Support of New Dependency Ecosystems in SCA Tool

BACHELOR THESIS

Louis Dümler

Submitted on August 14, 2025

Friedrich-Alexander-Universität Erlangen-Nürnberg
Faculty of Engineering, Department of Computer Science
Professorship for Open Source Software

Supervisor:
M. Sc. Martin Wagner
Prof. Dr. Dirk Rihele, M.B.A.

Declaration of Originality

I confirm that I have written this thesis unaided and without using sources other than those listed and that this thesis has never been submitted to another examination authority and accepted as part of an examination achievement, neither in this form nor in a similar form. All content that was taken from a third party either verbatim or in substance has been acknowledged as such.

Erlangen, August 14, 2025

License

Erlangen, August 14, 2025

Abstract

Modern software development heavily relies on third-party dependencies across diverse ecosystems, presenting significant challenges in maintaining security, legal compliance, and transparency. Software composition analysis (SCA) tools have become essential in addressing these concerns, but many struggle to support the growing number of programming languages and package managers. This thesis focuses on extending an existing software named *SCA Tool* to support additional dependency ecosystems, particularly Java (Maven/Gradle) and Python (pip, Poetry, Pipenv, uv). A modular architecture was implemented to enable ecosystem-specific analyzers and scanning capabilities. The system integrates established tools such as OSS Review Toolkit (ORT) and ScanCode while compensating for their limitations through custom extraction logic. Evaluation was conducted using a repository of representative multi-language projects to assess detection accuracy and performance. The results demonstrate that hybrid analysis strategies significantly improve dependency detection across inconsistent ecosystems. This work contributes to the field of SCA by providing practical insights into overcoming technical obstacles in multi-ecosystem software environments.

Contents

1 Introduction 1
1.1 Fundamentals 2
1.1.1 Software Composition Analysis (SCA) 2
1.1.2 License Governance 3
1.1.3 License Compliance 3
1.2 Motivation 3
1.2.1 License Diversity in Dependencies 3
1.2.2 Secure Supply Chain 4
1.2.3 Challenges of Manual SCA 4
1.3 Objectives and Scope 5
1.3.1 Primary Research Questions 5
1.3.2 Intended Contributions 5
1.3.3 Boundaries of Investigation 6
2 Literature Review 7
2.1 Foundational Technologies for SCA 7
2.1.1 Package Managers and Build Systems 7
2.1.2 Software Bill of Materials (SBOM) 9
2.1.3 Identification of Software using PURLs 10
2.2 State of the Art in SCA Tools 11
2.3 Multi-Ecosystem Analysis Tools 13
2.3.1 OSS-Review-Toolkit (ORT) 13
2.3.2 ScanCode 14
3 Requirements 16
3.1 Functional Requirements 16
3.2 Non-Functional Requirements 17
4 Architecture 18
4.1 Component-wise Construction of SCA Tool 18
4.2 Analyzer 19
4.2.1 Language-Specific Support Components 21
4.3 Scanner 24
4.3.1 Downloader 25
4.3.1.1 Download Strategy Resolution 26
5 Design and Implementation 28
5.1 Analyzer 28
5.1.1 General Behaviour 28
5.1.2 ORT Integration 30
5.1.3 Language-Specific Configuration & Behavior 30
5.1.4 Java 31
5.1.5 Javascript 34
5.1.6 Python 34
5.2 Scanner 42
5.2.1 Downloader 43

5.2.2 Tiered Quality Downloading of Source Code 43
5.2.2.1 Matching Algorithms for Packaged Dependencies 45
5.2.2.2 VCS Downloaders 46
5.2.3 ScanCode Integration 49
6 Evaluation 50
6.1 Functional Requirements 50
6.2 Non-Functional Requirements 52
7 Conclusion 54
7.1 Learnings 54
7.2 Limitations 54
7.3 Outlook on Future Developments 55
References 57

List of Figures

Figure 1 Service Communication Flow of SCA Tool 18
Figure 2 Analyzer Task Processing Flow 20
Figure 3 Language-Specific Support Flow Overview 22
Figure 4 Scanner Service Processing Flow Overview 24
Figure 5 Downloader Subsystem Class Architecture 26
Figure 6 Pseudocode for Java version download and installation script 33

List of Tables

Table 1 Analysis Method Comparison by Programming Language 23
Table 2 Table of Supported Python Package Managers 35
Table 3 Download strategies and their quality scores 44

List of Listings

Listing 1 Core analysis logic in `AnalyzerService.kt` class 29
Listing 2 Multi-extractor aggregation logic in `ExtractionService.kt` class 29
Listing 3 `ort.yaml` configuration for disabled package managers 30
Listing 4 Java-specific configuration for ORT's Gradle and Maven package managers 31
Listing 5 Conditional volume mounting in `DockeredOrtRunner.kt` 33
Listing 6 PoetryExtractor `extract()` implementation 36
Listing 7 PoetryExtractor lock file parsing 36
Listing 8 PipExtractor `runPipCommand()` implementation 37
Listing 9 `ExtractionResult.kt` data class 38
Listing 10 PoetryExtractor `parseLockFile()` implementation with Python version detection 39
Listing 11 PipExtractor actual `extract()` interface 39
Listing 12 PipExtractor `runPipCommand()` with Python version handling 40
Listing 13 Example of executed pip install command with Python version 40
Listing 14 Example of `scatool.yaml` configuration for Python version 40
Listing 15 License detection using PyPI metadata 41
Listing 16 Matching algorithm implementation in `AbstractVcsDownloader.java` 45
Listing 17 DownloaderService with registered VCS downloaders 47
Listing 18 GitHubDownloader implementation 48

Acronyms

SCA	Software Composition Analysis
DevSecOps	Development, Security, and Operations
SBOM	Software Bill of Materials
PURL	Package URL
URL	Uniform Resource Locator
ORT	OSS Review Toolkit
JDK	Java Development Kit
VCS	Version Control System
API	Application Programming Interface
CLI	Command Line Interface
CI	Continuous Integration
CD	Continuous Deployment

1 Introduction

Modern software development has fundamentally transformed from writing code from scratch to assembling complex applications from existing components. Today's applications are intricate mosaics of open-source libraries, commercial components, containers, and Application Programming Interfaces (APIs), with upwards of 94% of code in a typical codebase originating outside the organization. This shift toward component-based development has accelerated innovation but introduced new challenges that traditional development practices struggle to address (GitHub, 2024).

Each external dependency brings a cascade of legal, security, and operational obligations that must be continuously monitored and managed. A single overlooked vulnerability in a widely-used library can expose thousands of applications to attack, as demonstrated by incidents like Log4Shell in 2021, which affected hundreds of millions of devices with over 30% of instances remaining vulnerable one month after disclosure. Similarly, license incompatibilities can force costly code rewrites or legal disputes that eclipse the benefits of using open-source components (Cycode, 2024; GitHub, 2024; *Infographic*, 2022).

Software Composition Analysis (SCA) has emerged as an essential discipline to address these challenges through automated discovery, inventory, and risk assessment of third-party components. SCA tools have become critical infrastructure in modern Development, Security, and Operations (DevSecOps) pipelines, enabling organizations to maintain security, ensure license compliance, and generate Software Bill of Materialss (SBOMs) that regulators increasingly mandate (Gemmiti, 2020; *SBOM as a Legislative Requirement*, n.d.).

Among these essential SCA tools, "*SCA Tool*" exemplifies this category of software platforms as an open-source SCA solution developed at the Professorship for Open-Source Software at Friedrich-Alexander University Erlangen-Nürnberg designed to make working with open-source software "safe and easy." The platform provides comprehensive capabilities for SBOM management, open-source governance, license compliance, and vulnerability management. By automating the identification and evaluation of third-party code usage, *SCA Tool* enables development teams to detect potential security risks and ensure license compliance throughout the software development lifecycle (*SCA Tool*, n.d.).

However, the explosive growth of programming ecosystems presents a significant challenge for SCA tools. Modern technology stacks blend diverse dependency management systems — Java's Maven and Gradle, JavaScript's npm and Yarn, Python's pip and Poetry, Go modules, Rust's Cargo, and many others. Each ecosystem operates with distinct package formats, metadata structures, and dependency resolution mechanisms. Supporting this polyglot reality requires SCA tools to continuously adapt and extend their capabilities to new dependency ecosystems.

This thesis addresses the challenge of extending *SCA Tool* support to new dependency ecosystems, focusing on the technical and architectural considerations required to maintain accuracy and performance across diverse package management systems. Through analysis of existing SCA approaches and implementation of ecosystem-specific detection mechanisms, this work contributes to the broader goal of comprehensive dependency visibility in modern software development.

Following this introduction, chapter 2 presents a literature review of foundational technologies including package managers, SBOMs, and Package URLs (PURLs), while examining the current state of SCA tools and multi-ecosystem analysis approaches. Chapter 3 establishes the functional and non-functional requirements for extending *SCA Tool's* ecosystem support. Chapter 4 details the architectural framework that enables modular dependency analysis across diverse programming languages, focusing on the `Analyzer` and `Scanner` service components. Chapter 5 presents the detailed design and implementation of ecosystem-specific detection mechanisms, particularly for Java and Python environments. Chapter 6 evaluates the implemented solutions through testing across representative dependency scenarios, assessing both technical performance and detection accuracy. Finally, chapter 7 synthesizes the findings and identifies directions for future research in *SCA Tool* extensibility.

1.1 Fundamentals

To understand how organizations can effectively manage these complex dependency landscapes, it is essential to examine the fundamental approaches and technologies that enable comprehensive software composition visibility: SCA for automated dependency discovery and risk assessment, governance frameworks for establishing policies and organizational structures, and compliance mechanisms for ensuring adherence to legal and regulatory requirements.

1.1.1 Software Composition Analysis (SCA)

SCA represents the automated discovery, inventory, and risk assessment of third-party components within software applications. SCA tools emerged to replace unscalable manual processes like spreadsheet tracking and one-off audits, now forming the backbone of modern DevSecOps pipelines (Gemmiti, 2020; GitHub, 2024).

The core functionality of SCA encompasses three primary detection modes: manifest scanning, which analyzes package manager files like `package.json` or `pom.xml`; binary analysis, which identifies components through file signatures and metadata; and container scanning, which inventories dependencies within containerized applications. These approaches work together to provide visibility into software composition across different stages of the development lifecycle (GitHub, 2024).

1.1.2 License Governance

Governance in software development establishes the structures, processes, and policies that direct and control software development to yield business value and mitigate risk. Without proper governance, organizations cannot articulate requirements, manage projects effectively, or deal with failure consequences (Finkelstein, n.d.).

For SCA tools, governance is critical because it defines policies for what components are acceptable in software projects. Governance frameworks establish automated policies that block problematic dependencies, enforce license compliance rules, and manage security vulnerability responses across different programming ecosystems. Without governance, organizations have no consistent way to control which third-party components developers can use, leading to unmanaged security and legal risks in their software supply chain (*Open Source License Compliance*, n.d.).

