noise from scanners can create delays in releases or can cause teams to bypass the security checks in continuous delivery environments where speed and reliability are critical [40]. This reduces the efficacy of AppSec programs, which degrades the security posture at the organizational level. Additionally, it raises the possibility of vulnerability debt, which is nothing but the accumulation of unresolved issues over time that complicates long-term risk [37].

Additionally, this false alerts tolerance has wider security implications. This tolerance increases the possibility of developers missing real vulnerabilities when they ignore or disable specific scans because of getting false alerts too often [30]. Additionally, the lack of a unified security intelligence view affects the organization's ability to perform timely patching, maintain compliance, and also perform effective root cause analysis [40].

Production insights are rarely funneled back into earlier development stages further weakens the DevSecOps feedback loop [38]. This leads to the same scanning inefficiencies throughout several release cycles, as lessons learnt from the operational problems are not used to improve the vulnerability detection logic [40].

2.5.2 Existing approaches to reduce false positives

Over the years, both academic research and industry practices have proposed several approaches to reduce the false positives in static analysis. These can be broadly discussed as follows:

- Most tools allow for suppressing specific rules or adjusting sensitivity thresholds to suppress the generation of specific type of output. By calibrating rules to the context of the application, many irrelevant alerts can be filtered out easily. However, this approach requires significant manual effort and expertise to draft and execute such rules. [42]

- Combining static analysis and LLMs is another option to reduce the false positives. The integration of LLMs with static analyzers enables more context-aware and accurate code review generation. Methods such as Retrieval-Augmented Generation (RAG), Data-Augmented Training, and result concatenation have shown potential in bringing LLM outputs into line with analyzer feedback, enhancing the precision and interpretability of findings. By directly integrating the static analyzer results into LLM inference contexts (RAG, in particular) it helps in effectively reducing false positives [43].

- Neuro-symbolic approaches like Model Generated Code Queries (MoCQ) leverage the strengths of both LLMs and symbolic reasoning that helps reducing the false positives. These systems use LLMs to automate the extraction of vulnerability queries, and use feedback from static analysis to iteratively refine them. By establishing predictions based on symbolic validation, these models significantly reduce the errors that are common in purely learning-based systems [44].

- Empirical analyses of the output of SAST tools shows that some rule patterns and warning types tend to be more prone to errors. Targeting these high-

noise categories with stricter heuristics or improved semantic analysis has been shown to reduce the overall false positives. Inconsistent findings can also be filtered out by combining different tools and cross-validating the results using standardized formats like SARIF [45].

- Using SAST in combination with dynamic validation methods or Interactive Application Security Testing (IAST) is another useful approach to solve the problem of false positives. Through correlation between static results and runtime information, tools can verify if the flagged code is truly vulnerable in a live environment or not. False positives that arise from static-only assumptions are reduced with the use of this kind of hybrid approach [30].

- A comprehensive approach to reduce the false positives is provided by unified AI/ML frameworks trained on organisation-specific DevSecOps data. To enable context-aware alert filtering, these models take into account historical SAST results, code history, test coverage, and operational incidents. Such unified systems improve root cause analysis and alert relevance throughout the CI/CD pipeline in addition to lowering false positives [40].

In summary, current approaches to reduce false positives in the static security analysis ranges from LLM-powered augmentation, neuro-symbolic refinement, dynamic validation, developer-centric suppression handling, to AI/ML-based contextual modeling. These strategies indicate a trend towards the intelligent, integrated, and context-aware static analysis frameworks that can better aligned with developer workflows and support real-world application behavior.

2.6 AI and LLMs in AppSec

In cybersecurity, AI has become a transformative force that has completely changed how organizations detect, prevent, and respond to cyberthreats. AI has played an

important role in protecting the digital infrastructure as a result of its ability to manage massive amounts of data. It helps in recognizing pattern, and adaptively learn from emerging threats as cyberattacks become more prevalent and complex.

2.6.1 Applications of AI in cybersecurity

The integration of AI into AppSec has initiated a paradigm shift from conventional, reactive testing approaches to more proactive, intelligent, and context-aware security mechanisms. By utilizing Machine Learning (ML), Natural Language Processing (NLP), and predictive analytics, AI enables systems to automatically identify, assess, and address vulnerabilities in software applications, with very less human intervention. These technologies serves as the foundation for Adaptive Application Security Testing (AAST), a recent and cutting-edge approach that dynamically adjusts the testing methods in real-time based on user interactions, application behavior, and emerging threats [46].

Following are the use cases of AI in AppSec domain:

- **AI-Driven Vulnerability Identification:** Machine learning models are now central to the modern vulnerability detection strategies. These models are trained using both the supervised and the unsupervised learning techniques to identify security flaws by analyzing patterns in code, system behavior, and historical vulnerability data. Supervised models use labeled datasets to identify code segments as secure or insecure, while unsupervised models detect anomalies that might point to vulnerabilities that haven't been identified yet. This predictive capability allows for the identification of zero-day vulnerabilities and logic flaws that static rules-based systems may overlook at some point [46].

- **NLP for Security Insights:** NLP is essential for improving the overall

contextual understanding of system logs and code. NLP tools help identify the small indications of security flaws or configuration errors by scanning developer comments, issue tickets, and system error logs. This makes it possible for security systems to generate the more personalised vulnerability reports that improves the precision, prioritization, and recommend remedial actions better aligned with developers' workflows [47].

- **Risk Assessment and Predictive Scoring:** AI systems often goes beyond simple detection, and helps in assessing the potential impact and exploitability of the identified vulnerabilities. A wide range of inputs, such as system architecture, user access levels, operational context, and historical exploit data, are taken into account by predictive risk scoring mechanisms in order to prioritize the ranking of vulnerabilities relative to the likelihood of an actual threat. By combining these models with threat intelligence feeds (like MITRE ATT&CK), vulnerabilities that are actively targeted or exploited can be identified with much precision [46].

- **AI-Enabled Remediation Suggestions:** AI also helps with detailed remediation by providing developers code-level recommendations based on best practices and previous patches. The time between detection and resolution can also be greatly reduced by using sophisticated techniques that can even automatically generate patches or suggest "drop-in" code fixes within IDEs. Through reinforcement learning, these systems improve over time by continuously adapting to the developer feedback and resolution outcomes [46].

- **Integration into CI/CD and DevSecOps:** The integration of AI-enhanced security testing into CI/CD pipelines is seamless and in accordance with DevSecOps principles [40]. SAST, DAST, and IAST are all included in adaptive testing frameworks. These frameworks are driven by contextual signals such

as feature deployments, code commits, and the API usage patterns. As a result, on-demand security testing becomes more intelligent, reducing the false positives and test redundancy, and accelerating the delivery of secure software [46].

The implementation of an AI-driven adaptive security testing framework resulted in a 32% improvement in vulnerability detection and a 45% decrease in the false positives, according to a case study within the mid-size fintech company [46]. Additionally, this implementation also encouraged cultural changes in the development teams, establishing security as a fundamental part of the development process rather than being a last-minute requirement. However, labeled data, careful feature engineering, and limited generalization are all requirements for classic machine learning systems, which are currently being addressed by LLMs.

2.6.2 Introduction to LLMs and their potential in AppSec

LLMs have emerged as the transformative tools across a variety of domains, particularly in the field of software engineering and cybersecurity. LLMs are designed to understand and generate human-like language, but their capabilities extend far beyond just NLP. They can demonstrate a deep understanding of the code semantics, logical structures, and even the complex software vulnerabilities.

The application of LLMs in security domains has considerably increased in recent years because of their exceptional contextual and semantic analysis capabilities over static analysis techniques. Although, SAST are good at identifying well-known vulnerability patterns using rule-based techniques, but it faces limitations with complex, context-dependent, or unique threats. On the other hand, LLMs are able to detect vulnerabilities that the traditional tools can miss since they can infer potential risks based on deeper structural patterns [48].

Particularly in the area of AppSec, hybrid approaches such as combining LLMs with SAST results and incorporating RAG techniques produce much better outcomes [43]. These strategies dynamically integrates the up-to-date vulnerability information into the LLMs reasoning process. For instance, by combining the outputs of the traditional SAST tools with tailored vulnerability reports retrieved via semantic and structural similarity searches, LLMs can overcome both their privacy and recency limitations. In addition to enhancing LLMs' detection capabilities, this integration makes it possible for a more adaptable and privacy-preserving security architecture [48]. The integration paves the way for LLM-supported vulnerability scanning systems.

In summary, LLMs hold significant potential to enhance the traditional security tools, provided their limitations are mitigated through careful designs, such as localized deployments, knowledge retrieval integrations, and structured prompting informed by static analysis. As the field matures, LLMs have the potential to become a central component in the next generation of intelligent, and context-aware cybersecurity solutions.

2.7 Agentic AI: Concepts and Use Cases

The term "Agentic AI" describes a class of AI systems that can perceive, reason and act on their own in a predetermined environment to achieve the specified goal without requiring the explicit instructions at every step. Unlike passive models, which generate output in response to prompts, agentic systems behave in a goal-directed manner, usually by decomposing the tasks, planning steps, invoking tools, and learning from the feedback loops.

Agentic AI and AI agents may sound similar, but when it comes to their capabilities, they are very different. AI agents are task-specific which follow predefined rules

to handle repetitive tasks such as scheduling meetings or responding to customer queries. On the other hand, agentic AI demonstrates a higher degree of autonomy, and has the ability to navigate through complex environment and pursue long-term goals. By using iterative planning and adaptive reasoning, it is able to modify its strategies in real time and learn from feedbacks [49], [50].

While AI agents are more constrained and reactive in scope, agentic AI actively perceives its environment, by integrating with various technologies, and reacting proactively to changes. In general, AI agents are better at certain, defined tasks, whereas agentic AI is made for dynamic, and more autonomous tasks [51].

In software, Agentic AI gathers information from multiple sources, such as databases, APIs, and even online searches, to generate perception and context for decision-making [52]. A reasoning engine (often powered by LLMs), memory components, tool integration modules, and a decision-making layer with action sequencing capabilities are the usual components of these systems. Because of their emergent behavior, which enables them to operate semi-autonomously, agentic systems are ideal for tasks involving iterative problem solving and dynamic decision-making.

2.7.1 Agentic AI use cases in software engineering, security, and automation

Agentic AI makes use of adaptive intelligence, which allows it to react dynamically to complex situations and learn from dynamic contexts, in contrast to traditional automation or rule-based systems. It has applications in a number of IT domains, including cybersecurity, software engineering, IT ecosystem automation and orchestration, etc.

Cybersecurity: Threat Detection & Response, and Identity Management

By improving the real-time threat response and automating the routine SOC tasks, agentic AI is significantly advancing cybersecurity operations. AI agents are helping organizations in addressing both internal and external threats more accurately and quickly by streamlining processes such as anomaly detection, incident response, and alert triage [53], [54]. Renowned SOC tools like CrowdStrike's Charlotte AI, then ReliaQuest's GreyMatter platform, have took a step forward and are using Agentic AI technologies to accelerate their threat detection capabilities with precision [49].

Agentic AI also strengthens the identity and access management. Twines digital employee, Alex, proactively identifies and mitigates the vulnerabilities related to unauthorized accesses, reducing operational strain on IT and the cybersecurity teams. As organizations adopt zero trust models, AI agents dynamically adjusts access privileges based on the behavior and real-time risk assessments [49].

Software Engineering: Proactive AppSec and Secure SDLC

Agentic AI is revolutionizing the field of software development and is one of the key element that is delivering a shift from reactive to proactive AppSec. Incorporating AI agents into the SDLC enables workflows to automatically search code repositories, detect the vulnerabilities, and even suggest or implement context-aware fixes [49].

Agentic AI tailors its analysis to each application's specific architecture, in contrast to standard approaches which assign vulnerabilities, the standard severity scores. This makes prioritizing and fixing security issues more precise [49]. Moreover, these AI systems reduces the time and human efforts required for the vulnerability management, minimizing the risks of both the oversight and error in manual coding processes.

Perhaps the most significant aspect of agentic AI is it's capable of automated

vulnerability remediation. Rather than relying only on developers to manually address each security issues, AI agents can generate fixes that are both safe and non-disruptive, substantially accelerating the secure code deployment. Additionally, agentic AI systems are less susceptible to hallucinations as they have been trained for specific goal oriented security tasks and operates mostly in defined contexts, which also enhances reliability [55].

Automation and Orchestration in Security Ecosystems

In addition to automating particular set of tasks, agentic AI is increasingly playing a key role in the coordination of complex, and multi-layered security ecosystems. For example, 11 specialized AI agents created by Microsoft and its partners are integrated into Microsoft's Security Copilot to assist with a range of tasks, such as vulnerability prioritization, breach reporting, and phishing triage. These agents increases the accuracy for both the junior and the experienced professionals and improve mean time to react (MTTR) by up to 30% [49].

Looking ahead, agentic AI is also set to contribute towards the international cybersecurity coordination. Platforms such as Microsoft's envisioned Cyber Eagle initiative seeks to aggregate the threat intelligence from international sources including the dark web and Interpol in order to simulate attack scenarios and cooperatively modify the defenses in real time [56]. These advancements highlight the potential of agentic AI as a framework for collaborative, cross-border cyber defense systems as well as a tool for organizational security.

2.7.2 Potential of combining LLMs with agentic architectures

The integration of LLMs with agentic AI architectures represents a promising trajectory in the development of highly autonomous and intelligent systems. Although traditional LLMs are powerful at generative and static reasoning, they often fall short in contexts requiring memory retention, dynamic interaction, multi-step planning, and real-time tool integration. Agentic architectures addresses these limitations by embedding LLMs within the modular systems that enable autonomous decision-making, reflective reasoning, and procedural adaptability.

Agentic frameworks like the Agent Laboratory, AutoGPT, and Agentic RAG use LLMs as their core reasoning engines and enhance them with features like memory persistence, tool invocation, and environmental perception. This combination enables more reliable task execution, especially in domains requiring sequential tools use and long-horizon planning. For example, the Agentic RAG paradigm introduces the planning and the reflection loops within retrieval-augmented systems, which enable LLMs to refine their outputs iteratively using external knowledge sources and also using intermediate feedback mechanisms [57].

Furthermore, agentic architectures mitigate key challenges of the standalone LLMs which includes hallucinations and static knowledge limitations, by incorporating real-time data retrieval, environmental feedback, and usage of external execution engines. The outcome is an emerging class of systems that can both generate and implement the solutions in dynamic, and real-world contexts [57].

Future developments in areas like automated research, personalized digital assistants, and autonomous scientific discovery are projected to be inspired by the interaction between LLMs and agentic designs. Future developments could include

scalable agent collaboration strategies, reinforcement learning-trained adaptive reasoning pipelines, and robust safety layers that monitor the tool use and behavioral trajectories. The combination of LLMs and agentic architectures is therefore a significant step in development of self-directed and general-purpose AI systems [57].

2.8 Security Standards and Risk Benchmarking

Security standards and scoring systems provides a common language for identifying, categorizing, and prioritizing the software vulnerabilities. These systems are essential for the proper risk communication, compliance, and automated triage. An organization can utilize a single vulnerability management policy when it standardizes vulnerability scores across all of its hardware and software platforms. This policy may be similar to a service level agreement (SLA) that specifies the timeframe for validating and resolving a specific vulnerability [58].

The most widely adopted standards includes

- **Common Vulnerabilities and Exposures (CVE):** It is maintained by MITRE, and assigns publicly known vulnerabilities with distinct identities so that they may be consistently referenced across different tools and databases. The vulnerabilities are discovered then assigned and published by the global organizations, that have partnered with the CVE Program. CVE Records are published by these partners itself to provide consistent vulnerability descriptions. These CVE Records are then ready to be utilized by cybersecurity and information technology professionals to make sure they are discussing about the same issue and to coordinate their efforts to prioritize and fix the vulnerabilities [59]. Every vulnerability entry in CVE comes with CVE-ID, which contains the year it was released and is given a unique identification number. and it follows the format “CVE-YYYY-NNNNN”. The CVE database is ba-

sically an indexing tool rather than a source of comprehensive technical or exploit information.

- **European Union Vulnerability Database (EUVD):** It is a recent initiative which is aimed at complementing the global efforts in vulnerability disclosure, particularly in the context of European cybersecurity regulations. EUVD, which was established under the EU Cybersecurity Act and is managed by ENISA, aims to establish a trusted and sovereign platform for collecting and disseminating data regarding hardware and software vulnerabilities within the EU market. Unlike CVE, which caters the global market, EUVD places more emphasis on compliance with the EU's regulatory and industrial landscape. It also encourages vendors, researchers, and national authorities to responsibly report vulnerabilities by integrating with coordinated vulnerability disclosure (CVD) methods. Although EUVD currently uses CVE identifiers for consistency, it also has the potential to introduce region-specific categories, adding a variety of viewpoints on the relevance and impact of vulnerabilities to the global ecosystem [60].

- **Common Vulnerability Scoring System (CVSS):** Developed by FIRST, CVSS provides a numerical score (0-10) along with qualitative severity ratings such as Low, Medium, High, and Critical which indicates the severity of a vulnerability based on factors like exploitability, impact, and the scope. The most recent version of CVSS is v4.0. A collection of base metrics (such as attack vector, complexity, and privileges required), temporal metrics (such as exploit code maturity and remediation level), and environmental metrics (such as a potential impact within a specific organization) are used by CVSS to determine the final scores. Automated vulnerability management tools and security advisories commonly use these scores to prioritize the remediation efforts. Despite its usefulness, CVSS has been criticized for being static, which

means that it would not accurately reflect the real-world risk associated with a vulnerability as conditions evolve [61].

- **Exploit Prediction Scoring System (EPSS):** Maintained by the FIRST and estimates the probability that a particular vulnerability will be exploited in the wild within the next 30 days. By providing a dynamic and a probabilistic method, it addresses the limitations of static scoring systems like CVSS. Exploit activity, vulnerability details (obtained from CVE database), and social signals are among the real-world data that EPSS utilizes to train its machine learning models, which is the core behind predicting the exploitability. The most recent version of EPSS is v4.0. As a result of calculating EPSS for particular vulnerability, it generates a resulting score, which is expressed in percentage, and it helps organizations in prioritizing the patching efforts based on the actual threat landscape, rather than just theoretical risk alone [62]. Instead of addressing every vulnerability with equal urgency, this data-driven approach reflects a shift toward predictive vulnerability management, where organizations can focus more on potential exploitation scenarios.

A multi-layered ecosystem for vulnerability classification and scoring can be made possible by using CVE, EUVD, CVSS, and EPSS. As a central reference point, CVE offers a standardized identifier for vulnerabilities. By incorporating the classification within a regional legal framework, EUVD supports national Computer Science Incident Response TeamCSIRTs) and critical infrastructure operators in the European Union (EU), ultimately complementing CVE.

On top of these classification systems, CVSS adds a technical severity assessment, offering a way to compare the vulnerabilities based on their intrinsic characteristics. Meanwhile, EPSS introduces a predictive, real-world dimension to vulnerability prioritization by giving an exploitability score, enabling security teams to allocate

resources more effectively.

A more thorough and strategic approach to vulnerability management is supported by the integration of these systems, where identification (CVE/EUVD), technical severity (CVSS), and exploit likelihood (EPSS) all work together to produce a risk-informed remediation plans.

These solutions enable security teams and tools to rank vulnerabilities according to precision-driven objective metrics instead of relying on the personal opinions. For agentic AI, aligning with these standards provides the anchor points for better decision logic, enabling more accurate and interpretable triage.

2.8.1 Secure-SDLC Frameworks and Standards

As discussed in section 2.1, static security analysis is the integral part of secure SDL. Likewise, a number of industry frameworks transform these high-level "secure by design" principles into concrete tasks that can be easily incorporated into each stage of the SDLC.

The practices that are most relevant to automated risk triage and false-positive reduction are highlighted below by taking an in-depth study of each framework. An AI-first solution could employ the repeatable controls, metrics, and data feeds provided by the frameworks below, which synthesize the decades of secure engineering practice, to suppress the false positives early and to elevate only risk-bearing errors.

A. NIST Secure Software Development Framework (SSDF, SP 800-218v1.1)

The NIST SSDF provides a foundational set of software security best practices. It puts these practices into four categories: producing well-secure software (PW), protecting software (PS), responding to vulnerabilities (RV), and preparing the organization (PO). Establishing clear security requirements helps in creating guidelines

that can guide AI tools to focus more on critical issues. Managing the information about third-party components, such as libraries, also helps AI to understand where risks come from [12], [63].

Key points for AI:

- Static analysis rules can be improved by using specified security requirements.
- Utilize information from the software bill of materials (SBOM) to understand the third-party risks.
- SSDF is a perfect standard for assessing AI-driven triage capabilities because it explicitly calls for automation wherever possible.

B. Microsoft Security Development Lifecycle (SDL)

Microsoft SDL is a detailed process that involves security focused steps like secure design reviews, threat modeling and testing at every stage of the development cycle. It works effectively with Microsoft tools and can supply the information that AI agents can further utilize to improve their assessment ability. Exporting threat models, for instance, could help AI in understanding the potential attack paths throughout the code [12], [64].

Key points for AI:

- Utilize threat modeling data to improve AI's code analysis.
- Utilize the findings from fuzz testing to modify the risk rankings according to the ease of exploiting a vulnerability.
- Since CI/CD "quality gates" are used to enforce the security checkpoints, SDL is a model implementation of shift-left security at hyperscale cloud velocity.

C. OWASP Software Assurance Maturity Model (SAMM)

A maturity model called OWASP SAMM aids businesses in assessing and enhancing their software security procedures. It addresses topics including operations, design, implementation, governance, and verification. AI systems can use these maturity levels to modify how strictly they filter the findings, and organizations can also track their progress [12], [65].

Key points for AI:

- Utilize maturity scores as input to enhance the reduction of the false positives.
- AI models can be trained on real issues using defect management data.

D. Building Security In Maturity Model (BSIMM)

The foundation of BSIMM is the observation of what many companies actually do in practice to improve the overall software security. It lists the common activities and how often they are done within the organisation. This can help AI in giving priority to controls that have been shown to work well in real-world scenarios. For example, if many companies automate the static analysis checks, AI can he focus more on that first [12], [66].

E. ISO/IEC 27034 – Application Security Guidelines

ISO 27034 is a set of standards which promotes the integration of security practices into SDLC while taking the business and regulatory environments into account. It assists in tailoring security measures to the importance of the application, which helps AI assess vulnerabilities based on the software's criticality [12]. ISO 27034, which emphasizes auditability and process evidence, connects SDLC operations to the overall governance of ISO 27001.

The key important points from each of the standards and frameworks that have been addressed are covered in Table 2.6, which can be further used for designing the solution.

SSDLC Standard /Framework	**Key Takeaway for Designing Agentic AI Solution**
NIST SSDF	Clear tasks and supporting documentation, such as SBOMs and specif policies, can assist AI to filter out irrelevant static analysis alerts.
Microsoft SDL	Utilize the testing data from development tools and threat models to enhance AI's ability to identify and prioritize real risks while reducing noise.
OWASP SAMM (v2.1)	Utilize KPIs and maturity scores that enable AI to modify its filtering level in flagging out the potential issues based on organisation specific security level.
BSIMM	Utilize empirical data on others security practices, to help AI focus on the most crucial findings.
ISO/IEC 27034	Provides information on the business impact so AI can assess the severity of vulnerabilities based on the actual value of the asset, and not simply on the technical specifications.

Table 2.6: Key Takeaways from Secure SDLC Standards and Frameworks

2.8.2 OWASP Top 10 and how static tools align with it

Maintained by the Open Worldwide Application Security Project (OWASP), the OWASP Top 10 is a commonly recognized list of the most critical web application security risks, which can be seen as the backbone of AppSec. It provides developers, security professionals, and organizations with a useful manual for understanding the most common vulnerabilities and with the prevention strategies. Early in the development lifecycle, static analysis tools can be very useful in identifying many of these risks, way before the deployment or release. Table 2.7 discusses the OWASP

Top 10 risks as published in 2021 [67].

Rank	Risk	Description
A01:2021	Broken Access Control	Failures in enforcing proper user permissions and roles.
A02:2021	Cryptographic Failures	Problems with a weak or a lack of encryption.
A03:2021	Injections	SQL, command, and other input injection flaws.
A04:2021	Insecure Design	Architectural defects or unsafe default behaviors.
A05:2021	Security Misconfiguration	Unpatched flaws, unnecessary features, or Insecure settings.
A06:2021	Vulnerable & Outdated Components	Using Outdated libraries or libraries with known vulnerabilities.
A07:2021	Identification & Authentication Failures	Issues in login, session management, etc.
A08:2021	Software & Data Integrity Failures	Insecure code or data pipelines that lack validation or integrity checks.
A09:2021	Security Logging & Monitoring Failures	Missing or ineffective logging and alerting mechanisms.
A10:2021	Server Side Request Forgery	Forcing server to make unintended requests.

Table 2.7: Overview of OWASP Top 10 (2021 Edition)

Static Security Testing tools perform effectively when they are in line with known vulnerability patterns and secure coding techniques. Table 2.8 shows how SAST and SCA tools typically perform across the some of the OWASP Top 10 categories. Same table 2.8 shows that SCA is crucial for risk management in third-party components, whereas SAST is more suitable for examining code-level vulnerabilities. Combining both strategies or improving them with AI agents is advantageous for several high-

impact OWASP categories, particularly A01, A04, A06, and A09.

OWASP Risk	**SAST Support**	**SCA Support**	**Remark**
A01: Broken Access Control	Partial	No	Difficult to detect without runtime or business logic context.
A02: Cryptographic Failures	Yes	No	SAST can detect weak crypto, hardcoded keys, insecure modes.
A03: Injections	Yes	No	SAST excels at finding SQL, command, LDAP, & other injections.
A04: Insecure Design	Partial	No	Mostly architectural; needs threat modeling or manual review.
A05: Security Misconfiguration	Partial	Partial	SAST can check code misconfigs; SCA can detect insecure defaults.
A06: Vulnerable & Outdated Components	No	Yes	SCA is designed to flag known CVEs in libraries & dependencies.
A07: Identification & Authentication Failures	Partial	No	SAST detects common auth flaws; struggles with session flow issues.
A08: Software & Data Integrity Failures	Partial	Partial	Some tools catch unsafe deserialization or tampered packages.
A09: Security Logging & Monitoring Failures	No	No	Mostly a runtime issue; out of scope for SAST/SCA.
A10: Server Side Request Forgery	Yes	No	SAST can detect unsafe URL calls from untrusted input.

Table 2.8: SAST and SCA supporting OWASP Categories

2.8.3 Importance of standardized scoring in prioritization

In order to prioritize vulnerabilities in a consistent and scalable manner, standardized score systems such as CVSS, EPSS, and OWASP classifications are essential. These benchmarks provide a common framework that goes beyond individual inter-

pretation in environments with hundreds or thousands of security findings.

Key advantages include:

- **Consistency Across Teams and Tools:** By using standard definitions of severity, impact, and likelihood, security engineers, developers, and automation systems can make decisions more consistently.
- **Automated Risk Triage:** Programmatically ranking issues, setting remediation SLAs, and initiating alerts or workflows can all be done with numerical and categorical ratings. For instance, vulnerabilities that have a CVSS of 7.0 or higher are often marked for immediate attention.
- **Risk Aggregation and Reporting:** Standardized scores makes it easier to aggregate data across projects and platforms, which enables dashboards and compliance reports to promptly highlight critical issues.
- **AI Interpretability:** Standardized scores acts as ground truth anchors for agentic AI systems, assisting in decision-making, output formatting, and aligning automated triage in accordance with human expectations.
- When applying the standardized scoring methodology across static security testing, SARIF should also be taken into account. By doing this, the interoperability issue will be resolved.