1.1.3 License Compliance

Compliance represents the systematic adherence to established rules, standards, regulations, and legal requirements that govern an organization's operations and outputs. It ensures that products, processes, and practices meet both internal policies and external regulatory obligations, providing accountability and risk mitigation across business activities (*ISO 37301:2021(En), Compliance Management Systems — Requirements with Guidance for Use*, n.d.).

Within software development, compliance manifests most prominently through license management, where organizations must navigate diverse license families with varying obligations. Permissive licenses like MIT and Apache 2.0 require minimal compliance efforts, typically only attribution and notice retention. Weak copyleft licenses such as MPL-2.0 and LGPL-3.0 require sharing modifications of the library only, while strong copyleft licenses like GPL-3.0 and AGPL-3.0 demand distributing entire derivatives under the same license terms. The challenge intensifies in polyglot environments where different programming languages and package managers exhibit inconsistent approaches to handling licensing information with varying levels of license declaration quality (Harnik, 2023; *Licenses*, n.d.; *Thirdparty Licenses Examples – License Maven Plugin*, n.d.; *When License Field in Package.json Doesn't Match LICENSE*, n.d.; Xu et al., 2025).

1.2 Motivation

The complexity of modern polyglot software stacks creates three primary challenges that necessitate the drive towards more sophisticated SCA solutions: license diversity in dependencies across heterogeneous package managers, secure supply chain management in the face of increasing vulnerability threats, and the limitations of manual SCA approaches in modern development environments.

1.2.1 License Diversity in Dependencies

The heterogeneous nature of open-source licensing creates significant compliance challenges for organizations utilizing multiple programming ecosystems. Each package manager handles license information differently — Maven provides structured POM

metadata, npm uses varying license field formats, and Python packages often lack comprehensive licensing details entirely. This inconsistency forces organizations to manually investigate licensing terms, creating bottlenecks in development workflows and increasing legal risk exposure (Pfeiffer, 2022; *When License Field in Package.json Doesn't Match LICENSE*, n.d.; Xu et al., 2025).

Furthermore, the distinction between open-source, proprietary, and non-commercial licenses adds complexity to automated compliance assessment. A single application might incorporate MIT-licensed utilities, LGPL-licensed libraries requiring source disclosure, and proprietary components with audit clauses, each demanding different compliance workflows. Without comprehensive SCA coverage across all dependency ecosystems, organizations cannot confidently answer the fundamental question: "Can I legally use this software in my commercial product?"

1.2.2 Secure Supply Chain

Recent high-profile supply chain attacks have demonstrated the critical importance of dependency security monitoring. The SolarWinds attack in 2020 compromised 18,000 downstream organizations through build-system malware injection, while Log4Shell in 2021 exposed millions of systems to remote code execution vulnerabilities. These incidents highlight how a single compromised dependency can cascade through entire software stacks (*Infographic*, 2022; *SolarWinds Supply Chain Attack*, n.d.).

Traditional security approaches focus on application-level vulnerabilities but often overlook the transitive dependencies that comprise the majority of modern codebases. SCA tools address this gap by correlating SBOMs with real-time vulnerability feeds from sources like the National Vulnerability Database[1], OSV[2], and commercial security databases. However, the effectiveness of this approach depends critically on comprehensive dependency discovery across all package managers and ecosystems in use.

1.2.3 Challenges of Manual SCA

Manual dependency management approaches prove inadequate for modern software development demands. Organizations face significant cost disadvantages when relying on manual license compliance processes versus automated solutions. Modern package ecosystems create intricate dependency webs where single components trigger cascading chains of transitive dependencies, while continuous vulnerability discoveries across thousands of components strain manual review capabilities (Wang, 2018).

The polyglot nature of modern applications exacerbates these challenges. A typical microservices architecture might incorporate Java backend services, React frontend applications, Python data processing scripts, and Go utilities, each with distinct dependency management approaches. Manual tracking across these ecosystems becomes practically impossible, leading to incomplete visibility and elevated risk exposure across legal, security, and operational dimensions (Ramaswamy, 2022).

[1] https://nvd.nist.gov/

[2] https://osv.dev/

1.3 Objectives and Scope

Having established the fundamental challenges facing modern SCA, this research addresses these limitations through a systematic investigation of *SCA Tool's* extensibility requirements and architectural considerations.

1.3.1 Primary Research Questions

This thesis investigates three fundamental research questions regarding *SCA Tool* extensibility:

- How can SCA tools be architecturally designed to effectively support new dependency ecosystems while maintaining detection accuracy?
- What are the key technical challenges in implementing ecosystem-specific dependency detection mechanisms, and how do different package manager structures impact detection reliability?
- What evaluation methodologies can effectively assess *SCA Tool* performance across diverse dependency ecosystems with varying metadata quality and format consistency?

1.3.2 Intended Contributions

This thesis contributes to the SCA domain through three key deliverables:

- **Technical Architecture**: Extension of *SCA Tool's* existing architecture to support additional dependency ecosystems through modular detection components. The work demonstrates how different package manager parsing strategies can be integrated within the existing framework while preserving ecosystem-specific optimization opportunities.
- **Implementation Analysis**: Comparative evaluation of dependency detection approaches across Java (Maven/Gradle), JavaScript (npm/Yarn), and Python (pip/Poetry) ecosystems, highlighting the varying levels of metadata reliability and detection accuracy achievable in each environment.
- **Evaluation Framework**: Creation of a comprehensive test repository containing representative dependency scenarios across multiple ecosystems, enabling systematic assessment of *SCA Tool* performance and identification of ecosystem-specific limitations.

1.3.3 Boundaries of Investigation

This research focuses specifically on dependency detection and inventory capabilities within *SCA Tool*, with defined scope limitations:

- **Ecosystem Coverage**: Primary investigation concentrates on Java and Python ecosystems due to their prevalence in enterprise development and varying metadata quality characteristics. Java benefits from mature Maven repository infrastructure and structured dependency metadata, while Python represents the challenging case with inconsistent package manager implementations and limited metadata quality standardization.
- **Technical Scope**: The research addresses dependency discovery and matching algorithms but does not encompass vulnerability correlation, license analysis, or compliance workflow automation. These downstream SCA capabilities depend on accurate dependency identification, making detection accuracy the primary focus area (shown in section 5.2.2.1).
- **Collaborative Development Context**: This research occurs within an active development environment where multiple contributors work on *SCA Tool* simultaneously. Initial implementations and findings frequently led to collaborative improvements, with other team members enhancing and modifying the original code contributions. Testing was occasionally limited by system bugs and integration issues inherent to ongoing development cycles. Due to these constraints and system instabilities, evaluation often focused on individual components and subsystems rather than complete end-to-end workflows, enabling targeted validation despite broader system limitations.
- **Known Limitations**: Initial investigation reveals that Python ecosystem support presents significant technical challenges due to the lack of standardized package manager interfaces and inconsistent metadata availability.

The evaluation methodology employs a repository-of-repositories approach, aggregating representative projects across target ecosystems to enable systematic testing and validation of detection capabilities. This approach facilitates identification of both technical implementation requirements and fundamental ecosystem limitations that impact *SCA Tool* effectiveness.

2 Literature Review

This chapter provides an examination of the foundational concepts and technologies that facilitate SCA tools and their support for multiple ecosystems. To understand the challenges and requirements associated with extending *SCA Tool* to accommodate new dependency ecosystems, it is essential to first establish a thorough basis of core components that constitute modern software dependency management. This foundation encompasses three critical elements: package managers and build tools that orchestrate modern software component distribution and integration, SBOMs that provide structured inventories of software components, and PURLs that offer standardized identification schemes for software packages across different ecosystems. Second, I examine the current state of SCA tools, analyzing their architectural approaches and limitations in handling multi-ecosystem environments. Finally, I investigate existing multi-ecosystem analysis approaches, particularly examining tools like OSS Review Toolkit (ORT) and ScanCode that are utilized by *SCA Tool*.

2.1 Foundational Technologies for SCA

This section provides the necessary basis for understanding how developers manage software dependencies in development environments. I explore the role of package managers and build systems, the significance of SBOMs for software inventorying, and the standardization provided by PURLs for consistent software identification.

2.1.1 Package Managers and Build Systems

Package managers and build systems serve distinct but complementary roles in modern software development. They are specialized tools designed to automate the installation, management, and updating of external libraries required for code execution. These systems handle the complex task of dependency resolution, ensuring that all required components are available and compatible with each other. Build systems, in contrast, are responsible for packaging code and dependencies into deployable artifacts, transforming source code into production-ready applications (*An Introduction to Package Managers*, n.d.; Nejati, n.d.).

The fundamental distinction lies in their primary functions: package managers focus on dependency acquisition and management, while build systems orchestrate the compilation, linking, and packaging processes. However, this separation is not absolute, as many modern tools integrate both capabilities. For instance, Maven and Gradle in the Java ecosystem combine dependency management with build orchestration, while tools like npm in JavaScript serve primarily as package managers but also provide basic build capabilities (*Build & Package Management Concepts and Terminology*, n.d.).

The evolution of package managers and build systems has been driven by the increasing complexity of software projects and the growing reliance on third-party libraries. Modern applications often consist of 80% to 90% third-party packages, making effective dependency management crucial for security, performance, and maintainability. This reality has positioned package managers and build systems as critical components in

the software supply chain, serving as the primary interface between developers and the vast ecosystem of available libraries and frameworks (Vermeer, 2022).

The maturity and capabilities of package management ecosystems vary significantly across programming languages. Modern languages like Go and Rust have built-in dependency management systems that provide comprehensive metadata from the get-go. Java's ecosystem, with Maven and Gradle, offers sophisticated dependency resolution and extensive repository infrastructure. JavaScript's npm ecosystem provides detailed package information but presents SCA challenges through rapid package updates and micro-dependencies that create large, complex dependency graphs requiring extensive component analysis (*An Introduction to Package Managers*, n.d.; Kula et al., 2017; *Managing Dependencies*, n.d.; *Using Resolution Rules*, n.d.).

Python represents an interesting case with multiple competing package managers (pip, hatch, poetry), each with different approaches to dependency resolution and environment management. This fragmentation can complicate SCA, as different package managers may resolve dependencies differently or provide varying levels of metadata detail (Cofano et al., 2024).

C and C++ represent the most significant challenges for SCA tools due to their fragmented build ecosystem and lack of standardized package management. Several factors contribute to these difficulties:

- **Absence of Standardized Package Management**: Unlike modern languages with centralized package repositories, C/C++ projects often rely on manual dependency management, git submodules, or platform-specific package managers. This fragmentation makes it difficult for SCA tools to automatically discover and analyze dependencies (Haynes, 2025).
- **Complex Build Systems**: C/C++ projects frequently use complex build systems like Make, CMake, or Bazel, each with different approaches to dependency specifications. The build configuration may be split across multiple files, use conditional compilation, or rely on environment-specific settings that complicate automated analysis (Haynes, 2025).
- **Header-Only Libraries and Static Linking**: Many C/C++ libraries are distributed as header-only implementations or are statically linked into executables, making it difficult to identify their presence and versions in the final artifact. This contrasts sharply with languages that maintain clear separation between application code and dependencies (Haynes, 2025).