By integrating these scoring systems into the Agentic AI's decision logic and reasoning functions, it ensures that triage outcomes are not only accurate but also transparent, defensible, and industry-aligned.

2.9 Research Gap Analysis

The limitations of current security technologies, the current status of secure software development methods, and the evolving role of AI in security activities have all been discussed in the preceding sections. Although each of these factors enhances the software security posture individually, there are still several significant gaps, particularly regarding efficiently and accurately classifying static analysis outputs at scale.

2.9.1 Identified gaps

Significant issues with current static security analysis methods include fragmented outputs from separate programs without integrated reasoning for exploitability evaluation and consistently high false positive rates that require manual triage. Since the tools rely more on static severity rather than dynamic contextual indications, current prioritization techniques are frequently inadequate. Additionally, there is a significant gap between automated security decisions and well-established Secure-SDLC frameworks, which makes compliance and traceability more difficult, and the potential of Agentic AI with LLMs for autonomous, intelligent triage is still largely unrealized.

2.9.2 Justification for approach as proposed in this thesis

This thesis introduces a Agentic AI–based risk-triage system which provides a centralised, intelligent, and standards-compliant solution for automated security triage in an attempt to bridge the above gaps. The proposed system:

- Utilises LLMs with task-planning capability to automatically process and classify static analysis results.
- Eliminates redundancy and fragmentation through consolidation of results

from multiple scanning tools (SAST, SCA, secrets detection) into a single triage layer.

- Employs frameworks of reasoning that combine heuristics from secure development guidelines (e.g., SSDF, SAMM, BSIMM) and standardised scoring models (CVSS, EPSS).
- provides a web-based dashboard that presents risk categorisations, filtered results, developer-specific insights, and actionable fix recommendations.
- Allows human-in-the-loop validation, which allows security and development teams to review, validate, and improve the system's decisions over time.

Alongside reducing the expense of false positives, this combined approach is expected to enhance security triage's speed, accuracy, and transparency. The system surpasses traditional automation in that it enables contextual reasoning and is aligned with secure-SDLC principles, which helps develop a software security lifecycle that is more intelligent and robust.

3 Methodology

Hierarchical and multi-layered design of the proposed methodology is expected to solve the identified gaps in the field of static security analysis, including the large volume of false positives, long and time consuming triage processes, and fragmented output from tools. The system is built on an Agentic AI paradigm that inspects and ranks scan data independently by combining rule-based heuristics with LLM-based reasoning. The modular design of the architecture ensures the scalability, maintainability, and seamless integration with actual CI/CD pipelines and also makes this approach production ready.

3.1 Proposed Solution Architecture

The five primary layers in this system architecture are comprised of the Input Pipeline, Agentic AI Core, Memory and Feedback Mechanism, Output Pipeline, and Web-Based Dashboard. Altogether, these layers processes the raw, unstructured security findings and turn them into prioritised, actionable information that provides end users with clear, filtered findings and also supports them with remediation information.

3.1.1 Overview of Functional Layers

Figure 3.1 illustrates the complete system architecture, clearly outlining the interaction and connectivity among its layers and components. Each layer within this

architecture is responsible for distinct and critical functions, including:

- **The input pipeline:** It is the entry point of the system, where the security scanners, such as static analysis, dependency scanning, and secrets detection tools are called, their results are then aggregated, and the resulting data gets normalised into one internal representation. This normalisation step is necessary to enable underlying agents to reason uniformly across data sources, as different tools produce different outputs.

- **Agentic AI Core:** The underlying building block of the system is the Agentic AI Core. It is composed of a number of AI agents, each of which is designed to carry out a different task in the risk triage pipeline. Starting with the false positives detection, these agents operate in sequence and collaboration to create risk score, its explaination, and generating remediation. It enables the system to scale expert-level decision-making by combining lightweight heuristic reasoning with the flexible reasoning abilities delievered by LLMs. The accuracy and clarity of the final output are then continuously improved as each agent helps to better understand each reported issue.

- Contextual understanding is enabled by the system's **memory and feedback component**, which is often utilised along the entire pipeline. Agents takes help from and constructs upon intermediate results generated in earlier stages, instead of treating every problem in isolation. This makes it possible to mimic iterative human judgement, increase the overall consistency, and incorporate past reasoning into future decisions. Memory components stores key metadata such as classification outcomes, risk levels, and partial LLM outputs, which can be helpful to agents to deliver accurate results.

- After performing the risk triage and explainatory operations, the resulting data is then securely transmitted to a centralized backend for storage and

visualisation through the Output Pipeline. Cryptographic methods ensure the confidentiality of result data during transit, and the whole process is designed to maintain integrity and secrecy across the network boundaries. The pipeline can be employed in safe production environments as it can effortlessly integrate with the databases and web-based services.

- The final layer of the system is the Web Dashboard, which renders the result and serves it to end user. It delivers both unfiltered and filtered results, along with prioritization of filtered ones, and provides the remediation suggestions by following best security practices for developers, and provides full contextual information for each issue. With the help of statistical insight and visualisations, users easily understand the security posture of their codebase, analyse the risk trend, and asses the impact of remediation efforts over time.

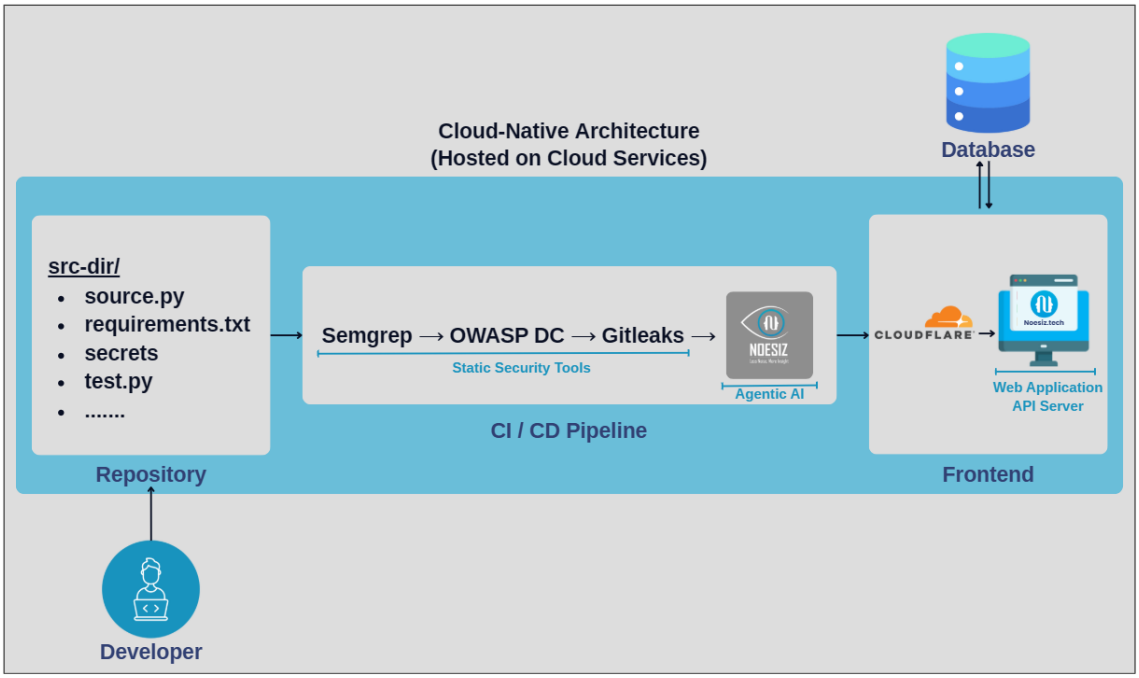

Figure 3.1: Proposed System Architecture

Extensibility is a primary factor in the design of the architecture. This architecture supports the addition of new scanning tools, additional agents, and evolving scoring

models (e.g., modified CVSS/EPSS logic), which makes it flexible to use. Moreover, this architecture offers a scalable foundation for modern secure software development pipelines through its augmentation with autonomous agents and reliance on standardised frameworks.

3.2 Agent Architecture and Reasoning Logic

To minimise false positives, categorise threats, and support the security decision-making, a coordinated set of AI agents makes up the core of the proposed system. The basis for this architecture is agentic AI, which holds up the independent components and agents to operate separately, yet toward a shared objective. Each agent performs a specific well-defined set of tasks while being a component of a broader, goal-driven system. Through this cooperative and the modular approach, the system is able to address the difficult reasoning problems while maintaining the explainability, flexibility, and transparency. Figure 3.2 shows the Agentic AI architecture that this proposed solution follows.

3.2.1 Decision Making Flow

Input normalisation is the first step, where the reports from tools are transferred using parsers, and then the system begins by normalising all the data received into a uniform internal representation. All subsequent agents used in the architecture, regardless of the role they play, will be able to reason over a uniform structured dataset due to this normalisation process.

The process of risk assessment begins with the false positive identification after normalisation, where the system identifies and filters out the potential results that are not likely to be actual security threats. This is a combination of advanced logic

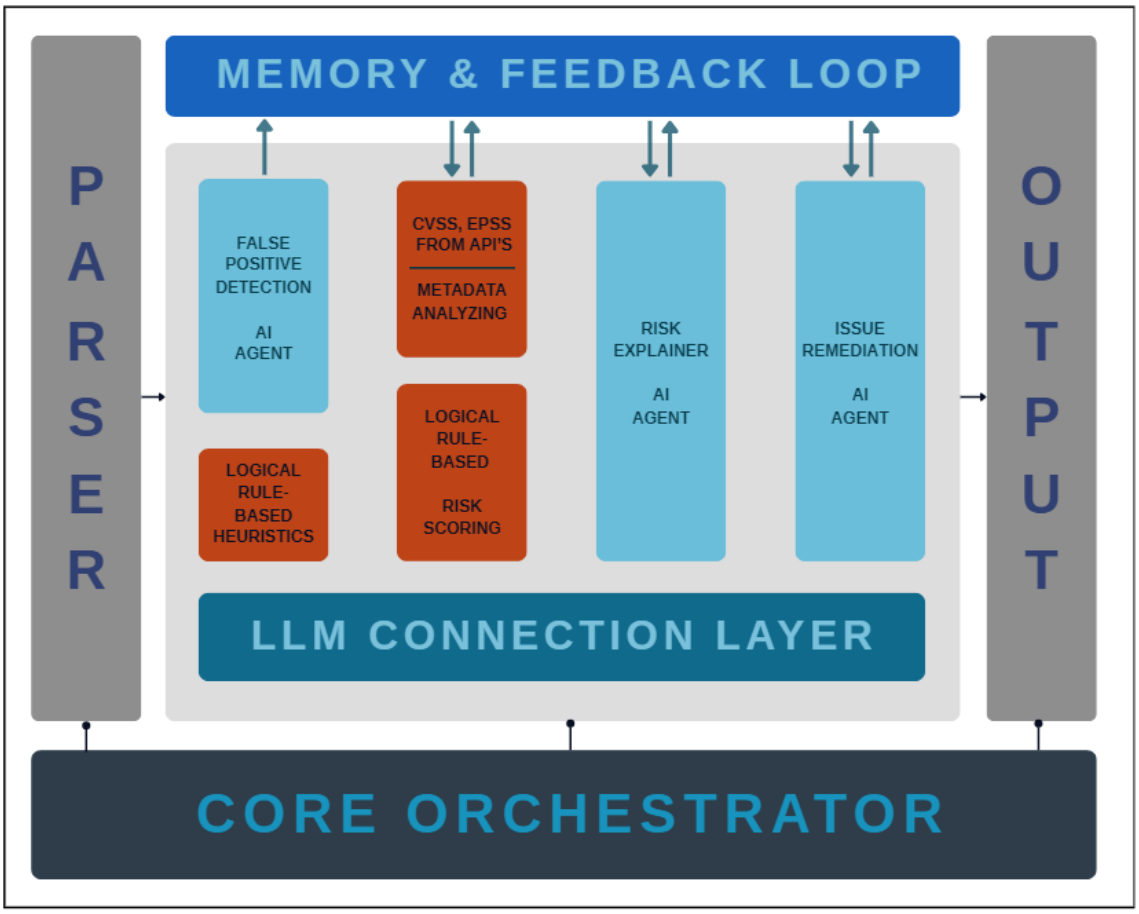

Figure 3.2: Agentic AI Architecture

informed by LLMs with some rule-based heuristics. To properly reduce the noise while maintaining the flexibility needed to process ambiguous or edge scenarios, the system uses the probabilistic AI-based logic with deterministic filters.

After filtering out the false positives, the remaining true positives are then handed over to the risk scoring agent to assess the risk. By employing widely accepted measures such as the CVSS, EPSS, false positive detection agent evaluates the severity and the exploitability of each issue. The agent uses rule-based reasoning that considers contextual factors, such as the type of vulnerability, its path in the codebase, and the sensitivity of the impacted asset, when there is no standard scoring information available. This module, unlike past agents, employs a limited quantity of LLMs and primarily dependent on logical and numerical computing since

risk scoring is founded on structured quantitative methods.

After calculating the risk score, agents are dependent on getting the explanation behind the score and also the remediation strategy to resolve the issue. Remedial guidance comes with accurate reference links from sources such as the OWASP Prevention Cheat Sheet to enhance its effectiveness and readability.

Memory and feedback loops provide a contextual foundation to the underlying agents, upon which they can start from previous evaluations and maintain consistency within the agents' decision, supporting multi-stage reasoning and maintaining accuracy. This architecture simulates human experts' decision-making process using a knowledge corpus to inform their selections.

This architecture provides a scalable and an intelligent method of automated security triage by combining the rule-based methods with AI-driven reasoning and then arranging them into a logical, memory-aware agent sequence. It enables the organisations to focus on the most critical risks.

3.3 Tool and Dataset Selection

A set of popular static security analysis tools was selected to effectively demonstrate and analyse the proposed system. This selection ensures comprehensive coverage of the three principal threat surfaces, exposed secrets, insecure dependencies, and source code flaws, that are commonly encountered in the modern systems. In addition, domain knowledge from the secure software development practices is integrated into the created datasets and rulesets.

Three tools were chosen based on their popularity, open-source nature, reliability, and ease of integration within automated pipelines, as shown in Figure 3.3 and described in Tables 2.3, 2.4, and 2.5. These tools are then connected to proposed

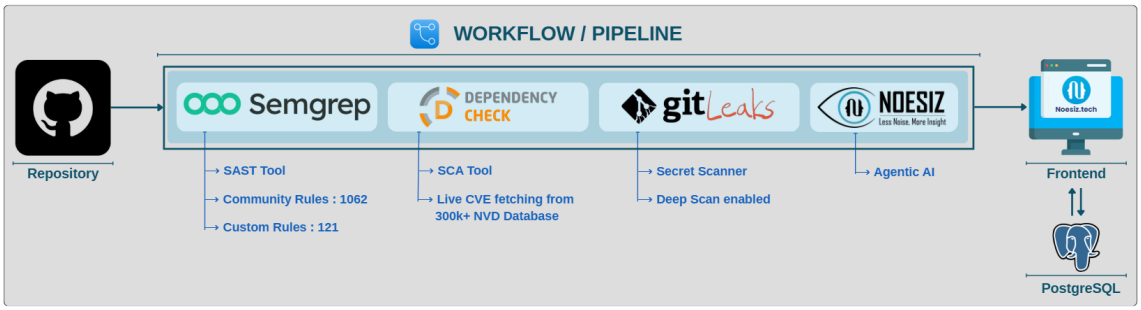

Figure 3.3: Security Tools for Pipelines

Agentic AI (named Noesiz):

1. The primary SAST tool is Semgrep Community Edition (CE). It supports multiple language with having 1,062 rules maintained by the community, and also provides the feature to create custom rules.

2. SCA utilises OWASP Dependency-Check (DC) to scan project dependencies and match them to known vulnerabilities. System is configured to fetch data directly from the National Vulnerability Database (NVD) using NVD API, which has more than 300,000 entries per scan.

3. GitLeaks is selected to discover hardcoded secrets in the codebase, such as cloud credentials, authentication tokens, and API keys. It supports both general-purpose and service-specific (e.g., AWS, Discord, etc) secret detection.

3.3.1 Alignment with Industry Standards and Frameworks

The solution incorporates external data sources to improve vulnerability benchmarking in addition to using the data produced by static analysis tools. Metrics from the CVSS and EPSS are specifically used to enhance discoveries linked to known CVEs. These ratings, which the triage agents receive via trusted APIs, enables associated agent to sort vulnerabilities by their severity and exploitability.

Well-established frameworks outlined in Section 2.8.1 act as a reference for creat-

ing custom rules and filters to ensure the consistency between the reasoning logic of the AI agents and standard secure development practices. These guides offer data for both the underlying logic employed to qualify severity based on vulnerability type and contextual code data, as well as rule content used in static analysis.

3.4 Evaluation Metrics

This research employs a collection of quantitative and qualitative measures, which are aligned with the primary objectives of the system to measure the performance of the proposed agentic AI system. These metrics are grounded in both academic evaluation standards and organisation-specific practical security engineering needs, ensuring that the results can be meaningfully interpreted by practitioners and researchers alike.

3.4.1 False Positive Reduction Rate (FPRR)

The False Positive Reduction Rate (FPRR) is an important assessment metric employed to measure how effectively the system is addressing the reported issue. FPRR quantifies the proportion of first-flagged results that are correctly eliminated by the system as not being alarming while preserving the integrity and importance of actual security threats. This measure provides a clear picture of how effectively the system can reduce noise without compromising on the detection accuracy. A high FPRR indicates how well the system reduces pointless alarms, which saves the triage time and boosts the trust in the precision and dependability of the tool's output.

3.4.2 Triage Accuracy

Triage accuracy is also an important evaluation metric that checks how accurately the system is able to label reported issue as true positives or false positives. This

evaluation is conducted by manually checking the system's AI-based labels against those generated by experienced security professionals.

3.4.3 Time Efficiency

Processing time is reported as an added measure of system effectiveness but not benchmarked against a fixed performance target. Traditional triage approaches often involve an intensive manual review process that frequently takes hours or even days. On the other hand, the proposed system is designed to operate autonomously and typically completes the overall triage process within a few minutes, which is roughly how long modern CI/CD pipelines take to run.

Nevertheless, because several outside influences like hardware configurations, CI/CD pipeline settings, LLM reaction time, and architectural variances can easily have a substantial impact on the execution time, it should be mentioned that processing time is not utilised for direct comparative purposes against other systems.

4 Implementation

The entire prototype[1] of the proposed architecture, as can be seen in figure 3.1 is designed to operate inside a Github Actions (GHA) pipeline in order to replicate a development environment that is almost ready for production. The end-to-end process involves integral components like codebase scanning, AI-based triage, and result visualisation through an online dashboard accessible to the end-user.

Moreover, the framework utilises production-level components such as cloud-hosted dashboards, API servers, secure key management, and a PostgreSQL database backend to enable scalability and maintainability. The solution is designed for cloud-native deployment instead of local run-time, which aligns more towards modern DevSecOps practices and allows easy integration into next-generation software delivery pipelines.

4.0.1 Technology Stack and Tooling

The system is built using a carefully selected stack of languages, tools, APIs, and platforms that ensure performance, maintainability, and security as mentioned in Table 4.1, 4.2, 4.3, and 4.4.

[1] All the source code and assets required for this project is stored on Github, and is available at: *https://github.com/sumitUTU/Noesiz*

Tool/Platform	**Purpose/Description**
GitHub	Used for version control and automation via GHA.
Runner.com	Hosts the API server and frontend web dashboard.
Cloudflare	Handles domain management and security (e.g., HTTPS DDoS protection).
Google Cloud Platform	Provides LLM (Gemini API), Generative Language API, and external integrations
Noesiz.tech	Registered domain pointing to the deployed application and dashboard.
Google Trust Services	Provides the SSL certificate for secure HTTPS communication.

Table 4.1: Infrastructure and Platform used for Implementation

Tool/Platform	**Purpose/Description**
Python	Core backend logic, agent implementation, and API communication.
Flask	Web framework used for serving the dashboard and APIs.
JavaScript, HTML, Tailwind CSS	For frontend visualisation.
SQL (PostgreSQL)	Database for storing triaged results.

Table 4.2: Programming Languages and Frameworks used in Architecture

Tool/Platform	**Purpose/Description**
Semgrep CE	Used as the SAST engine for detecting insecure coding patterns.
OWASP Dependency-Check	Analyses third-party libraries and flags known CVEs.
GitLeaks	Detectshardcoded secrets, API keys, and credentials in source code.

Table 4.3: Security Tools used in Architecture

Tool/Platform	**Purpose/Description**
Gemini 2.5 Flash (via Gemini API)	The LLM used to power all agent reasoning and responses.
FIRST EPSS API	For dynamic exploit prediction scoring.
NIST CVSS API	To fetch severity scores and vulnerability metadata.

Table 4.4: APIs and External Sources in use by the Agentic AI

These tools are used in combination with custom Python scripts and logic driven heuristics to enable the autonomous functioning of the system, from ingesting raw scan results to returning filtered, actionable outputs.

4.1 Input Pipeline

The system's initial step, the Input Pipeline, is tasked with gathering, categorising, and preparing raw data to the point that the agentic AI framework can further process it. It acts as the system's input layer, adding particular security scanning tools directly into the CI/CD process of GHA. Each tool generates the report in JSON format. However, the structure and parameters of these reports vary across tools. To address this inconsistency, a custom parser is employed to transform and standardize the outputs into a uniform schema.

The pipeline, established to reflect real development processes, adheres to standard CI/CD triggers, such as pull requests or merges to main branches, where static analysis is typically performed automatically. Likewise, as soon as there is a valid commit in the repository along with a push request, it will automatically trigger the GHA, and so the Agentic AI. Figure 4.1 shows the successfully triggered GHA pipeline on a push, and it runs all the jobs, and stored the report from the tools in its artifact storage.

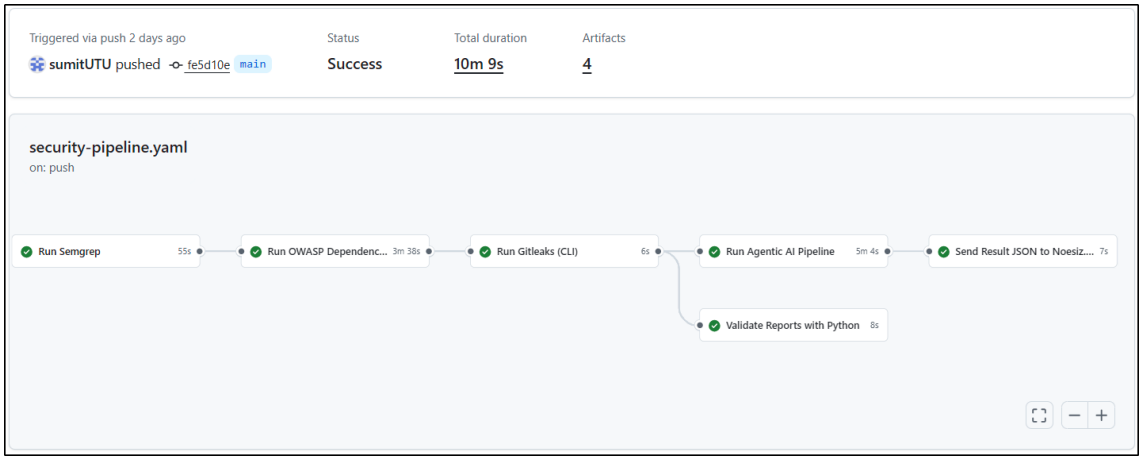

Figure 4.1: Github Actions Pipeline (triggered)

4.1.1 Folder Structure and Scanning Setup

The application code resides within a particular project directory, for example, src, where the system executes. Three security scanning tools are executed as part of the GHA pipeline workflow upon run:

- **Semgrep CE:** Employs a combination of default and custom rulesets to scan source code. GHA configuration for semgrep used in pipeline can be seen at Figure 4.2.

```yaml
jobs:
  semgrep:
    name: Run Semgrep
    runs-on: ubuntu-latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v4
      - name: Set up Python
        uses: actions/setup-python@v5
        with:
          python-version: '3.x'
      - name: Install Semgrep
        run: python3 -m pip install semgrep
      - name: Semgrep Scan Output
        run: semgrep scan --json --output=semgrep-report.json --config="tools/semgrep" --config="auto" --include=src
      - name: Upload semgrep.json
        uses: actions/upload-artifact@v4
        with:
          name: semgrep-report
          path: semgrep-report.json
```

Figure 4.2: Semgrep GHA Configuration

- **OWASP Dependency-Check:** Searches for known vulnerabilities listed under CVEs by scanning project dependencies. GHA configuration for OWASP Dc used in pipeline can be seen at Figure 4.3.

```yaml
owasp:
  name: Run OWASP Dependency-Check
  runs-on: ubuntu-latest
  needs: semgrep

  steps:
    - name: Checkout code
      uses: actions/checkout@v4
    - name: Download and extract Dependency-check
      run: |
        curl -L -o dependency-check.zip https://github.com/jeremylong/DependencyCheck/releases/download/v12.1.0/dependency-check-12.1.0-release.zip
        unzip dependency-check.zip -d dependency-check
        chmod +x dependency-check/dependency-check/bin/dependency-check.sh
    - name: Run OWASP Dependency-Check and generate JSON
      env:
        NVD_API_KEY: ${{secrets.NVD_API_KEY}}
      run: |
        ./dependency-check/dependency-check/bin/dependency-check.sh --project Noesiz  --scan "./src" --format "JSON" --enableExperimental --out .
        --nvdApiKey "$NVD_API_KEY"
    - name: Upload dependency-check-report.json
      uses: actions/upload-artifact@v4
      with:
        name: owasp-report
        path: dependency-check-report.json
```

Figure 4.3: OWASP DC GHA Configuration

- **GitLeaks:** Searches for hardcoded secrets within the codebase, including source files, env vars, and config files. GHA configuration for OWASP Dc used in pipeline can be seen at Figure 4.4.

```yaml
gitleaks:
  name: Run Gitleaks (CLI)
  runs-on: ubuntu-latest
  needs: owasp

  steps:
    - name: Checkout code
      uses: actions/checkout@v4
    - name: Install Gitleaks
      run: |
        curl -sSL https://github.com/gitleaks/gitleaks/releases/download/v8.27.2/gitleaks_8.27.2_linux_x64.tar.gz -o gitleaks.tar.gz
        tar -xzf gitleaks.tar.gz
        chmod +x gitleaks
        sudo mv gitleaks /usr/local/bin/gitleaks
    - name: Run Gitleaks and generate JSON
      run: |
        gitleaks detect --source=./src --report-format=json --report-path=gitleaks-report.json --exit-code=0
    - name: Upload gitleaks report
      uses: actions/upload-artifact@v4
      with:
        name: gitleaks-report
        path: gitleaks-report.json
```

Figure 4.4: Gitleaks GHA Configuration

Each tool generates a formatted output, typically in JSON, and it then gets stored in GHA artifact, as can be seen in figure 4.1. The downstream components of the AI triage system consume these output files for further processing once they are temporarily held in the GHA run-time environment.

4.1.2 Secrets and Configuration Management

GitHub Secrets are utilized to securely store sensitive credentials, such as the shared secret for the API server, the Gemini API key, and the NVD API key. These are never stored to disk or viewed in plaintext logs; rather, they are injected into the pipeline during execution. Additionally, the customized configurations are included in each scanner, like specific rulesets (121 custom rules) were added on top of 1062 community rules in Semgrep SAST, to check and flag insecure practices. With the NVD API key, the pipeline fetches the latest vulnerability data from the NVD database during each execution,which ensures OWASP Dependency-check to cover newly discovered threats. Also, to minimise noise and optimise efficiency, only relevant folders and files are targeted. Each tool is configured to output reports in the JSON format for streamlined processing later.