Modern package managers significantly ease the SCA process by providing structured metadata about dependencies, including version information, licensing details, security advisories, and dependency trees. This structured information is readily accessible to SCA tools without requiring complex parsing of build scripts. Centralized repositories like npm registry, Maven Central, and PyPI serve as authoritative sources for package information and security vulnerability databases. Lock files and dependency resolution capabilities ensure that SCA tools have precise information about the actual software

composition rather than just the declared dependencies (*JSON API - PyPI Docs*, n.d.; *Package-Lock.json*, n.d.; *Package.json*, n.d.).

2.1.2 Software Bill of Materials (SBOM)

SBOMs represent a keystone advancement in software transparency and supply chain security, providing standardized methods for documenting software composition that enable effective SCA.

A SBOM is a comprehensive inventory that provides detailed information about all software components, libraries and dependencies used in a software product. Similar to a manufacturing bill of materials that lists all parts in a physical product, an SBOM creates transparency in the software supply chain by documenting the composition of software applications (Bals, 2024).

The importance of SBOMs has been dramatically highlighted by major security incidents such as the Apache Log4j vulnerability (Log4Shell) in 2021, which demonstrated the critical need for organizations to understand their software dependencies. This vulnerability affected millions of applications worldwide, yet many organizations struggled to identify whether their systems were vulnerable due to a lack of visibility into their software supply chain (Bals, 2024).

The National Telecommunications and Information Administration (NTIA) has established minimum elements that SBOMs must contain to be considered effective. These baseline requirements include component name, supplier name, component version, unique identifier, dependency relationships, SBOM author, and timestamp. While these represent minimum requirements, comprehensive SBOMs often include additional information such as cryptographic hashes, license details, and vulnerability assessments (*The Minimum Elements for a Software Bill of Materials (SBOM)*, 2021).

The software industry has developed several standardized SBOM formats, with CycloneDX and SPDX representing the two dominant approaches. CycloneDX, developed by the OWASP Foundation, takes a security-centric approach with native support for vulnerability management, making it well-suited for DevSecOps environments focused on rapid threat response. SPDX, developed by the Linux Foundation, emphasizes comprehensive license compliance and intellectual property management, holding the distinction of being the only ISO-standardized SBOM format (ISO/IEC 5962:2021) with extensive legal documentation capabilities, though CycloneDX is also pursuing ISO standardization to achieve similar recognition. The choice between formats typically depends on organizational priorities: CycloneDX for security-focused organizations requiring continuous vulnerability monitoring, and SPDX for enterprises prioritizing license compliance and operating in regulated environments with complex intellectual property requirements (*About SPDX*, n.d.; *Authoritative Guide to SBOM - Implement and Optimize Use of Software Bill of Materials*, n.d.; *ISO/IEC 5962:2021 Information Technology - SPDX® Specification V2.2.1*, 2021; Neil, 2024; *Standardization Process*, 2023).

Despite their benefits, SBOM implementations face several significant challenges including accuracy and completeness issues, tool interoperability problems, time and resource consumption, and transparency concerns. Research has revealed significant challenges in SBOM generation and analysis regardless of format choice, with studies showing substantial variability in vulnerability detection when comparing SBOMs generated using different tools and formats (Sarwar, 2024; Xia et al., 2023).

2.1.3 Identification of Software using PURLs

Standardized software identification forms the foundation for effective dependency tracking and vulnerability management across diverse software ecosystems, with PURLs serving as the primary standardization mechanism.

Historically, the lack of standardization in software package identification led to fragmentation and limited interoperability between tools, languages, and databases. Each ecosystem and system used proprietary or ad-hoc methods to reference packages, complicating tasks such as vulnerability tracking, license compliance, and component reuse. Multiple organizations, including Grafeas, JFrog, and Libraries.io, developed unique, mutually incompatible schemes for package identification, resulting in inefficiencies and barriers to collaborative security efforts (Ombredanne, n.d.; *Package-Url/purl-Spec*, 2025).

A PURL is a standardized Uniform Resource Locator (URL) string format designed to identify and locate software packages in a universal and uniform way across programming languages, package managers, packaging conventions, tools, APIs, and databases. The primary purpose of PURLs is to provide a reliable reference system for software packages using a simple and expressive syntax based on familiar URL structures (Ombredanne, n.d.; *Package-Url/purl-Spec*, 2025).

The PURL specification defines a clear structure composed of seven distinct components that together create a unique and unambiguous identifier for a software package. A PURL is structured as follows:

```
pkg:<type>/<namespace>/<name>@<version>?<qualifiers>#<subpath>
```

Each component serves a specific purpose in uniquely identifying a software package:

- **Scheme**: The constant value "pkg" facilitates the potential future official registration of the "pkg" scheme to become a formally recognized URI, similar to how "http:" and "ftp:" are officially registered. (*required*)
- **Type**: Represents the package "type" or "protocol" such as maven, npm, nuget, gem, pypi, etc. (*required*)
- **Namespace**: A name prefix such as a Maven groupid, Docker image owner, or GitHub user/organization (*optional*)
- **Name**: The name of the package (*required*)
- **Version**: The version of the package (*optional but typically included*)
- **Qualifiers**: Extra qualifying data for a package such as operating system, architecture, or distribution (*optional*)

- **Subpath**: Specifies an extra subpath within a package, relative to the package root (*optional*)

These components form a hierarchy from the most significant on the left to the least significant on the right, creating a consistent and readable format for package identification. For example, the Maven package `pkg:maven/com.diffplug.spotless/spotless-plugin-gradle@6.25.0` demonstrates this structure where "maven" is the type, "com.diffplug.spotless" is the namespace (Maven groupId), "spotless-plugin-gradle" is the name (Maven artifactId), and "6.25.0" is the version (*Package-Url/purl-Spec*, 2025; *What Is a Package URL? - Amazon Inspector*, n.d.).

The PURL specification includes provisions for packages that don't fit into established package ecosystems through the generic PURL format. The generic format uses the structure

```
pkg:generic/<namespace>/<name>@<version>?<qualifiers>
```

and is particularly useful for compiled binaries, web platforms like Apache and WordPress, and custom legacy software or internally developed systems lacking formal package management (*What Is a Package URL? - Amazon Inspector*, n.d.).

SCA tools benefit significantly from the standardization provided by PURLs. These tools can consistently identify and track packages across different ecosystems and versions, map discovered vulnerabilities to specific package versions with precision, share information with other security tools and databases using a common identification scheme, and generate more accurate and interoperable reports.

2.2 State of the Art in SCA Tools

The contemporary SCA tool landscape demonstrates three primary architectural approaches to dependency detection, each addressing different aspects of software composition analysis complexity.

Manifest-Based Analysis

Manifest-based analysis represents the foundational approach that *SCA Tool* employs, analyzing package manager files such as Maven's `pom.xml`, Python's `requirements.txt`, or JavaScript's `package.json` to construct dependency graphs. Tools like *OWASP Dependency-Check*[3] exemplify this approach with their plugin-based analyzer architecture . This method offers high accuracy for projects with well-maintained package manager configurations and provides detailed version and licensing information (*Analyzing Projects for Dependencies (SCA)*, n.d.; *OWASP Dependency-Check*, n.d.).

However, manifest-based approaches create blind spots for dependencies introduced through binary inclusion, code cloning, or shading techniques (Dietrich et al., 2023).

[3]https://owasp.org/www-project-dependency-check/

Snippet Scanning

Snippet scanning identifies source code fragments that have been incorporated into projects through direct code inclusion rather than formal dependency management mechanisms. Such code incorporation practices create undocumented dependencies that do not appear in manifest files, resulting in untracked license obligations and potential security vulnerabilities. Snippet scanning addresses the gap between formally declared dependencies and actual code utilization patterns in software development practices (Black, 2024).

Binary Analysis

Binary analysis employs signature-based detection to identify components in compiled artifacts without source code access. Commercial tools like *Black Duck*[4] implement this approach using cryptographic fingerprinting against extensive signature databases. Binary analysis proves valuable for legacy systems and containerized applications where manifest files may be unavailable (*Black Duck Binary Analysis*, 2025).

Binary analysis faces significant technical challenges including compiler optimizations that alter signatures, static linking that obscures component boundaries, and substantial computational requirements for signature database maintenance. These challenges explain why binary analysis remains primarily in commercial tools rather than open-source solutions (Ren et al., 2021).

Runtime Analysis

Runtime analysis examines deployed environments to identify components missed by static methods. This includes container scanning and runtime environment inspection for dynamically loaded components. Tools like *Snyk*[5] demonstrate container-based approaches that analyze final deployment compositions (*How Snyk Container Works*, 2024).

Architectural Limitations

SCA tools use different approaches — manifest-based analysis, snippet scanning, binary analysis, and runtime analysis — but each has limitations preventing complete dependency detection. Manifest-based methods miss components introduced outside package files, snippet scanning produces false positives and lacks usage context, binary analysis struggles with compiler optimizations and requires extensive signature databases, and runtime analysis only captures what's active during execution. No single approach guarantees complete coverage, so effective SCA typically requires combining multiple methods while managing their trade-offs (Black, 2024; Dietrich et al., 2023; *How Snyk Container Works*, 2024; Ren et al., 2021).

[4]https://www.blackduck.com/

[5]https://snyk.io/

2.3 Multi-Ecosystem Analysis Tools

This section examines two prominent tools that address different aspects of multi-ecosystem software composition analysis. The ORT provides dependency resolution and SBOM generation capabilities across programming languages, while ScanCode Toolkit focuses on license and copyright detection through comprehensive text analysis. These tools are used and extended within *SCA Tool* to support new dependency ecosystems.

2.3.1 OSS-Review-Toolkit (ORT)

The ORT addresses the challenge of dependency analysis across polyglot software projects by providing unified dependency resolution and SBOM generation capabilities. ORT's primary focus is solving the problem of inconsistent dependency management practices across programming ecosystems, enabling organizations to maintain software composition visibility regardless of the underlying technology stack (*Analyzer | OSS Review Toolkit*, n.d.).

ORT employs a plugin-based architecture that abstracts the complexities of different package managers while leveraging existing dependency resolution tools. Rather than reimplementing dependency resolution from scratch, ORT coordinates with native package managers through standardized interfaces, preserving ecosystem-specific behaviors while providing consistent output formats. The analyzer component serves as the foundation of the ORT pipeline, whose output feeds into compliance and reporting tools (*Oss-Review-Toolkit/ort*, 2025).

The tool supports over 20 package managers across multiple programming languages. Each package manager is implemented as a plugin extending an abstract PackageManager class, with implementation approaches varying from direct API integration to command-line wrapper methods depending on ecosystem characteristics (*Analyzer | OSS Review Toolkit*, n.d.; *Oss-Review-Toolkit/ort*, 2025).

ORT's architecture follows a "hands-off" approach, requiring no modifications to existing project source code or build configurations, making it suitable for Continuous Integration (CI) / Continuous Deployment (CD) integration. The analyzer produces standardized output files containing complete dependency trees, package metadata, and project relationships that serve as input for downstream compliance and vulnerability analysis tools (*Analyzer | OSS Review Toolkit*, n.d.; *Oss-Review-Toolkit/ort-Ci-Gitlab*, 2025).