4.1.3 Normalization and Data Preparation

The system includes a normalising layer to ensure consistency since the output of every scanning tool is generated in a unique format. To align these outputs to one internal schema and avoid confusion during LLM-based analysis, some special parsing is still required, despite the fact that the selected tools allow SARIF and generate JSON reports. Parser scripts are used to perform such operations.

Subsequent to parsing each of these reports from Semgrep, OWASP Dependency-Check, and GitLeaks, the most important information is reformatted into a uniform format, with the help of parsers, and then stores it in memory and feedback loop. The Agentic AI Core accepts this normalized output, ensuring downstream reasoning and uniform triage.

4.1.4 Summary of Input Pipeline Workflow

In order to prepare and deliver the enhanced security information to the Agentic AI Core, the input pipeline performs a series of sequential actions. The following constitute this data flow:

1. **Code push:** When code is pushed to the GitHub Repository by a developer, the GHA pipeline is initiated.

2. **Tool execution:** The pipeline starts the Semgrep CE, GitLeaks, and OWASP Dependency-Check scanners, which scan the particular folders of the repository where the developer has committed changes.

3. **Report:** JSON reports are produced by each tool after a successful scan.

4. **Parsing and normalisation:** The outputs from the tools are then processed by custom parser scripts (*gitleaks_parser.py*, *owasp_dc_parser.py*, and *semgrep_parser.py*) and transformed into a standardised, consistent structure.

5. **Data handoff:** The Agentic AI Core takes in the normalised results for automated triage and reasoning.

The visual representation of the data flow can be seen in figure 4.5. The outputs from different security tools are effectively integrated and streamlined by this optimised pipeline, thereby laying the groundwork for automated and smart risk analysis.

4.2 Agentic AI Core

The central intelligent layer of the proposed system is the Agentic AI core. It processes the normalized outputs from static security scanners, applies layered reasoning on it, and then outputs a filtered and prioritized list of risks along with the reasoning

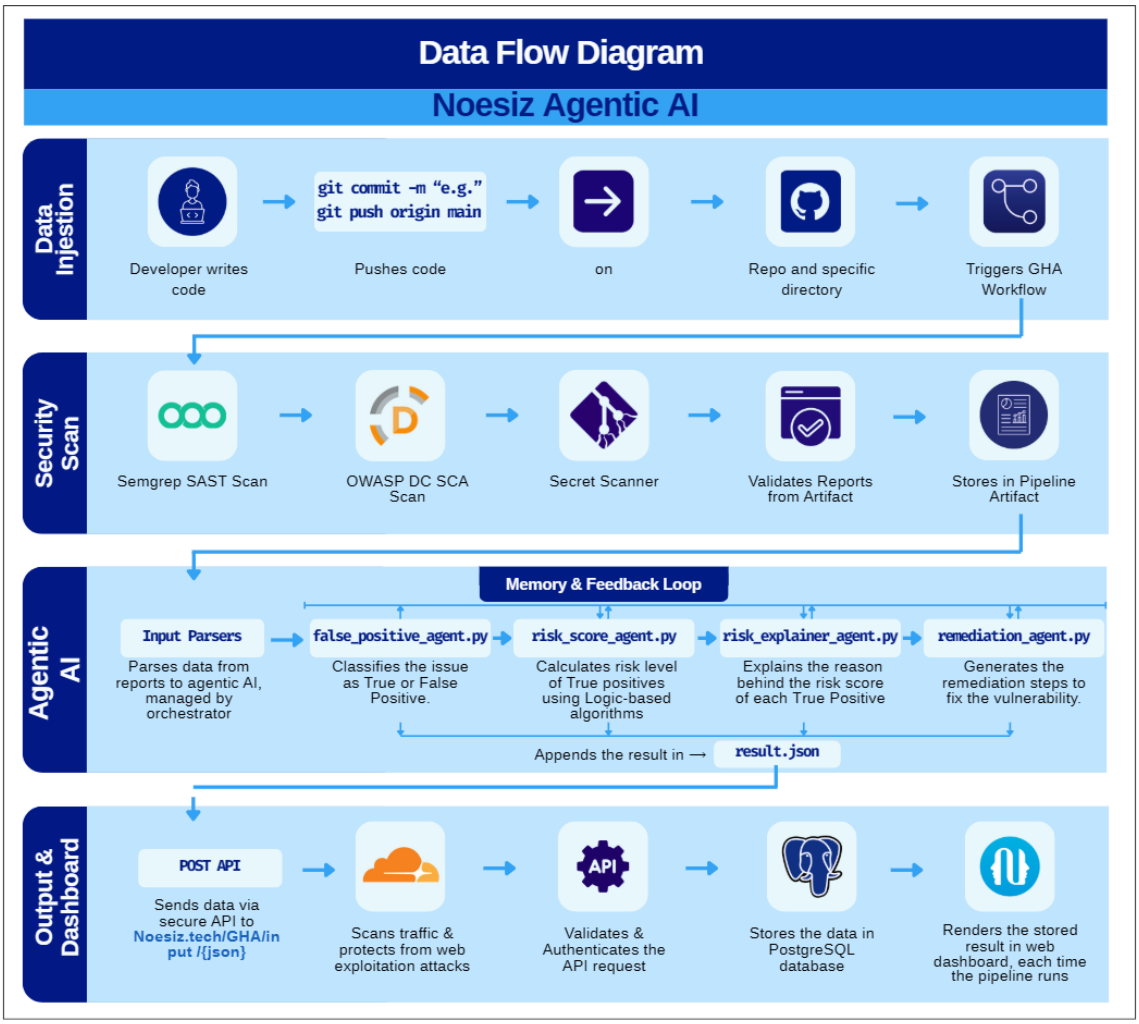

Figure 4.5: Data flow Diagram of the system

explaination. Built on the principles of agent-based architecture, this core consists of multiple autonomous agents, each one is responsible for handling one part of the triage process, while moving towards the common objective.

To improve the quality of decision-making, these agents work within a sequential, memory-aware pipeline where context is built up at each step, which is again read up and write to by all agents. The architecture acts the same way as human security engineers would, by evaluating and ranking vulnerabilities in terms of integration with heuristics, rule-based elimination, and LLM-driven reasoning.

A master orchestrator governs the core, scheduling each agent's run, related

scripts, and decision logic, as illustrated by Figure 4.6. The elements of the Agentic AI Core are the False Positive Detection Agent which detects and filters out results that are not applicable or not exploitable. Then the Risk Scoring Agent which is more focused on logical rules, and deploys CVSS, EPSS, and mathematical logic to derive the risk score. Each outcome then is supported by human-readable context evidence and reasoning furnished by the Risk Explanation Agent. Lastly, the Remediation Agent offers actionable remediation recommendations, which are guided by best security practices in secure coding. The most important component which serves all this agent is the Memory & Feedback Loop, that facilitates the iterative improvement, remembers the previous triage results, and preserves agent state.

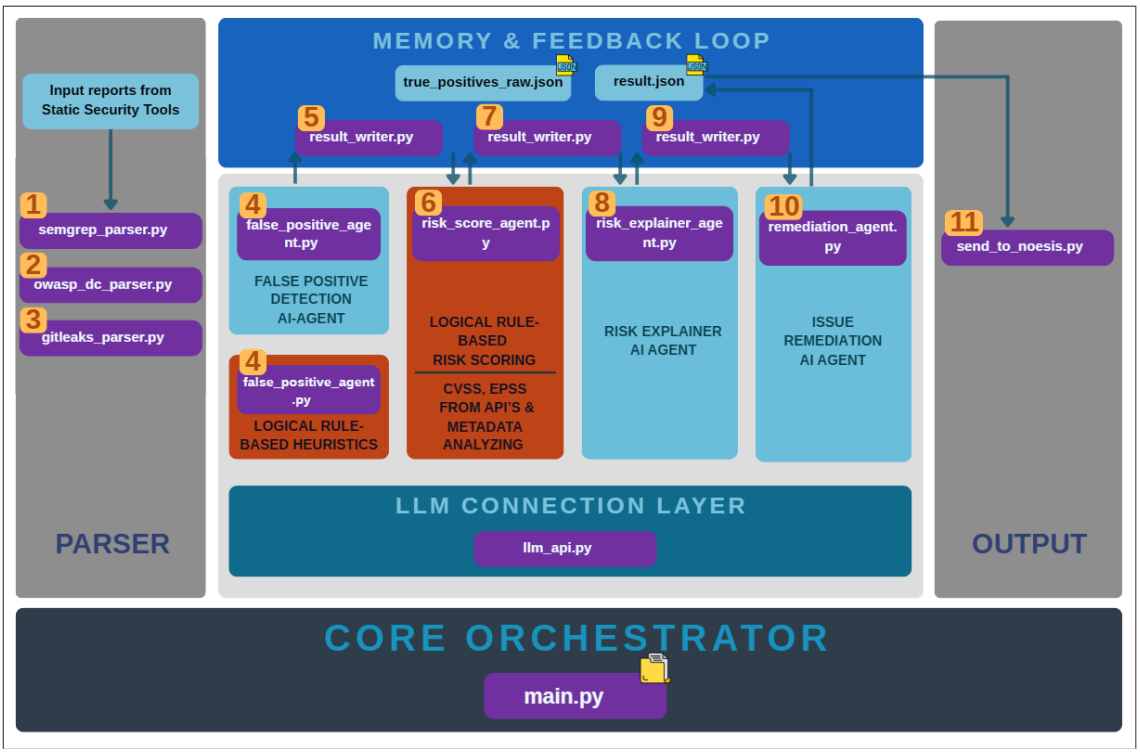

Figure 4.6: Implmented Agentic AI Architecture

The goals, implementation rationale, and interaction patterns of each of the components of the Agentic AI Core are discussed in detail in the subsequent subsections.

4.2.1 Triage Orchestrator

Inside the Agentic AI Core, the Triage Orchestrator is the principal controlling module. It is responsible for orchestrating the sequential and successful execution of each agent component, ensuring that the triage process, all together from input normalisation to the final risk output, moves in a deterministic, traceable, and systematic manner. It also maintains and manages the logs, again following the security best practices.

The orchestrator, used in *main.py*, is the primary entry point to the AI pipeline. It sets up the runtime environment, manages the ingestion of scanner outputs (through parsers), and calls each agent sequentially in the correct run order when it is called during the CI/CD process with GHA.

4.2.2 LLM Communication Layer

A specialised Python program called *llm_client_api.py* makes sure that the communication with the Gemini API is secure, reusable and dependable. Through the abstraction of API complexities, this package provides an easy-to-use and unambiguous interface for communication with LLMs. It takes care of error retries, response parsing, prompt building, authentication using securely held API keys (via GitHub Secrets), and fault-tolerant and consistent integration across the Agentic AI pipeline.

Additionally, by decoupling LLM communication from agent logic, this component is critical to maintaining modularity. The *llm_client_api.py* package consolidates the interaction with the Gemini model so that agents focus on what they want to ask, while the communication layer handles how to ask it.

4.2.3 False Positive Detection Agent

Implemented in *false_positive_agent.py*, it is a specific module created to address the problem of false positives in security scan results. Through the combination of contextual analysis powered by LLM and using tool-specific heuristics, it decides whether a given reported issue is a true positive (TP) or a false positive (FP). Its primary objective is to minimise noise without removing actual vulnerabilities, and this will enhance the overall triage process's accuracy and efficiency.

Tool-Specific Heuristics

The False Positive Detection Agent initially applies a sequence of deterministic heuristics derived from domain knowledge and known properties of every scanning tool before leveraging the LLM for context analysis. The heuristics listed below are customised for every tool:

- **Semgrep CE:** Findings labeled with "dead code" in their message or those assigned a severity level of INFO are filtered, especially if they appear in non-runtime files. Code fragments flagged in unreachable branches or legacy test files are deprioritized.
- **OWASP Dependency Check:** Dependency-related vulnerabilities that are only discussed in test scenarios are often mentioned as potential false positives since they could not be exploited.
- **Gitleaks:** Secrets appearing in test, sample, or documentation files and with entropy scores less than some cutoff (e.g. 3.5) are likely to be non-sensitive placeholders. Example: DUMMY_SECRET="123" would not appear in a test file like env/.env.example.

The *false_positive_agent.py* script already has these filtering rules integrated into it, making for efficient pre-screening before applying the LLM to more advanced

triage.

LLM-Based Contextual Judgment

Some results needs further analysis even after applying the initial heuristic. The agent employs contextual reasoning through the LLM to address issues. It forms a structured prompt using data and metadata of the issue and calls LLM through *llm_client_api.py*. The LLM is requested to respond if the problem would pose a security threat in a production environment or not. The agent interprets the outcome and derives a binary classification from the LLM response, either true positive or false positive, and why.

Output and Memory Updates

After evaluation, the agent updates two files in memory and feedback loop:

- ***true_positive_raw.json***: Contains detailed reasoning (including LLM responses) and the original finding context. This acts as shared memory for other agents to learn from.
- ***result.json***: Contains only the filtered, confirmed findings, moving forward for risk scoring and remediation.

False Positive Detection Agent plays a foundational role in the AI triage system. It ensures that the rest of the pipeline focuses on action

4.2.4 Risk Scoring Agent

The other security findings that remain after the algorithm suppresses false positives need to be prioritised based on their risk or exploitability. Prioritisation of these findings by the use of a mix of industry-standard criteria (e.g., CVSS and EPSS) and custom rule-based logic for non-CVE issues. Risk Scoring Agent, is responsible

for this operation which is deployed in the *risk_score_agent.py* file. As a result, every issue receives a final risk rating from the agent after scoring, **Critical**, **High**, **Medium**, or **Low**.

Scoring for CVE-Based Issues

For vulnerabilities identified by tools like OWASP DC which map directly to recognised CVEs, the agent appends metadata from CVSS and EPSS.These data are obtained programmatically through the NIST CVSS API for severity scores, and FIRST EPSS API for the likelihood of exploitation. The agent then applies a rule-based classification based on combined thresholds, as can be seen in Table 4.5.