SCA Tool currently integrates ORT's analyzer capabilities through Command Line Interface (CLI) and API interfaces. This research evaluates the strengths and limitations of this integration, identifying opportunities to either bypass ORT's constraints or enhance its capabilities within *SCA Tool's* architecture.

2.3.2 ScanCode

ScanCode Toolkit addresses the challenge of license and copyright detection across diverse programming ecosystems through a fundamentally language-agnostic approach. Unlike dependency-focused tools, ScanCode's primary purpose is accurately identifying and extracting all licensing and intellectual property information embedded in software codebases, regardless of programming language or file format. This includes not only licensing obligations, but also copyright holders, license texts, author information, project metadata, disclaimers, and other legal notices found within license documentation (*Home — ScanCode-Toolkit Documentation*, n.d.).

ScanCode's core architecture combines universal text analysis with targeted ecosystem-specific enhancements. The tool's license, copyright, and origin detection operate on plain text extracted from files or binaries, using indexed patterns and grammar rules rather than programming language syntax. Because licenses are expressed in natural language phrases (e.g., "Copyright 2025 Alice Smith under MIT"), the detection heuristics can analyze unfamiliar programming languages or binary content without requiring new parser development (*Home — ScanCode-Toolkit Documentation*, n.d.; *License Detection Updates*, n.d.).

The license detection system employs a sophisticated pattern-matching approach with over 32,000 curated snippets, full texts, and notice templates compiled into an inverted index. For every file, ScanCode extracts text and queries the index, with matches above configurable thresholds becoming detections. A grammar engine merges adjacent snippets into single expressions and flags unknown references when context is insufficient (*Overview — ScanCode-Toolkit Documentation*, n.d.; *Scancode-Toolkit/CHANGELOG.rst*, n.d.).

While universal text scanning provides ScanCode's foundation, the tool includes over 100 package-manifest parsers spanning major ecosystems from JavaScript's npm to Python's pip. These parsers extract structured metadata—name, version, declared license, PURL — complementing rather than replacing the universal text-scanning engine. Each parser operates as a language-specific add-on that requires separate development for new ecosystems (*Supported Package Manifests and Package Datafiles*, n.d.).

ScanCode's plugin architecture enables extensibility through three primary hook groups: pre-scan for file filtering and extraction hints, scan for per-file analyzers (license, copyright detection), and post-scan for result aggregation and output formatting. The tool produces comprehensive results regardless of programming language, with consistent JSON output formats suitable for integration into larger SCA workflows (*Overview — ScanCode-Toolkit Documentation*, n.d.; *Plugin Architecture — ScanCode-Toolkit Documentation*, n.d.).

Current limitations include reduced accuracy with non-English license text and challenges with obfuscated or minified code where license information may be stripped. However, these limitations do not significantly impact ScanCode's utility for standard source code analysis across diverse programming ecosystems (*Scancode-Toolkit/CHANGELOG.rst*, n.d.; Wagner, 2023).

SCA Tool fully integrates ScanCode's capabilities for license and copyright detection components. Given ScanCode's mature and comprehensive approach to language-agnostic license detection, this functionality is considered sufficiently robust for operational requirements. Consequently, this research concentrates on dependency analysis and ecosystem extensibility challenges, where significant gaps remain in current *SCA Tool* capabilities.

3 Requirements

This chapter defines the requirements for extending *SCA Tool's* dependency ecosystem support capabilities, building upon the challenges and motivations established in chapter 1. The requirements address the specific limitations identified in supporting polyglot software environments and the need for comprehensive dependency detection across diverse package management systems. They are categorized into functional and non-functional specifications, with each assigned a unique identifier for reference throughout this thesis. These requirements were initially derived from the research objectives outlined in section 1.3.1 and subsequently refined during the development process to incorporate additional capabilities deemed necessary for effective multi-ecosystem support.

The primary focus centers on extending *SCA Tool's* capability to support additional dependency ecosystems, particularly Gradle/Maven for Java projects and pip with `requirements.txt` files for Python projects. Additional Python dependency managers are evaluated and implemented based on feasibility and impact during the development process. The fulfillment of these requirements will be assessed in the evaluation chapter, providing measurable criteria for the success of the implemented solutions.

3.1 Functional Requirements

Functional requirements define the specific behavior and capabilities that *SCA Tool* must provide to effectively analyze and manage dependencies across newly supported ecosystems. These requirements specify what the system must do to fulfill its intended purpose of comprehensive dependency analysis. A critical constraint is that existing functionality must be preserved and not degraded when adding support for new dependency ecosystems.

These requirements were iteratively refined during development as implementation challenges and ecosystem-specific complexities became apparent, reflecting the exploratory nature of extending SCA capabilities.

Existing SCA Tool Functionalities

- **F-01:** Support PURL version matching to remote artifacts and source code
- **F-02:** Fetch source code and artifacts that exhibit the closest version correspondence from remote sources, minimizing false positives in dependency resolution
- **F-03:** Support downloading source code from Git repositories
- **F-04:** Implement platform specific optimisations for dependency resolution and downloading
- **F-05:** Support transitive dependency analysis
- **F-06:** Implement ecosystem and platform specific license detection

New Functionalities for Extended Ecosystem Support

- **F-07:** *SCA Tool* must be able to analyze Java projects using Gradle and Maven as build tools.
- **F-08:** *SCA Tool* must be able to analyze standard Python projects using a *requirements.txt* file.
- **F-09:** Support Python Poetry dependency resolution (`pyproject.toml`, `poetry.lock`)
- **F-10:** Support Python Pipenv dependency resolution (`Pipfile`, `Pipfile.lock`)
- **F-11:** Support Python Conda environment files (`environment.yml`)
- **F-12:** Support Python uv environment files (`pyproject.toml`, `uv.lock`)
- **F-13:** Implement a tiered quality matching strategy for downloading dependencies
- **F-14:** Implement a fallback mechanism for dependency resolution when preferred methods fail

3.2 Non-Functional Requirements

Non-functional requirements establish the quality attributes and constraints that govern how the functional capabilities should be implemented and performed.

- **NF-01:** Features shall be added with performance considerations in mind. The additions should not introduce significant overhead compared to existing implementations.
- **NF-02:** Code shall be maintainable. Future developments should easily integrate into the existing codebase.
- **NF-03:** Error Handling: Tool shall gracefully handle network timeouts, invalid manifests, and missing repositories
- **NF-04:** Fault Tolerance: Tool shall continue analysis when individual dependencies cannot be resolved
- **NF-05:** Data Integrity: Tool shall validate all downloaded artifacts using checksums when available
- **NF-06:** Secure Communication: All network communications shall use HTTPS/SSH protocols

4 Architecture

Building on the foundational concepts of SCA and the challenges of diverse dependency ecosystems, this chapter presents an architectural overview of *SCA Tool* and its components, detailing the decisions that enable effective multi-language dependency processing. The focus lies on how the system's modular architecture enables extensibility while maintaining consistency across different programming language ecosystems.

4.1 Component-wise Construction of SCA Tool

The architectural foundation of *SCA Tool* implements a modular design strategy centered on separation of concerns. Figure 1 demonstrates this distributed service-oriented approach, where each component maintains distinct responsibilities within the software composition workflow, enabling system adaptation to new dependency ecosystems without compromising existing functionality.

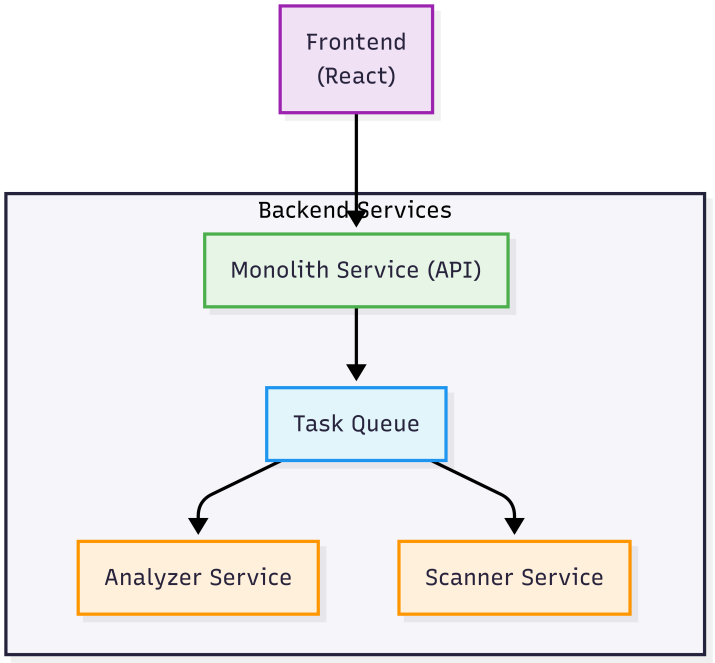

Figure 1: Service Communication Flow of SCA Tool

At the heart of the architecture lies the **Monolith Service**, which functions as the central coordinator and primary entry point for all system operations. The monolith integrates with several supporting infrastructure services including Redis for caching, MinIO for object storage, PostgreSQL for persistent data storage, and RabbitMQ for message queuing. Kratos is used for authentication and identity management capabilities, which are also integrated into the system.

The **Task Queue** component serves as the communication backbone between services, enabling asynchronous processing and decoupling of the analysis and scanning workflow. This queue-based architecture allows for scalable processing of SCA tasks and ensures system resilience by providing reliable message delivery between components.

The **Analyzer Service** and **Scanner Service** represent the core functional components where this thesis work is concentrated. These services are designed as independent, specialized components that can operate autonomously while communicating through the task queue system. This design pattern allows for:

- **Modularity**: Each service can be developed, tested, and deployed independently
- **Scalability**: Services can be scaled based on demand
- **Extensibility**: New dependency ecosystems can be supported by extending these services without affecting the overall system architecture

This architectural approach provides a solid foundation for supporting new dependency ecosystems, as the modular design allows for the extension of both analysis and scanning capabilities without requiring significant changes to the existing system infrastructure.

4.2 Analyzer

The `Analyzer` component serves as the central processing engine within *SCA Tool*, serving as the primary determinant of the platform's analytical quality and accuracy. As the central processing unit responsible for dependency identification and SBOM generation, the `Analyzer`'s implementation directly influences the reliability and comprehensiveness of security assessments and license compliance evaluations performed by the entire platform.

Operating within the distributed microservices architecture, the `Analyzer` executes a systematic examination workflow that processes software projects through a well-defined pipeline, ensuring consistent analysis across diverse project types and programming languages. The service maintains architectural flexibility through message-based communication with other system components, enabling horizontal scaling through multiple container instances while preserving loose coupling and fault tolerance in distributed deployment scenarios.