CVSS Score	**EPSS Score**	**Assigned Risk Level**
>=9.0	>0.7	Critical
>=7.0	>=0.5	Medium
>=4.0	>=0.2	High
Else	Else	Low

Table 4.5: Threshold for CVE-based Risk Scoring

Scoring for Non-CVE Findings

Some scanners like Semgrep and GitLeaks, detect the issues that do not have a CVE identifier. For such cases, the Risk Scoring Agent uses some custom classification rules, that consider the type of vulnerability (e.g., hardcoded credentials, insecure cryptography, unsafe deserialization), File context (e.g., if found in production vs. test files), and Severity tags reported by the tool itself (e.g., HIGH, INFO).

4.2.5 Risk Explainer Agent

While quantitative risk scores and raw triage are necessary for automated security processes, they often do not have narrative explanations, especially for managers,

developers, and product owners. Without explanation, security results can be incorrectly understood, assigned less weight, or even ignored. This is addressed by the Risk Explainer Agent, which is scripted in *risk_explainer_agent.py* and generates comprehensible, human-readable explanations of all confirmed risks.

The agent reads all of the filtered and scored problems, and formulates a structured prompt for the Gemini LLM through the *llm_client_api.py* module based on the information the agent has gathered from the feedback loop for every problem.In response, the LLM provides a customised and simple explanation, which is then parsed by the agent and written back into both memory files. By answering the implicit "why should I care?" question that lies behind each reported issue, this agent is a significant contributor to developer enablement.

4.2.6 Remediation Agent

The final component of the agentic AI, the Remediation Agent, is scripted in *remediation agent.py* and is responsible for context-sensible recommendations for how to resolve each issue. While many security scanners provide broad recommendations (such as "delete hardcoded credentials"), they often don't contain actionable details that are appropriate to the specific scenario where the vulnerability occurs. In this case, the remediation agent can deliver accurate remediation that is contextual.

4.2.7 Shared Memory and Feedback Loop

True_positive_raw.json and *result.json* are two key files that are utilised to build a shared memory mechanism and feedback loop. The actual positive *raw.json* file is an agent-working memory bridge. Every agent contributes its output to this file once it has finished what it was tasked to do. These files collectively form the system's feedback loop. Agents effectively "learn" from one another's output by reading

from and writing to *True_positive_raw.json*, permitting a modular but coordinated process of reasoning. In agentic AI applications, this feedback also enhances the overall quality and accuracy of AI-based decisions.

4.3 Output Pipeline and Web Dashboard

The scored and filtered results (*result.json*) are transmitted to the output pipeline for storage and visualisation in front-end mode after the Agentic AI Core has completed its analysis and triage. This stage is required for developers and security teams to be able to access and use the results. Figure 4.7 shows the flow of result data returned by Agentic AI for storing and rendering output dashborad.

Figure 4.7: Output Pipeline Data Flow

There are three main purposes of the output pipeline:

1. Deliver the CI/CD pipeline output securely to the dashboard server.
2. Store those results in an orderly, queryable fashion using a backend database.
3. Present the results on a web front-end where users can receive remedial recommendations, review details, and see summary statistics.

Figure 4.8 shows the basic structure of result returned by Agentic AI, which then used by frontend server to serve it to the end-users via interactive dashboard.

```
{
  "security_tools": {
    "SemgrepCE": [
      {
        "issue": "Code_Vulnerability_ID_Line_Number",
        "details": "Issues details provided by SemGrepCE"
      },
      \\ other SemgrepCE issue
    ],
    "OWASP DC": [
      {
        "issue": "Outdated_Library_Name_Version",
        "details": "Issues details provided by OWASP DC"
      },
      \\ other OWASP DC issues
    ],
    "GitLeaks": [
      {
        "issue": "Hardcoded_API_Key_Type",
        "details": "Issues details provided by GitLeaks"
      },
      \\ other GitLeaks issues
    ]
  },
  "filtered_risks": [
    {
      "issue": "Code_Vulnerability_ID_Line_Number",
      "details": {
        "description": "Comprehensive description of the code vulnerability.",
        "associated_tool": "SemgrepCE",
        "risk_explanation": "Reason why"
      },
      "fix": {
        "steps": "Clear steps to resolve the issue",
        "reference": "Relevant Link"
      },
      "from_tool": "SemgrepCE",
      "priority": "High",
        "issue_id": "unique-code-vulnerability-id"
    },
    //like this for all other true positives
  ]
}
```

Figure 4.8: structure of result_.json file

4.3.1 Secure Result Transmission

The end *result.json* file is transmitted to the external web service on Render.com by GHA pipeline using *send_to_noesiz.py* script. The same script generates a

16-character random unique hexadecimal identifier (UNIQUE_ID) and generates a HMAC signature that signs the payload securely using the SHA-256 algorithm and a shared secret key.

The authenticity of the request is ensured by this cryptographic signature. For ensuring that only authorized agents can post to the database, the server side verifies the signature of the incoming request with the same shared key and HMAC. A 200 OK shows successful storage after the POST request is successfully validated at API server. Following the 200 Ok response from the server, GHA automatically serves the output link to the user as can be seen in Figure 4.9.

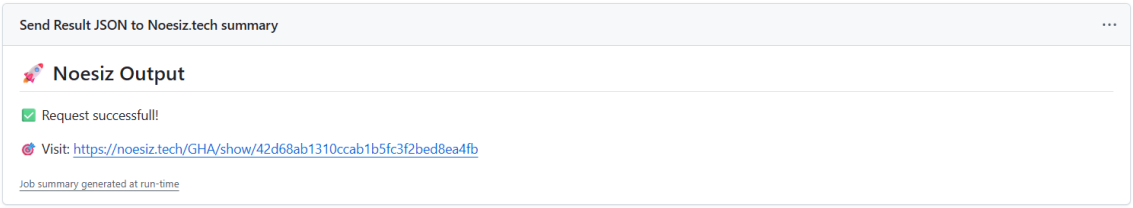

Figure 4.9: Output Redirect in GHA Summary

4.3.2 API Server and Database Integration

The backend web server is hosted with Render.com and built with the Python flask framework. Following authentication, the JSON payload is received, the relevant fields are parsed, and the data is put in a PostgreSQL database managed by the same hosting. It is also possible to load the dashboard view (/GHA/show/unique_id) and fetch the raw JSON (/GHA/api/unique_id) using the unique ID that indexes the saved data.

4.3.3 Dashboard Rendering and Visualization

HTML, Tailwind CSS, and Flask template are utilized to build the front-end dashboard as can be seen in Figure 4.10 and Figure 4.11. The system requests the saved

result, extracts the relevant fields, and shows the following when a user accesses a specific report URL (e.g., /GHA/show/5e8f3a.....):

- **Holistic issue perspective:** User can see a list of all issues generated by security tools on the left, and issues filtered by agentic AI on the right pane. Also, the details of each issue can be seen by clicking "Details", and about how to resolve the issue by clicking on the "Fix" button.
- **Data visuals:** Pie charts of raw issue distribution by tool, View of validated risks by risk level, with split of True Positives (TP) and False Positives (FP), can also be seen.

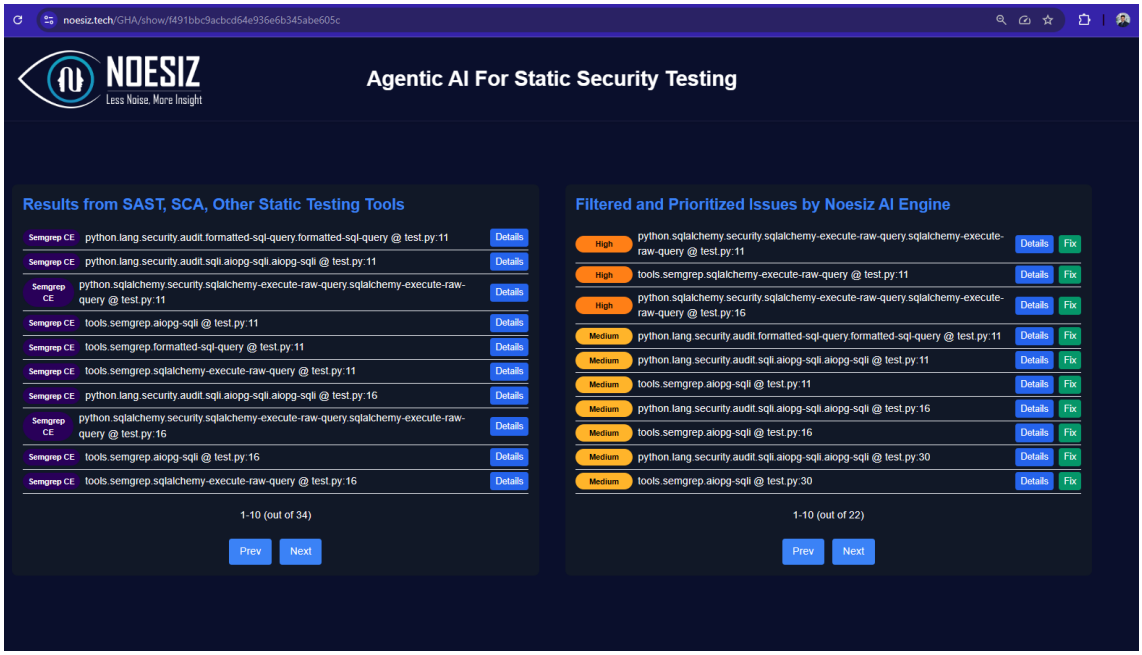

Figure 4.10: Frontend WebApp: Noesiz Dashboard Page-1

4.3.4 System Integration and Security

The dashboard resides within the noesiz.tech domain, and Cloudflare manages TLS certificates, DDoS protection, and basic firewall rules to protect from abuse and bot traffic. The pipeline structure for the entire system is as follows:

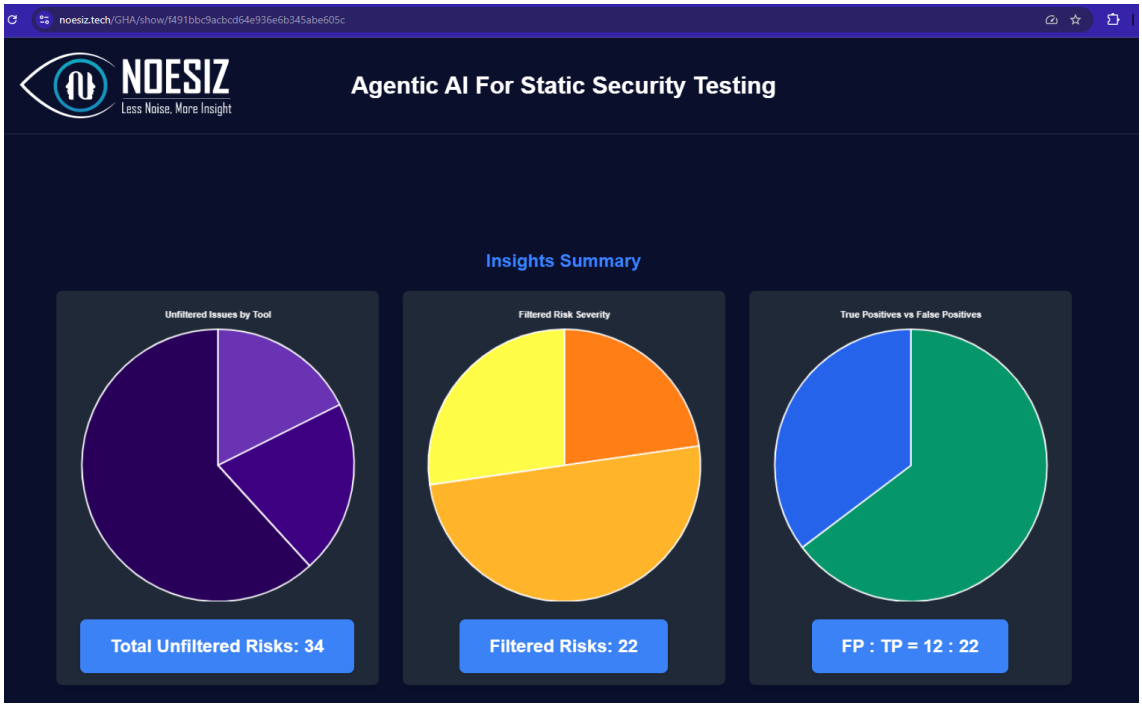

Figure 4.11: Frontend WebApp: Noesiz Dashboard Page-2

- The pipeline is initiated and the Agentic AI Core is executed within the GHA.
- Once *result.json* is ready, it is securely transmitted to the API server through *send_to_noesiz.py*.
- The API Server authenticates and stores the result in PostgreSQL. The frontend dashboard applies dynamic routes to fetch and display the triaged results live on *https://www.noesiz.tech/GHA/show/UNIQUE_ID*.

This ensures a seamless shift from code scanning to real-time, actionable, and observable results. Figure 4.12 shows, how different cloud services are configured to deliever the end result.

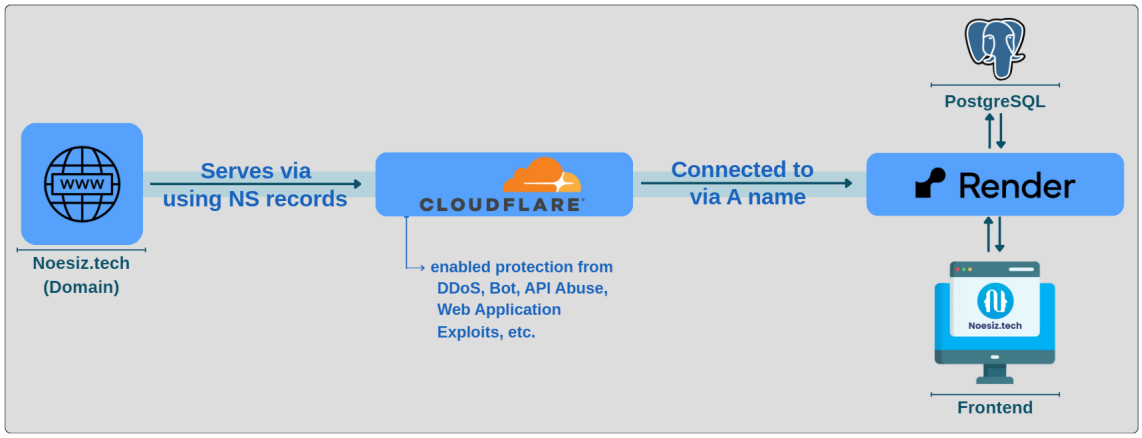

Figure 4.12: Output Pipeline Configuration

5 Results

The performance evaluation of the system is presented in this chapter. It compares manual versus agentic triage and contains information on false positive reduction and classification accuracy.