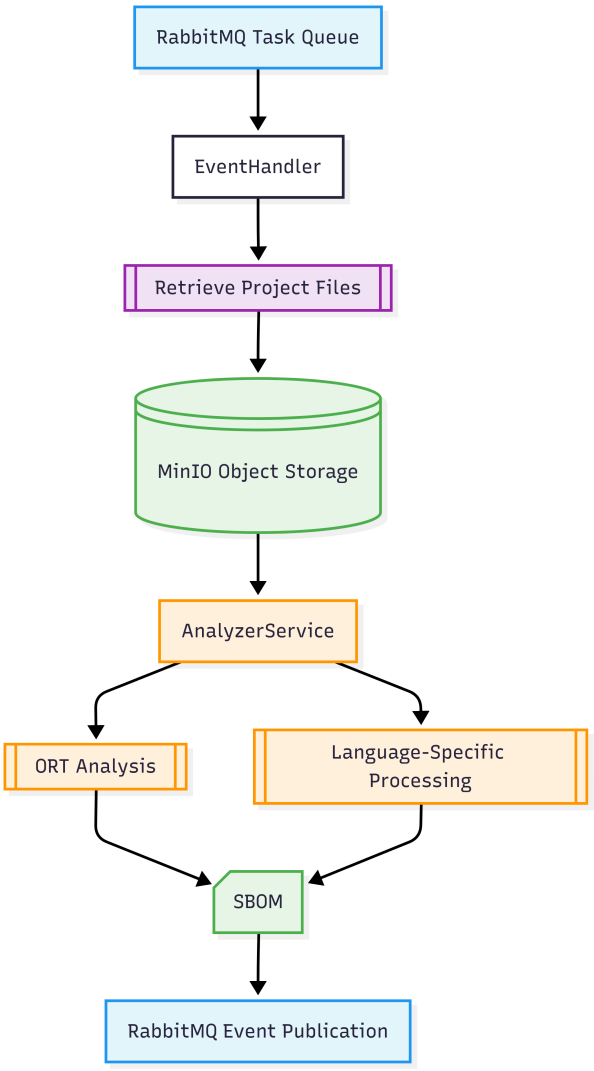

Figure 2: Analyzer Task Processing Flow

Figure 2 illustrates the `Analyzer`'s core processing architecture. The workflow initiates when the `EventHandler` receives analysis tasks from the RabbitMQ task queue, each containing project metadata and storage location information. Upon task receipt, the system retrieves project artifacts from MinIO object storage, establishing the local workspace necessary for dependency analysis.

The `AnalyzerService` coordinates the central analysis process, leveraging ORT as the primary dependency resolution engine. As established in previous chapters, ORT provides comprehensive language-agnostic support for numerous programming ecosystems through its plugin-based architecture. This enables the `Analyzer` to handle the majority of dependency analysis scenarios through standardized interfaces and proven resolution algorithms.

Following successful dependency resolution, the system generates SBOMs containing comprehensive metadata about identified components, including version information, license details, and dependency relationships. The resulting analysis artifacts are packaged into structured results and transmitted to downstream services via RabbitMQ

event publication, maintaining the asynchronous processing model throughout the system.

However, the heterogeneous nature of modern software development presents challenges that extend beyond ORT's standard capabilities. While ORT excels in many scenarios, certain programming ecosystems exhibit unique characteristics that require specialized handling approaches. Complex dependency resolution mechanisms, non-standard package formats, and ecosystem-specific metadata requirements can necessitate enhanced or alternative analysis strategies to maintain accuracy and completeness in dependency identification.

4.2.1 Language-Specific Support Components

While ORT provides comprehensive support for analyzing dependencies across numerous programming language ecosystems, practical implementation reveals certain quality deficits and limitations that impact the accuracy and reliability of dependency analysis. Through extensive testing and evaluation during this thesis work, as well as observations from other contributors to the project, several programming languages and package managers have been identified as requiring enhanced or completely custom implementation approaches to achieve the analytical quality standards demanded by *SCA Tool*.

These quality issues manifest in various forms: incomplete dependency resolution, inaccurate version handling, insufficient metadata extraction, and compatibility problems with specific package manager configurations (detailed analysis in chapter 5). Additionally, the need to support emerging package managers, proprietary dependency formats, and organization-specific configuration patterns necessitates implementation flexibility that extends beyond ORT's current scope. To address these limitations while maintaining architectural agility, *SCA Tool* implements language-specific support components that either enhance ORT's existing capabilities or completely bypass ORT's analysis pipeline in favor of custom extraction logic.

This strategic approach acknowledges the practical constraints of contributing to large-scale open-source projects like ORT, where modification cycles may not align with *SCA Tool's* development requirements. The modular architecture enables rapid adaptation to new dependency management paradigms and supports the long-term architectural goal of achieving complete analytical independence. Specific implementation challenges and solutions for Python ecosystem support are examined comprehensively in section 5.1.6, demonstrating the practical application of these architectural principles.

Architecture and Data Flow

The language-specific support components integrate seamlessly through a standardized interface that maintains compatibility with ORT's output format. This architectural design enables the combination of ORT-generated results with custom extraction outputs, ultimately producing unified SBOMs regardless of the underlying analysis method employed.

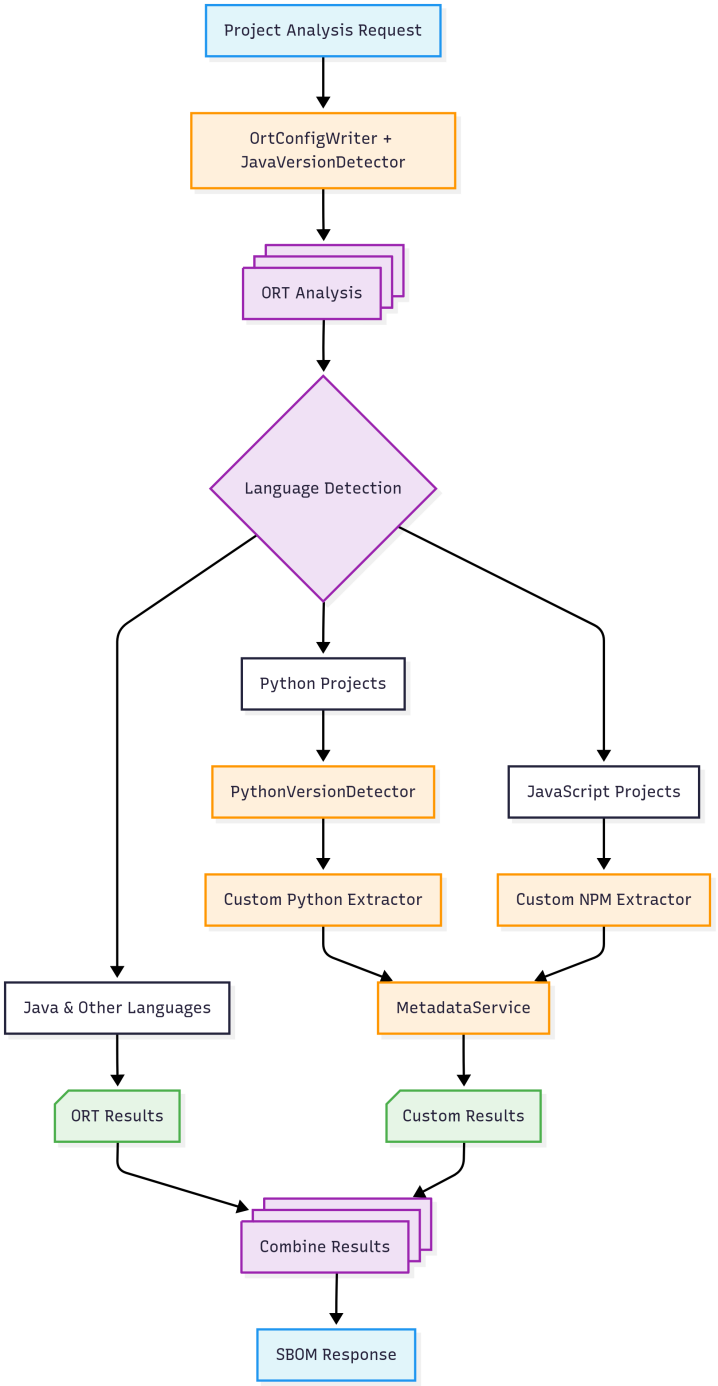

Figure 3: Language-Specific Support Flow Overview

As illustrated in figure 3, for enhanced ORT implementations, such as Java analysis, the system maintains ORT's core analysis pipeline while injecting additional configuration and environment setup before execution to ensure optimal compatibility. The JavaVersionDetector first identifies the appropriate JDK version requirements and then configures ORT accordingly, preventing version mismatches that could compromise dependency resolution accuracy.

When an analysis is performed regardless of the programming language or package manager the entire project is processed through ORT to ensure that polyglot characteristics are handled correctly. This means that even if a project contains multiple

languages, ORT will analyze all dependencies (for enabled languages) and generate a comprehensive SBOM that includes all components if no exclusion rule is specified.

Custom extraction always concludes after a comprehensive ORT analysis. Excluded package managers or programming languages that have been skipped during the ORT analysis phase are now processed through specialized extractors that implement tailored - ecosystem-specific dependency resolution logic. The `MetadataService` plays a crucial role in this architecture, as custom extractors require independent metadata resolution capabilities. While ORT-processed dependencies include comprehensive metadata as part of the standard analysis pipeline, bypassed implementations must explicitly fetch and resolve package metadata through dedicated service calls. After Metadata fetching, custom extractors are fully compatible with ORT's output format, allowing simple integration into the overall analysis pipeline and ensuring that all dependencies, regardless of their source or analysis method, are represented in the final SBOM.

Table 1 showcases a comparision of the analysis methods employed for different programming languages within *SCA Tool*. The table highlights the primary components used for dependency resolution, the configuration sources required, and the reasons for custom implementations in specific cases (explained further in chapter 5).

Language	**Analysis Method**	**Primary Components**	**Configuration Sources**	**Reason for Custom Implementation**
Java	ORT Enhanced	JavaVersion-Detector, OrtConfigWriter	pom.xml, build.gradle, gradle.properties	JDK version compatibility requirements
Python	Full Custom	PythonVersion-Detector, Custom Extractor, MetadataService	Too many to list. See table 2 for a complete list.	Poor quality of python-inspector in ORT
JavaScript	Full Custom	NPM Package Parser, Registry Client, MetadataService	package-lock.json, npm-shrinkwrap.json	Insufficient module support in ORT
Other Languages	Standard ORT	ORT Native Analyzers	Language-specific manifests	

Table 1: Analysis Method Comparison by Programming Language

This hybrid approach ensures that *SCA Tool* maintains broad language ecosystem support through ORT's extensive capabilities while providing improved accuracy and reliability for the most critical and frequently encountered programming languages in enterprise software development environments.

Despite the need for custom implementations in specific cases, ORT remains an excellent foundational choice for *SCA Tool's* analysis infrastructure. The continuous development and improvement of ORT directly benefits *SCA Tool*, as updates to ORT's analyzers automatically enhance the platform's capabilities for supported languages. When ORT contributors improve language-specific implementations or add support for new package managers, these enhancements are immediately available to *SCA Tool* without requiring additional development effort. This symbiotic relationship ensures that *SCA Tool* benefits from the collective expertise of the broader open-source community while maintaining the flexibility to implement targeted improvements where necessary.

4.3 Scanner

The `Scanner` service functions as the specialized analytical engine within *SCA Tool*'s existing architecture, dedicated exclusively to performing comprehensive license and copyright analysis across software dependencies. Operating as a Spring Boot-based microservice within the distributed architecture, the `Scanner` executes a systematic pipeline that processes source code artifacts to identify licensing terms, copyright statements, and intellectual property obligations essential for legal compliance assessment.

Architecture and Processing Pipeline

The `Scanner`'s architecture is fundamentally composed of two primary subsystems: the **Downloader** for source code acquisition and the **ScanCode** integration layer for license and copyright analysis execution. This modular design ensures clear separation of concerns.

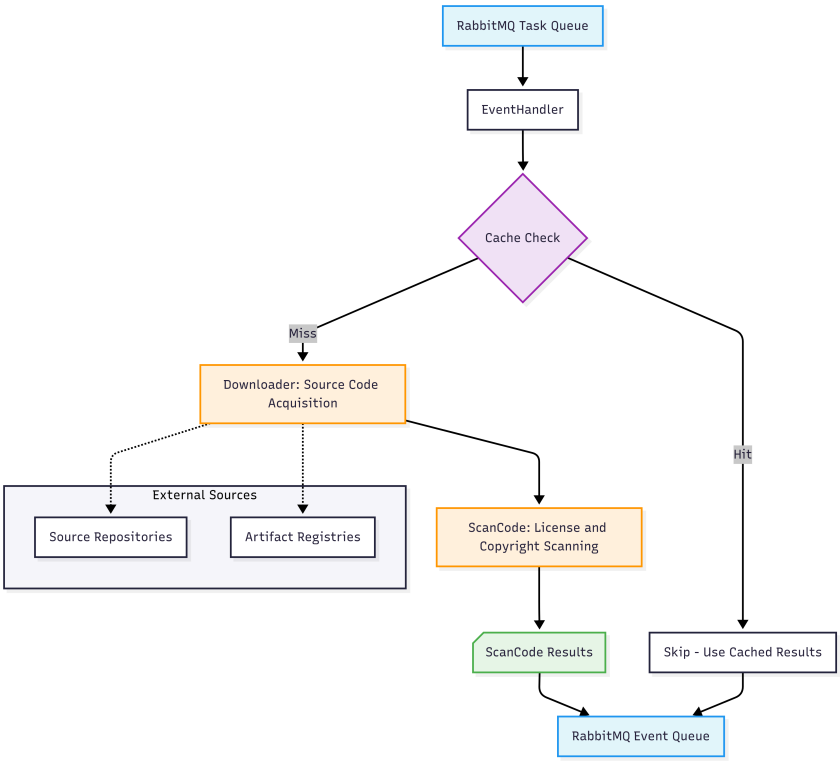

Figure 4: Scanner Service Processing Flow Overview

Figure 4 illustrates the `Scanner`'s core processing architecture. The workflow initiates when the `EventHandler` receives a scan request from the RabbitMQ task queue, each containing dependency metadata and source code location information. Upon request receipt, the system immediately performs cache validation against Redis to eliminate redundant processing of previously analyzed components.

For uncached dependencies, the Downloader subsystem retrieves source code artifacts from their respective repositories or artifact registries. ScanCode then performs comprehensive static analysis, generating detailed a report containing identified licenses and copyright statements.

The scanning results are processed and packaged into structured `ScanCompleteMessage` events, which are published to the RabbitMQ event queue for consumption by downstream services. Throughout this pipeline, the system maintains efficient resource utilization through intelligent caching strategies and parallel processing capabilities.

The two-subsystem architecture provides distinct advantages: the Downloader component implements extensible source acquisition strategies supporting multiple platforms and protocols, while the ScanCode integration maintains flexibility across different deployment environments. This separation enables independent optimization and maintenance of each subsystem while preserving cohesive operation.

ScanCode Integration Rationale

The utilization of ScanCode as the core analysis engine aligns with strategic considerations validated during the literature research phase (detailed in section 2.3.2). The findings support this architectural decision, as ScanCode provides mature license detection capabilities with extensive database coverage spanning thousands of known licenses and license variants. Its language-agnostic design philosophy demonstrates compatibility with *SCA Tool's* multi-ecosystem support requirements, enabling consistent analysis quality regardless of programming language or framework. The tool's active development community and comprehensive documentation validate the team's decision to adopt this solution, ensuring long-term viability and continuous improvement of detection capabilities.

4.3.1 Downloader

The Downloader subsystem implements a generalized approach to source code acquisition that handles dependencies regardless of their hosting platform or package manager, while maintaining the flexibility necessary to optimize acquisition strategies for specific platforms and protocols. Current iterations of the downloader focus on Git-based repositories, but the architecture is designed to accommodate future expansion to additional hosting paradigms as needed.

4.3.1.1 Download Strategy Resolution

Figure 5 demonstrates the extensible class hierarchy that enables platform-specific optimization while maintaining architectural consistency. The `DownloaderService` serves as the central orchestrator, implementing a strategy pattern through its collection of `VcsDownloader` implementations. This design enables seamless addition of new platform support without modifying existing code.

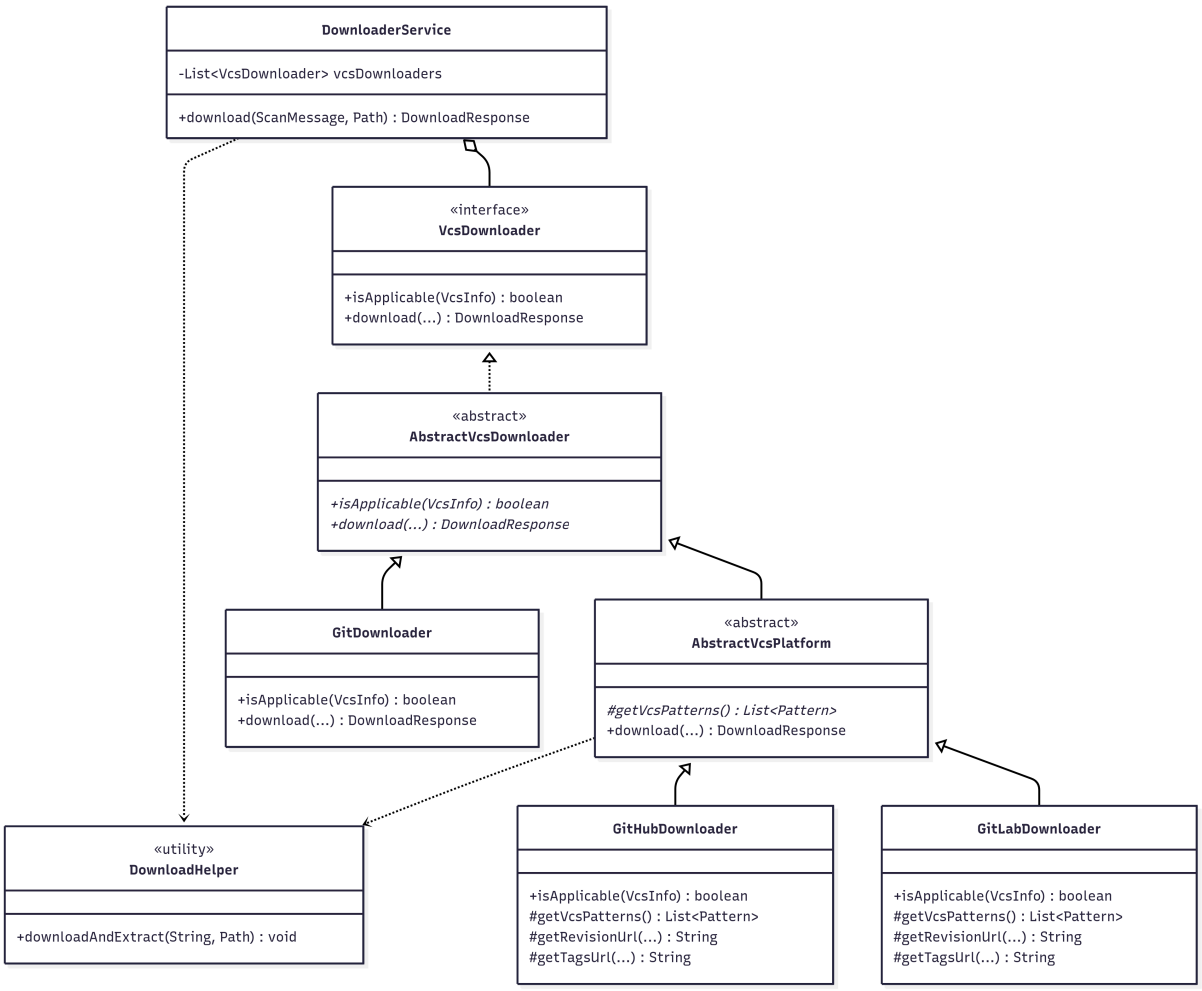

Figure 5: Downloader Subsystem Class Architecture

The download resolution process follows a systematic decision hierarchy. When a scan request contains a direct download URL, the system immediately delegates to the `DownloadHelper` utility for efficient artifact retrieval and extraction.

When direct URLs are unavailable, the system extracts Version Control System (VCS) information from the package metadata and initiates VCS-based acquisition. The platform detection mechanism evaluates the repository URL against known patterns, selecting the most appropriate downloader implementation. GitHub and GitLab repositories receive specialized handling through their respective `GitHubDownloader` and `GitLabDownloader` implementations, which leverage platform-specific APIs for optimized access to release archives and tag resolution.

Repositories that do not match known platform patterns utilize the generic `GitDownloader` implementation, which provides coverage through the standard Git protocol. This fallback mechanism ensures successful acquisition even for self-hosted Git instances, enterprise repositories, or emerging platforms not yet supported by dedicated implementations.

The `AbstractVcsPlatform` base class encapsulates common API-based functionality, providing template methods for URL construction, tag processing, and revision resolution. This abstraction significantly reduces implementation overhead for new platform-specific downloaders while ensuring consistent behavior across different hosting services.

Architectural Considerations

While the conceptual architecture maintains relative simplicity through its abstraction and strategy-based approach, the implementation encompasses significant complexity to accommodate the heterogeneous nature of modern software ecosystems. Package registries such as PyPI, Maven Central, and npm present unique challenges when source repositories cannot be automatically identified by the *Analyzer Service* in section 4.2.

These artifacts require different handling since they don't come from simple Git repositories. Instead, the system must query package registries directly, determine the packaging format for each artifact, and extract the contents accordingly. However, these registry artifacts typically contain only the essential code files, excluding project configuration files, documentation, license files, and other metadata that would be present in the full source repository. This reduced file set can decrease the accuracy of license detection in subsequent analysis phases (implementation in section 5.2.3).

Within the scope of this thesis work, it has been determined that the correct identification of download sources based on package identifiers is non-trivial and presents various challenges. The complexity arises particularly from inconsistent versioning schemas between package registries and the underlying VCSs, as well as from the necessity to distinguish between different artifact types (source code, pre-built packages, documentation) and to identify the artifacts relevant for license analysis.