5.1 Testing Dataset and Criteria

The performance and stability of the suggested Agentic AI system for automated risk triage and false positive reduction were assessed using a set of publicly available intentionally vulnerable software repositories. This offered a varied testing environment that reflected the security flaws and the coding patterns found in the real world.

Custom scripts or part of code from total of ten vulnerable open-source application codebases were chosen from OWASP's list of vulnerable applications and GitHub. Each of these codebases included the known vulnerabilities, common misconfigurations, and instances designed to trigger the static analysis tools. These repositories provided a wide range of risk profiles, library usages, and the code quality, which served as a thorough basis for evaluating the system's flexibility.

The following metrics have been identified for each dataset:

- The total number of the raw issues reported by all of the tools.

- The number of issues that were filtered out and labelled as false positives.
- Final true positives are grouped into three risk categories: high, medium, and low.
- False Positive Reduction Rate (FPRR)

5.2 Overview of the Performance Evaluation Process

Using GHA, the pipeline was triggered ten times, once for each vulnerable dataset, to replicate the actual CI/CD scenarios. Every execution included the Scanning of the target repository for security flaws. Then running Agentic AI triage process within the pipeline, which parsed the results, removed low-confidence and false-positives findings, and applied structured reasoning for scoring and explanation. Final step in the pipeline is transferring the processed output to the Noesiz web dashboard and storing it. Lastly, a sample of AI-filtered problems is manually reviewed to evaluate the accuracy and alignment with the expected results.

The results of the ten runs are summarized in Table 5.1.

Dataset Name	**Raw Issues**	**True Positives**	**False Positives**	**FPRR**	**High**	**Medium**	**Low**
DVWA	112	74	38	33.9%	16	39	19
Juice Shop	126	112	14	11.1%	22	83	21
WebGoat	138	92	46	33.33%	23	48	21
NodeGoat	106	70	36	33.96%	14	39	17
Vulnerable Flask App	177	77	40	34.19%	13	42	22
Security Shepherd	129	88	41	31.78%	21	44	23
PythonGoat	114	75	39	34.21%	18	39	18
RailsGoat	121	79	42	34.71%	16	43	20
DSVW	104	70	34	32.69%	12	38	20
Custom Repo	137	91	46	33.58%	19	50	22
Average	**126.4**	**82.8**	**37.6**	**31.34%**	**17.4**	**46.5**	**20.3**

Table 5.1: Summary of results

5.3 False Positive Reduction Results

Reducing the number of false positives generated by static security scanners which are known to provide noisy or irrelevant findings in complicated codebases was a primary objective of this thesis. The evaluation showed that by using contextual filtering and intelligent reasoning, the suggested Agentic AI system greatly increases the signal-to-noise ratio in static security analysis.

Key Observations:

- The system's ability to filter noise without excluding significant findings was validated by the average false positive reduction rate of roughly 31.34% across all datasets.
- About 82.8% of the initial alerts have been classified as true positives for each run. These were then further categorised into High, Medium, and Low categories using scoring logic embedded in Agentic AI.
- The majority of issues fall into Medium severity (46.5 avg), followed by Low (20.3 avg) and High (17.4 avg), according to risk distributions, which show a balanced triage.
- The assessment shows that the system can manage large scanner output volumes while providing dependable, actionable security insights with a significant reduction in manual overhead.

5.4 Risk Statistics and Classification Summary

A structured classification of all validated hazards into three main categories, high, medium, and low based on a combination of static heuristics, EPSS probability, and CVSS score thresholds is part of the Agentic AI system's final output. By

prioritising the most important vulnerabilities, developers and security experts may make sure that remediation efforts are in line with the actual risk impact.

The distribution of risk severity for the remaining issues is shown in Table 5.2

Risk Level	**Average Count**	**Percentage of total TPs**
Critical	0	NA
High	17.4	21.01%
Medium	46.5	56.15%
Low	20.3	24.51%
Total	**84.2**	**100%**

Table 5.2: False-Positive Reduction by Tool

This distribution shows that most of the issues that the system retained were medium-priority risks, which usually involved using obsolete third-party dependencies, insecure APIs, or poor cryptography techniques. Even if they can't be exploited right away, these problems present serious long-term security risks if left ignored.

The three integrated tools Semgrep, OWASP Dependency-Check, and GitLeaks, were also analysed to ascertain the origin of the true positives. A representation of the segmentation is provided in Table 5.3.

Tool	**Avg. TPs/Dataset**	**Percentage of Total Tps**
Semgrep (SAST)	38.2	47.7%
OWASP DC (SCA)	21.4	26.7%
Gitleaks (Secret Scan)	20.5	25.6%

Table 5.3: Tool-wise Final True Positives and Risk Distribution

Semgrep accounted for the highest number of triaged issues, many of which were related to insecure code constructs and poor logic vulnerabilities. GitLeaks was particularly effective in identifying the credential exposures, while OWASP

Dependency-Check identified the third-party libraries with some known vulnerabilities and associated metadata.

5.5 Evaluation of Risk Prioritization Accuracy

A manual validation exercise was conducted to assess the precision of the system's automated risk classification. A random subset of the filtered issues was reviewed to evaluate their relevance, severity, and contextual accuracy, based on the established security standards and practical considerations.

The manual review confirmed that all the issues that the system had labelled as High were well-prioritized, such as sql injections. Furthermore, it was also found that the findings in the middle category of risk, such as poor cryptography or default unsafe configurations—were well-ranked and were having well-supported explanations.

The prioritized findings, when added to the benefits of faster processing and orderly explanation, tended to have a high level of consistency with manual expectation. The AI-based triage exhibited extensive explanation that enabled understanding, especially in edge situations, and showed higher consistency across concerns with the similar structures.

6 Conclusion

The sheer volume of false positives and the human effort required for sifting through static analysis results are two of the important problems of modern AppSec, that are addressed in this thesis. The demand for advanced as well as autonomous triage systems is increasing as software development pipelines increasingly rely on automated security tools following DevSecOps culture. To mitigate the key issues, the research created and implemented a new system that integrates heuristics, planning logic, shared memory, and LLMs for automatically filtering, classifying, and explaining security flaws discovered through static analysis.

6.1 Summary of Key Contributions

Several security scanner results, including those from Semgrep (SAST), OWASP Dependency-Check (SCA), and GitLeaks (secret scanner), were accommodated by the system's design, and contributes to :

- The design and implementation of an autonomous AI pipeline with self-directed decision-making abilities, such as risk assessment, false positive filtering, and LLM-based contextual reasoning, are some of the research's key achievements.
- Design of a memory-augmented feedback loop that can facilitate the coordination among different agents during the filtering, scoring, and explanation generation stages.

- The risk assessment logic built on top of best security practices (SSDF, BSIMM, SAMM) and security industry standards (CVSS, EPSS, OWASP).
- Production-ready dashboard interface design that provides detailed, graphical, and valuable insights to developers and security professionals, and is directly connected to and redirected from GHA.
- Empirical evaluation of the system showed benefits in usability compared to the traditional scanner outputs, a stable risk classification accuracy, and around 31% false positive reduction rate.

The method is more appropriate for real-world CI/CD environments because it not only reduces the human effort to triage the security issues but also enhances visibility and prioritization, and promotes DevSecOps culture.

6.2 Revisiting Research Questions

The research conducted in this thesis was guided by four key questions, each addressing a critical aspect of improving static security analysis through LLM-driven intelligent automation. The implementation of the system, its design, and empirical evaluation all determine the responses to all these questions.

6.2.1 Research Question 1 (RQ1)

How can reasoning agents and LLMs be used to differentiate between true and false positives in static analysis results?

To assess the findings from static analysis tools, the system developed an agentic AI framework that combined rule-based reasoning and LLM-driven reasoning. Each agent performed particular operations, such as cross-referencing metadata (e.g., file

paths, entropy, test directories), querying LLMs for contextual judgment, and heuristically filtering issues. This combination led to a reliable categorization of issues, demonstrating that LLMs can reliably assist in distinguishing true security results from false alarms when used alongside guardrails and well-defined context.

6.2.2 Research Question 2 (RQ2)

How can results from multiple security tools be combined and standardized to present a unified and consistent view of risks?

The solution introduced a normalized ingestion pipeline that gathered raw outputs from tools like GitLeaks, OWASP Dependency-Check, and Semgrep into a common internal format using parsers. Then, the solution relied heavily on the memory and feedback loop of the Agentic AI. The risk scores, explanations, and corrective measures for each entry were added as the agentic system processed data consistently using shared memory structures (*result.json* and *true_positives_raw.json*).

6.2.3 Research Question 3 (RQ3)

How can a web-based interface be designed to efficiently support developer workflows, offering clear visualization of triaged risks and actionable remediation insights?

The thesis included the development of a custom dashboard accessible via a secure web interface, and directly from the piplines. This interface provided detailed summaries of the filtered issues, interactive severity breakdowns, tool-wise comparisons, and one-click access to the remediation instructions. The structure and presentation of the data were intentionally designed to support clarity, traceability, and relevance in a development workflow.

6.2.4 Research Question 4 (RQ4)

To what extent can such a system reduce the overall triage time, minimize the manual effort, and improve the developer trust and engagement with security tools?

The solution reduced the number of issues requiring manual inspection by over 31% through automated reasoning, contextual filtering, and risk classification. Manual evaluations in the routine triaging were minimised when manual assessments confirmed that AI outputs were consistent with expert expectations. Comprehensive contextual explanations and organized remediation improved developer workflows, and with the help of dashboard visibility and integration with CI/CD pipelines, it improved the usability and adoption.

6.3 Limitations of the Study

While the proposed system demonstrates promising results in risk triage automation and reduction of false positives, several limitations need to be considered. The evaluation of the system was limited by a small dataset, which might have affected its wider relevance. Although the LLM-based reasoning is effective, it still has problems like overgeneralisation, and its accuracy is currently reliant on the external scanning technologies and their inherent limitations. Additionally, since the solution depends on cloud infrastructure and external APIs, it is vulnerable to problems with external services. Lastly, in the absence of end-user validation, usability and trust findings are derived from the design hypotheses rather than firsthand user inputs.

6.4 Future Work

The research presented in this thesis introduces a practical and production-grade implementation of Agentic AI for static security analysis. While the system demon-

strates effective results in reducing false positives and automating triage, there are still various directions in which this work could be taken technically and academically.

Domain-Specific Fine-Tuning of Language Models

Currently, the system generates the remediations by reasoning about scan outputs with general-purpose LLMs. Subsequent versions might require LLMs to be tuned with domain-specific security facts. Like LLMs specifically trained to solve AppSec reasoning.

Performance Optimization Using Local LLMs and Rate-Limit Handling

To interface with LLMs installed on other cloud platforms, the existing system employs remote APIs, which is higher-latency and is rate-limited to some extent. By deploying local or on-premise LLMs, organizations could achieve reduced inference times, improved control over data privacy, and can eliminate third-party usage constraints. This would also be enabled for higher-frequency agentic execution cycles, especially in CI/CD scenarios.

Optimized Handling of Vulnerability Databases (e.g. NVD)

Even through the use of the official API Key, the OWASP Dependency-Check tool needs to download approximately 300,000+ items per pipeline execution in order to get vulnerability metadata directly from the NVD. This increases processing time significantly. More scalable approach would be to establish a local NVD mirror server which is queried by the pipeline in real time and synchronizes on regular basis. It would enhance speed, reduce bandwidth usage, and enhance offline reliability.

Broader Application of Agentic AI in Cybersecurity Domains

While risk triage and static security analysis were the primary focus of this thesis, the basic agentic architecture is extendable to other areas of cybersecurity. Agent-based reasoning frameworks can be applied to such areas as AI-assisted threat modeling, incident response, cloud misconfiguration detection (along with self-repair options), and chaining of vulnerabilities.

Advancing Explainability, Trust, and User Feedback Integration

Further work is encouraged in building interpretable layers on top of the LLM decisions, especially for compliance-heavy environments. In addition to this, enabling real-time developer feedback (e.g., accepting or rejecting decisions) could be looped into the memory system to support continuous learning and adaptation of the agent.

References

[1] A. Petrosyan. "Internet and social media users in the world 2025", Accessed: Apr. 1, 2025. [Online]. Available: https://www.statista.com/statistics/617136/digital-population-worldwide.

[2] McKinsey. "What is digital transformation?", Accessed: Aug. 7, 2024.

[3] E. C. Eurostat, *Digitalisation in Europe* (Digitalisation in Europe ...). LU: Publications Office, 2025. DOI: 10.2785/3102705. [Online]. Available: https://data.europa.eu/doi/10.2785/3102705.

[4] M. Souppaya, K. Scarfone, and D. Dodson, *Secure Software Development Framework (SSDF) version 1.1:: recommendations for mitigating the risk of software vulnerabilities*. Feb. 2022. DOI: 10.6028/nist.sp.800-218. [Online]. Available: http://dx.doi.org/10.6028/NIST.SP.800-218.

[5] M. Paul, *The 7 qualities of highly secure software*. CRC Press, 2012.

[6] C. Dimastrogiovanni and N. Laranjeiro, "Towards understanding the value of false positives in static code analysis", in *2016 Seventh Latin-American Symposium on Dependable Computing (LADC)*, IEEE, 2016, pp. 119–122.

[7] M. Nadeem, B. J. Williams, and E. B. Allen, "High false positive detection of security vulnerabilities: A case study", in *Proceedings of the 50th annual ACM Southeast Conference*, 2012, pp. 359–360.

[8] Z. Guo et al., "Mitigating false positive static analysis warnings: Progress, challenges, and opportunities", *IEEE Transactions on Software Engineering*, vol. 49, no. 12, pp. 5154–5188, 2023.

[9] Z. D. Wadhams, C. Izurieta, and A. M. Reinhold, "Barriers to using static application security testing (sast) tools: A literature review", in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering Workshops*, 2024, pp. 161–166.

[10] M. Nachtigall, M. Schlichtig, and E. Bodden, "A large-scale study of usability criteria addressed by static analysis tools", New York, NY, USA: Association for Computing Machinery, 2022, ISBN: 9781450393799. DOI: 10.1145/3533767.3534374. [Online]. Available: https://doi.org/10.1145/3533767.3534374.

[11] T. Tiensuu, "Devsecops adoption: Improving visibility in application security", 2022.

[12] S. Singh, "Secure software development life cycle: Implementation challenges in small and medium enterprises (smes)", Apr. 2025. DOI: 10.22541/au.174585836.63395541/v1. [Online]. Available: http://dx.doi.org/10.22541/au.174585836.63395541/v1.

[13] S. Norberg, "Secure software development lifecycle (ssdlc)", in *Advanced ASP.NET Core 8 Security: Move Beyond ASP.NET Documentation and Learn Real Security*. Berkeley, CA: Apress, 2024, pp. 417–443. DOI: 10.1007/979-8-8688-0494-6_13. [Online]. Available: https://doi.org/10.1007/979-8-8688-0494-6_13.

[14] M. Shaikh, P. H. Ali Qureshi, M. Shaikh, Q. A. Arain, A. Zubedi, and P. Shaikh, "Security paradigms in sdlc requirement phase — a comparative analysis approach", in *2021 International Conference on Engineering and Emerg-*

ing Technologies (ICEET), 2021, pp. 1–6. DOI: 10.1109/ICEET53442.2021.9659614.

[15] A. H. A. Kamal, C. C. Y. Yen, G. J. Hui, P. S. Ling, and Fatima-tuz-Zahra, *Risk assessment, threat modeling and security testing in sdlc*, 2020. arXiv: 2012.07226 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2012.07226.

[16] M. Dawson, D. Burrell, E. Rahim, and S. Brewster, "Integrating software assurance into the software development life cycle (sdlc)", *Journal of Information Systems Technology and Planning*, vol. 3, pp. 49–53, Jan. 2010.

[17] B. Loriot, F. Madeiral, and M. Monperrus, "Styler: Learning formatting conventions to repair checkstyle violations", *Empirical Software Engineering*, vol. 27, no. 6, Aug. 2022, ISSN: 1573-7616. DOI: 10.1007/s10664-021-10107-0. [Online]. Available: http://dx.doi.org/10.1007/s10664-021-10107-0.

[18] M. Learn, *Threats - Microsoft Threat Modeling Tool - Azure*, https://learn.microsoft.com/en-us/azure/security/develop/threat-modeling-tool-threats, 25-08-2022.

[19] L. Conklin, *Threat Modeling Process | OWASP Foundation*, https://owasp.org/www-community/Threat_Modeling_Process.

[20] EC-Council, *DREAD Threat Modeling: An Introduction to Qualitative Risk Analysis — eccouncil.org*, https://www.eccouncil.org/cybersecurity-exchange/threat-intelligence/dread-threat-modeling-intro/, 2022.

[21] T. UcedaVelez and M. M. Morana, "Intro to pasta", in *Risk Centric Threat Modeling: Process for Attack Simulation and Threat Analysis*. 2015, pp. 317–342. DOI: 10.1002/9781118988374.ch6.

[22] *Triage | GitLab Docs — docs.gitlab.com*, https://docs.gitlab.com/user/application_security/triage/.

[23] J. Nagarajan, *How To Perform a Cybersecurity Risk Assessment | CrowdStrike — crowdstrike.com*, https://www.crowdstrike.com/en-us/cybersecurity-101/advisory-services/cybersecurity-risk-assessment/, 2024.

[24] P. Vishwakarma, *Combining CVSS and EPSS to prioritize vulnerability | SecOps® Solution — secopsolution.com*, https://www.secopsolution.com/blog/combining-cvss-and-epss-to-prioritize-vulnerability, 2023.

[25] A. Miles, *Triage Your Cloud Security: Risk Prioritization Methods — cyberark.com*, https://www.cyberark.com/resources/blog/triage-your-cloud-security-risk-prioritization-methods, 2024.

[26] P. M. Mell, T. Bergeron, and D. Henning, *Creating a patch and vulnerability management program:* en, 2005. DOI: https://doi.org/10.6028/NIST.SP.800-40ver2.

[27] M. Parkin, *The Hidden Costs of Poor Risk Prioritization — balbix.com*, https://www.balbix.com/blog/drowning-in-vulnerabilities-the-hidden-costs-of-poor-risk-prioritization/, 2023.

[28] *State of Software Security 2025: A New View of Maturity | Veracode — veracode.com*, https://www.veracode.com/resources/analyst-reports/state-of-software-security-2025/, 2025.

[29] A. G. Bardas et al., "Static code analysis", *Journal of Information Systems & Operations Management*, vol. 4, no. 2, pp. 99–107, 2010.

[30] D. B. Cruz, J. R. Almeida, and J. L. Oliveira, "Open source solutions for vulnerability assessment: A comparative analysis", *IEEE Access*, vol. 11, pp. 100 234–100 255, 2023. DOI: 10.1109/ACCESS.2023.3315595.

[31] G. Staff, *Octoverse: AI leads Python to top language as the number of global developers surges — github.blog*, https://github.blog/news-insights/octoverse/octoverse-2024/, 2024.

[32] Synopsys, *Synopsys 2024 Open Source Security and Risk Analysis Report*, https://static.carahsoft.com/concrete/files/1617/1597/8665/2024_Open_Source_Security_and_Risk_Analysis_Report_WRAPPED.pdf, 2024.

[33] P. Kemppainen, "Managing 3rd party software components with software bill of materials", 2023.

[34] A. Volkova, "Modern methods of automated software security analysis: From static analysis to comprehensive approach", *EUROPEAN JOURNAL OF NATURAL HISTORY*, p. 10, 2024.

[35] D. Stefanovic, D. Nikolic, D. Dakic, I. Spasojevic, and S. Ristic, "Static code analysis tools: A systematic literature review", in *Proceedings of the 31st International DAAAM Symposium 2020*. DAAAM International Vienna, 2020, pp. 0565–0573. DOI: 10.2507/31st.daaam.proceedings.078. [Online]. Available: http://dx.doi.org/10.2507/31ST.DAAAM.PROCEEDINGS.078.

[36] *SARIF Home — sarifweb.azurewebsites.net*, https://sarifweb.azurewebsites.net/.

[37] O. Security, *2022 cloud security alert fatigue report*, https://orca.security/wp-content/uploads/2022/03/Orca-2022-Cloud-Security-Alert-Fatigue-Report.pdf, 2022.

[38] A. H. Jerónimo, P. M. Moreno, J. A. V. Camacho, and G. C. Vega, "Techniques of sast tools in the early stages of secure software development: A systematic literature review", in *2024 IEEE International Conference on Engineering Veracruz (ICEV)*, 2024, pp. 1–8. DOI: 10.1109/ICEV63254.2024.10766004.

[39] H. J. Choi, H. Lee, and J.-Y. Choi, "Is a false positive really false positive?", in *2022 24th International Conference on Advanced Communication Technology (ICACT)*, 2022, pp. 145–149. DOI: 10.23919/ICACT53585.2022.9728948.

[40] H. Muthukrishnan, V. Viradia, and D. Yadav, "Unified ai and ml framework in devsecops practices, solving real-world problems", in *SoutheastCon 2025*, IEEE, Mar. 2025, pp. 1250–1257. DOI: 10.1109/southeastcon56624.2025.10971458. [Online]. Available: http://dx.doi.org/10.1109/SoutheastCon56624.2025.10971458.

[41] R. Kullberg, *Identifying and Mitigating False Positive Alerts - Panther | The Security Monitoring Platform for the Cloud*, https://panther.com/blog/identifying-and-mitigating-false-positive-alerts, 11-04-2024.

[42] G. Liargkovas, E. Panourgia, and D. Spinellis, *Quieting the static: A study of static analysis alert suppressions*, 2023. arXiv: 2311.07482 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2311.07482.

[43] I. Jaoua, O. B. Sghaier, and H. Sahraoui, *Combining large language models with static analyzers for code review generation*, 2025. arXiv: 2502.06633 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2502.06633.

[44] P. Li et al., *Automated static vulnerability detection via a holistic neuro-symbolic approach*, 2025. arXiv: 2504.16057 [cs.CR]. [Online]. Available: https://arxiv.org/abs/2504.16057.

[45] B. Aloraini, M. Nagappan, D. M. German, S. Hayashi, and Y. Higo, "An empirical study of security warnings from static application security testing tools", *Journal of Systems and Software*, vol. 158, p. 110 427, Dec. 2019, ISSN: 0164-1212. DOI: 10.1016/j.jss.2019.110427. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2019.110427.

[46] P. Pavan, "Adaptive application security testing with ai automation", *International Journal of AI, BigData, Computational and Management Studies*, vol. 4, no. 1, pp. 55–63, 2023.

[47] M. F. Ansari, B. Dash, P. Sharma, and N. Yathiraju, "The impact and limitations of artificial intelligence in cybersecurity: A literature review", *IJARCCE*, vol. 11, no. 9, 2022, ISSN: 2278-1021. DOI: 10.17148/ijarcce.2022.11912. [Online]. Available: http://dx.doi.org/10.17148/IJARCCE.2022.11912.

[48] M. Keltek, R. Hu, M. F. Sani, and Z. Li, *Boosting cybersecurity vulnerability scanning based on llm-supported static application security testing*, 2024. arXiv: 2409.15735 [cs.CR]. [Online]. Available: https://arxiv.org/abs/2409.15735.

[49] N. Kshetri, "Transforming cybersecurity with agentic ai to combat emerging cyber threats", *Telecommunications Policy*, vol. 49, no. 6, p. 102 976, 2025, ISSN: 0308-5961. DOI: https://doi.org/10.1016/j.telpol.2025.102976. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0308596125000734.

[50] *Agentic AI (and AI Agents) — cyberark.com*, https://www.cyberark.com/what-is/agentic-ai-and-ai-agents/.

[51] S. Nawaz, *Agentic AI vs AI Agents: 9 Key Differences — ampcome.com*, https://www.ampcome.com/post/agentic-ai-vs-ai-agents-a-detailed-comparison, 12,12,2024.

[52] RedHat, *What is agentic AI?*, https://www.redhat.com/en/topics/ai/what-is-agentic-ai, 6-11-2024.

[53] L. Columbus, *Cybersecurity at AI speed: How agentic AI is supercharging SOC teams in 2025 — venturebeat.com*, https://venturebeat.com/ai/cybersecurity-at-ai-speed-agentic-ai-supercharging-soc/, 24-02-2025.

[54] E. Lisowski, *AI Agents vs Agentic AI: What's the Difference and Why Does It Matter?*, https://medium.com/@elisowski/ai-agents-vs-agentic-

ai-whats-the-difference-and-why-does-it-matter-03159ee8c2b4, 29-12-2024.

[55] M. Chiodi, *Part 1 - Agentic AI in Cybersecurity: Closing the Last Mile in Identity Security.* https://www.cerby.com/resources/blog/agentic-ai-in-cybersecurity, 5-02-2025.

[56] C. T. Brayda, *Agentic AI Vs. AI Agents: Shaping The Future Of Cybersecurity — forbes.com*, https://www.forbes.com/councils/forbestechcouncil/2025/04/14/agentic-ai-vs-ai-agents-shaping-the-future-of-cybersecurity/, 14-04-2025.

[57] M. A. Ferrag, N. Tihanyi, and M. Debbah, *From llm reasoning to autonomous ai agents: A comprehensive review*, 2025. arXiv: 2504.19678 [cs.AI]. [Online]. Available: https://arxiv.org/abs/2504.19678.

[58] P. Mell, K. Scarfone, S. Romanosky, et al., "A complete guide to the common vulnerability scoring system version 2.0", in *Published by FIRST-forum of incident response and security teams*, vol. 1, 2007, p. 23.

[59] cve.org, *About the cve program*, https://www.cve.org/About/Overview.

[60] EUVD, *About European Union Vulnerability Database (EUVD)*, https://euvd.enisa.europa.eu/about.

[61] M. Albab, *Severity vs risk : The limitations of cvss*, Jan. 2025. [Online]. Available: http://essay.utwente.nl/106247/.

[62] first.org, *Exploit Prediction Scoring System (EPSS)*, https://www.first.org/epss/.

[63] M. Souppaya, K. Scarfone, and D. Dodson, *Secure Software Development Framework (SSDF) version 1.1 -recommendations for mitigating the risk of software vulnerabilities.* Feb. 2022. DOI: 10.6028/nist.sp.800-218. [Online]. Available: http://dx.doi.org/10.6028/NIST.SP.800-218.

[64] M. Learn, *Microsoft Security Development Lifecycle (SDL) - Microsoft Service Assurance*, https://learn.microsoft.com/en-us/compliance/assurance/assurance-microsoft-security-development-lifecycle, 23-05-2024.

[65] OWASP, *OWASP software assurance maturity model (SAMM)*, https://owaspsamm.org/model/.

[66] BlackDuck, *What Is the BSIMM and How Does It Work?*, https://www.blackduck.com/glossary/what-is-bsimm.html.

[67] owasp.org, *OWASP Top 10:2021*, https://owasp.org/Top10/, 2021.