These findings have led to the development of robust fallback mechanisms and intelligent heuristics that ensure reliable source code acquisition even under suboptimal conditions, while simultaneously maintaining the analytical quality of the downstream scanning process. The modular architecture enables targeted improvements to specific platform integrations without compromising the stability of the overall system.

5 Design and Implementation

This chapter presents the detailed design and implementation of the *SCA Tool's* core components, focusing on the integration of new dependency ecosystems and the challenges associated with multi-language software composition analysis.

5.1 Analyzer

As outlined in section 4.2, the `Analyzer` service serves as the dependency resolution engine within *SCA Tool*, responsible for generating SBOMs. During the experimental phase of this thesis where *SCA Tool's* (commit `dea3c420e5a7da9a5653647b097d9acb796ae03b`) Java and Python capabilities were initially tested, ORT (version `44.0.0` to `55.0.0`) didn't support critical aspects of dependency resolution that *SCA Tool* requires to offer a satisfactory user experience. Therefore a hybrid approach was chosen as the path forward as illustrated in figure 2 and table 1. The `Analyzer` combines ORT for a broad range of programming languages with custom extensions for specialized dependency analysis. The implementation addresses the fundamental challenge that different dependency ecosystems require varying levels of specialized handling. While some ecosystems benefit from ORT's mature analysis capabilities, others necessitate fully custom implementations to handle ecosystem-specific complexities and emerging package management patterns that are not yet supported by standard tooling.

5.1.1 General Behaviour

The `Analyzer`'s general workflow is built upon a message-driven architecture that transforms analysis tasks into SBOMs through a multi-stage processing pipeline. The component operates as a RabbitMQ consumer, processing analysis requests asynchronously while maintaining high reliability through retry mechanisms and dead letter queue handling.

Upon the `AnalyzerEventHandler` receiving an `AnalyzeEvent`, the system creates a temporary directory as a workspace and initiates a three-phase process: first, it downloads the project's source code from a MinIO object storage instance; second it executes the dependency analysis using the `AnalyzerService`, which orchestrates the analysis workflow; and finally, it publishes the resulting SBOM data through an `AnalyzerCompletedEvent` back to the message queue for further processing by downstream services.

```
fun analyze(tempDir: Path, input: Path): AnalyzerResult {
  // First we run ORT
  val ortResult = ortService.analyze(tempDir, input)

  // Then we run our own analyzer
  val result = doAnalyze(input)

  // Finally the results are merged
  return ortResult.merge(result)
}

fun doAnalyze(input: Path): AnalyzerResult {
  // The ExtractionService contains all our custom extractors
  val bom = extractionService.resolveBom(input)
  ...
  return AnalyzerResult(...)
}

// For merging the results we use a simple merge function
fun AnalyzerResult.merge(other: AnalyzerResult): AnalyzerResult { ... }
```

Listing 1: Core analysis logic in `AnalyzerService.kt` class

The analysis workflow in the `AnalyzerService` class employs a dual-stage approach (see listing 1), combining ORT's comprehensive ecosystem support with custom extraction logic. The ORT analysis handles a vast set of programming languages and package managers, that act as solid foundation for the majority of ecosystems to increase *SCA Tool's* coverage, generating an initial SBOM that includes all dependencies detected by ORT's native analyzers.

This second stage supplements the standard ORT dependency extraction with an `ExtractionService` that runs a set of *SCA Tool* specific `Extractor`'s through its `resolveBom` method. Custom extractors implement the `Extractor` interface and are passed along to the `ExtractionService` available in listing 2, which combines their individual `ExtractionResult` outputs into a unified dependency analysis.

```
@Service
class ExtractionService(private val extractors: List<Extractor>) {
  fun resolveBom(input: Path): ExtractionResult {
    val extractors = extractors.mapNotNull { it.extract(input) }
    if (extractors.isNotEmpty()) {
      return extractors.reduce(ExtractionResult::merge)
    }
    return ExtractionResult(...)
  }
}
```

Listing 2: Multi-extractor aggregation logic in `ExtractionService.kt` class

Not only does the merging process in listing 1 and listing 2 enhance the overall accuracy of the SBOM, but it also ensures that all relevant dependencies are captured, regardless of the analyzer that detected them. Projects don't have to be from a single supported ecosystem but can be polyglot, meaning that they can contain dependencies from

multiple programming languages and package managers. This is particularly important for modern software projects that often utilize a mix of different front-end and backend languages, employ different scripting languages for tools, and contain various components all within a large monorepo. The merged SBOM provides a unified view of the project's complete dependency landscape.

5.1.2 ORT Integration

The OrtService contained in listing 1 employs an `OrtRunner` that orchestrates the execution of ORT through either a containerized Docker environment or a local installation. The system dynamically generates ORT configurations based on detected project characteristics such as the project's Java version and executes the analysis through ORT's command-line interface.

The Docker-based execution approach utilizes a specific ORT container image (`ghcr.io/oss-review-toolkit/ort-minimal:55.2.0`) to provide isolated analysis environments. The `startAnalysis` method constructs a Docker command that mounts three critical filesystem volumes: the generated ORT configuration file is mounted to `/.ort/ort.yaml` within the container, the project source code is mounted to `/project/input`, and a dedicated output directory is mounted to `/project/output`. This volume mounting strategy allows the containerized ORT instance to access the project files while writing analysis results back to the host filesystem in a controlled manner. The Docker container executes ORT's `analyze` command with specified input and output paths, producing structured analysis results in JSON format (`analyzer-result.json`) that are subsequently parsed by the `Analyzer` to extract package information, dependency relationships, and metadata for integration into the unified SBOM structure.

5.1.3 Language-Specific Configuration & Behavior

As already outlined, the `Analyzer` implements distinct analysis strategies for different programming language and package manager ecosystems, reflecting the heterogeneous nature of dependency management across software development.

To ensure timely completion of analysis tasks and prevent inaccurate results, the `Analyzer` is configured to skip certain package managers that are handled by custom extractors within the `ort.yaml` configuration file (see listing 3).

```yaml
ort:
  analyzer:
    disabledPackageManagers: [NPM, PIP, Pipenv, Poetry]
```

Listing 3: `ort.yaml` configuration for disabled package managers

JavaScript and Python ecosystems are particularly notable examples where custom extractors are employed to handle specific package management patterns that ORT does not adequately support. The subsequent sections, section 5.1.5 and section 5.1.6, detail the reasoning and implementation strategies for these ecosystems.

5.1.4 Java

Java represents an interesting case within the ORT, as it is one of the most widely used programming languages in enterprise software development and notably, ORT itself is implemented in Kotlin, a JVM-based language that compiles to Java bytecode. The `Analyzer` leverages ORT's native support for Java through the Gradle and Maven package managers, which provide dependency resolution capabilities.

During the experimental testing phase of this thesis, it was observed that ORT's Java analysis capabilities were generally reliable, but certain aspects such as build tool and runtime version compatibility required additional effort to ensure accurate dependency resolution. Fortunately, ORT allows the user to configure the Java environment through the `ort.yaml` configuration file, enabling the specification of the Java home directory and other relevant settings as shown in listing 4.

```yaml
ort:
  analyzer:
    packageManagers:
      GradleInspector:
        options:
          javaHome: "{{JAVA_HOME}}"
      Maven:
        options:
          javaHome: "{{JAVA_HOME}}"
```

Listing 4: Java-specific configuration for ORT's Gradle and Maven package managers

Detecting the correct Java version

The Analyzer's implementation includes a custom `JavaVersionDetector` developed by the *SCA Tool* team that identifies the required Java Development Kit (JDK) version based on project configuration files. This detection mechanism is critical for Java ecosystem projects where build tool and runtime version compatibility directly impacts analysis success.

The detection strategy employs a hierarchical approach, examining multiple sources in order of reliability:

1. Wrapper Properties Files: Primary detection through `gradle/wrapper/gradle-wrapper.properties` or `.mvn/wrapper/maven-wrapper.properties`
2. Build Scripts: Secondary detection via `build.gradle`, `build.gradle.kts`, or `pom.xml` version declarations
3. Multi-module Support: Recursive scanning of subdirectories for wrapper configurations
4. Fallback Mechanisms: Default to Java 21 when detection fails

The version mapping logic follows official compatibility matrices from Gradle and Maven documentation[6].

[6]https://docs.gradle.org/current/userguide/compatibility.html

Correct Java version detection is essential because build tools exhibit version-specific behaviors that directly affect dependency resolution accuracy. Projects utilizing Java 17+ language features will fail analysis when executed with Java 8, resulting in incomplete dependency graphs. Similarly, Gradle plugins often require specific Java versions, and version mismatches cause plugin loading failures that prevent proper transitive dependency discovery.

The impact extends beyond mere compatibility—using incorrect Java versions can lead to missing security-critical dependencies, incomplete SBOMs, and false vulnerability assessments. For instance, a Spring Boot 3.x project requiring Java 17+ would produce fundamentally flawed analysis results if processed with Java 8, compromising the entire security assessment pipeline.

JDK Acquisition

The ORT Docker image includes a pre-configured Java environment, which is utilized for analysis tasks. ORT itself is implemented in Kotlin and Java, requiring a minimum Java 11 runtime for execution. If a project requires a different JDK version than the one provided by the ORT image, a compatible installation must be mounted into the container and configured through the `ort.yaml` configuration file.

To address this requirement, a dedicated script (pseudocode see listing 6) downloads JDK installations from the Eclipse Adoptium project (formerly AdoptOpenJDK) for versions 8, 11, 17, and 21 into a named ("analyzer-libs-volume") Docker volume. This script must be executed before *SCA Tool* is started, ensuring that the required Java runtimes are available for analysis tasks. The choice of Adoptium as the source provides reliable, TCK-tested OpenJDK distributions with consistent security updates and long-term support.

```
constant JAVA_DOWNLOADS = {
 | 8: {…}, 11: {url, filename, checksum}, 17: {…}, 21: {…}
}

function main
 | for each version in JAVA_DOWNLOADS
 |  | call downloadAndInstallJava(version)
 | end for
end function

function downloadAndInstallJava(version)
 | set downloadPath to TEMP_DIR + "downloads" + version.filename
 | set installDir to JAVA_HOME_BASE + "jdk-" + version.key
 | if installDir already exists
 |  | return
 | end if
 |
 | call downloadFile(version.url, downloadPath)
 | call validateChecksum(downloadPath, version.checksum)
 | call extractTarGz(downloadPath, installDir)
 | call deleteFile(downloadPath)
end function
```

Figure 6: Pseudocode for Java version download and installation script

A notable limitation of this approach is the unavailability of Java versions below 8 through Adoptium. Legacy Java versions (6, 7) are increasingly difficult to obtain due to Oracle's licensing changes, end-of-life status, and the lack of maintained open-source distributions. Modern security practices also discourage the use of these unsupported versions, making their absence less problematic for contemporary software analysis (*Oracle Java SE Support Roadmap*, 2025).

For local development and testing environments, a key enhancement was implemented involving conditional mounting (listing 5) of the analyzer-libs volume containing pre-downloaded Java runtimes.

```
val command = mutableListOf("docker", "run", ...)

if (analyzerLibsPath.isNotEmpty()) {
    command.addAll(listOf("-v", "analyzer-libs-volume:$analyzerLibsPath"))
}
```

Listing 5: Conditional volume mounting in `DockeredOrtRunner.kt`

The one time-execution of the script (listing 6) beforehand and mounting of the volume in the `DockeredOrtRunner` class ensures that the ORT container can access the required Java versions without needing to download them individually for each analysis task, significantly improving performance and reducing unnecessary network overhead. This approach also simplifies local development setups, as developers can easily execute the script on their machines without having to download Java versions manually. The mounted volume provides a consistent environment for analysis tasks, ensuring that the correct Java version is always available when needed.

5.1.5 Javascript

JavaScript presents unique challenges that required custom extraction capabilities beyond ORT's native support. The JavaScript ecosystem's complexity stems from its dynamic dependency resolution and modern workspace/monorepo architectures that standard analysis tools struggle to handle reliably. This section provides only a brief overview as the JavaScript extractor implementation was developed by other team members. The following Python section demonstrates the comprehensive extractor architecture and metadata service integration patterns that apply across supported ecosystems.

The custom extractor uses the `npm ls --json --package-lock-only` CLI command to generate complete dependency trees that properly distinguish between production dependencies, development dependencies, and optional dependencies. It also correctly identifies workspace modules within monorepo structures by detecting `file:` protocol references, ensuring that internal project relationships are captured accurately.

Unlike other package ecosystems, JavaScript packages often contain complex repository metadata that requires custom parsing. The npm registry API provides rich information that requires ecosystem-specific processing, similar to the metadata acquisition approaches detailed in the Python implementation (section 5.1.6).

5.1.6 Python

Another language that requires a custom extractor is Python, which presents unique challenges due to its fragmented package management ecosystem. The existing ORT Python extractor already supports the package managers pip, Pipenv and Poetry, but various issues during initial testing arose, necessitating the development of a more robust solution where changes to the underlying analysis infrastructure can be made without going through hoops of upstreaming changes to ORT and its dependencies, such as *python-inspector*[7].

The existing ORT Python extractor exhibited several technical limitations that compromised dependency resolution reliability. Analysis revealed frequent `ResolutionImpossible` and `RequirementsConflicted` exceptions in the underlying *python-inspector* tool, particularly when processing projects with pinned dependency versions that created incompatible constraint sets as ORT provided an incompatible Python version. Additionally the deprecation notice of *pkg_resources* APIs during de-

[7]https://github.com/aboutcode-org/python-inspector

pendency extraction indicated the use of a too modern version of Python for resolution of these dependencies. These issues were exacerbated by the ORT's incorrect conversion of PyPI index URLs, where the `-i https://pypi.org/simple` directive was improperly tokenized by *python-inspector* into individual character-based repository entries, leading to `Unsupported URL scheme` errors.

Therefore, the `Analyzer` implements multiple specialized extractors (see table 2), each tailored to specific package management tools while maintaining a unified interface through a common `PipExtractor`. This means that all Python packaging formats other than pip's standard `requirements.txt` file are first converted to a `requirements.txt` format before being processed by the `PipExtractor`. This design allows for centralized dependency resolution logic while accommodating most of the unique characteristics of each package manager.

Package Manager	**Dependency Files**	**ORT Included**
pip	`requirements.txt`	Yes
Pipenv	`Pipfile.lock`	Yes
Poetry	`poetry.lock`	Yes
uv	`uv.lock`	No - Added by implementation

Table 2: Table of Supported Python Package Managers

As for an example, the `PoetryExtractor` converts Poetry's `poetry.lock` file into a `requirements.txt` format using Poetry's native export command (`poetry export`), ensuring that all dependencies are captured accurately.

Upon the `ExtractionService`'s call of `extract()` on all extractors (see listing 2), the `PoetryExtractor` is invoked to process the given input directory, which contains a projects source code and its package management files (see listing 6).

```
@Component
class PoetryExtractor(private val pipExtractor: PipExtractor) : Extractor {

  override fun extract(input: Path): ExtractionResult? {
    val lockFiles = Files.walk(input).use { stream ->
      stream.filter {
        val name = it.fileName.toString()
        (name == "poetry.lock") // handle project files that match this name
      }.toList()
    }

    var finalResult: ExtractionResult? = null
    lockFiles.forEach { lockFile ->
      parseLockFile(lockFile).let { partial ->
        finalResult = finalResult?.merge(partial) ?: partial
      }
    }

    return finalResult
  }
}
```

Listing 6: PoetryExtractor `extract()` implementation

If the `poetry.lock` file is found, the extractor parses it and converts it to a `requirements.txt` format using the `parseLockFile()` method (see listing 7). This method runs the `poetry export` command to generate a requirements file, which is then processed by the `PipExtractor` to extract dependencies in a standardized format.

```
  private fun parseLockFile(lockFile: Path): ExtractionResult {
    val workingDir = lockFile.parent

    val exportProcess = runPoetryCommand(
      workingDir.toAbsolutePath(),
      "export", "-o", "converted-requirements.txt"
    )

    val requirementsFile = workingDir.resolve("converted-requirements.txt")

    pipExtractor.extract(requirementsFile).let { result ->
      return result ?: throw RuntimeException(...)
    }
```

Listing 7: PoetryExtractor lock file parsing

The `PipfileExtractor` (Pipenv) and `UvExtractor` implementations follow similar patterns to ensure comprehensive coverage of Python package management formats. The `PipfileExtractor` processes `Pipfile.lock` files generated by Pipenv, while the `UvExtractor` handles `uv.lock` files from the uv package manager — a format not natively supported by ORT.

Both extractors exclusively target lock files rather than manifest files like `Pipfile` or `pyproject.toml`. This represents a deliberate architectural decision that prioritizes

reproducible dependency analysis over broader format support. While this approach limits compatibility with projects that only maintain manifest files without corresponding lock files, it ensures that security analysis operates against the exact dependency tree that was resolved at build time. This precision is critical for accurate vulnerability assessment, as manifest files contain version ranges that could resolve to different dependency sets across environments, potentially missing or misidentifying security issues in the actual runtime dependencies.

The trade-off involves requiring development teams to commit lock files to their repositories, but this constraint delivers more reliable and reproducible security scanning results.

Unified Requirements Processing Strategy

ORT's native Python support heavily relied on `python-inspector`, which under the hood converts various Python package management formats into a common `requirements.txt` format. This approach proved itself to be effective, as it allows for a consistent dependency resolution logic that can be applied across different package managers. The `PipExtractor` serves as the central component that processes this common format, ensuring that all dependencies are extracted in a standardized manner.

Pip is the most widely used package manager in the Python ecosystem, and its `requirements.txt` format is a de facto standard for specifying dependencies. By converting other package management formats into this common format, the `PipExtractor` can leverage pip's native capabilities to resolve dependencies accurately. To achieve this, the `PipExtractor` executes the `pip install` command with the `--dry-run` and `--report` flags to generate a detailed report of the dependencies contained within the `requirements.txt` without actually installing them (see listing 8).

```
val reportFile = Files.createTempFile("pip-report-", ".json").toAbsolutePath()
val installProcess = runPipCommand(
  getPythonExecutable(workingDir, versionHint), workingDir.toAbsolutePath(),
  "install",
  "--dry-run",
  "--report",
  reportFile.toString(),
  "-r",
  input.toAbsolutePath().toString()
)
```

Listing 8: PipExtractor `runPipCommand()` implementation

The `-r` flag specifies the path to the `requirements.txt` file, while the `--report` flag generates a JSON report at the location specified using `reportFile.toString()` containing detailed information about the dependencies, including their versions, hashes, and other metadata.

The contents of this report are then parsed to extract the relevant dependency information, which is returned as an `ExtractionResult` object (see listing 9).

```
data class ExtractionResult(
  // Modules are subprojects
  val modules: Map<String, Module>,
  // Direct or transitive dependencies (Package objects contain PURLs)
  val packages: Map<String, Package> = emptyMap(),
  // Relationships between packages: creating a graph structure
  val dependencies: Map<String, Set<String>> = emptyMap(),
)
```

Listing 9: `ExtractionResult.kt` data class

The `ExtractionResult` object encapsulates the extracted dependencies, their relationships, and any additional data required for further analysis. It is returned back to the `AnalyzerService`, where it is merged with the results from other extractors to create a comprehensive view of the project's dependencies.

Python Version Detection

A critical problem during testing of *python-inspector* was the correct handling of Python versions, as different package managers and project configurations may specify different Python versions or constraints. Due to the version that ORT uses, *python-inspector* (version `0.12.0`) does not support Python 3.13 and above, which is a common version used in modern Python projects. Additionally, the *python-inspector* project appeared to be inactive, with numerous unresolved issues and pull requests accumulating without or little maintainer response. Given this maintenance status and the team's strategic goal of gradually moving away from ORT dependencies, creating a pull request to address the limitations was deemed impractical. This situation necessitated the development of a custom version detection mechanism that can handle multiple configuration sources and version specification formats.

The custom version detection mechanism is implemented in the `PythonVersionDetector` class, which is responsible for identifying the required Python version based on various project configuration files. The detector prioritizes configuration files based on their specificity and reliability, ensuring that the most accurate version information is used for analysis, e.g. lock files are preferred over manifest files like `pyproject.toml` or `Pipfile`.

The version detector uses a collection of regex patterns to find and extract version information from different files. Since each file type has its own unique format, no single pattern works for everything. `Pipfile` and `Pipfile.lock` files both use the `python_version` field, though `Pipfile.lock` requires quotes around the value because it follows JSON formatting rules. Poetry handles things differently - in its `pyproject.toml` file, it uses the `python` field, while the `poetry.lock` file stores this information in the `[metadata]` section under `python-versions`. uv takes yet another approach. Although it also uses `pyproject.toml` files like Poetry, it has its own specific format. Both uv's `uv.lock` and `pyproject.toml` files rely on the `requires-python` field to specify version information.

Every Extractor implementation (e.g., `PoetryExtractor`, `PipfileExtractor` and `UvExtractor`) is responsible for providing the necessary file names in the order of priority to scan, ensuring that the `PythonVersionDetector` can accurately determine the required Python version for the project as shown in listing 10.

```
    private fun parseLockFile(lockFile: Path): ExtractionResult {
        val pythonVersion = PythonVersionDetector.findPythonVersion(
          workingDir,
          listOf("poetry.lock", "pyproject.toml")
        )

        // convert poetry.lock to requirements.txt

        return pipExtractor.extractExactly(requirementsFile, pythonVersion)
```

Listing 10: PoetryExtractor `parseLockFile()` implementation with Python version detection

To achieve most accurate results the `PythonVersionDetector` is called from each extractor implementation, passing along the found version information to the `PipExtractor` (see listing 11 for method header) for final dependency extraction. This ensures that mutually incompatible standards for specifiying Python versions are handled correctly, and that the correct Python version is used for analysis tasks.

```
// This is called if the pip extractor is used directly
override fun extract(input: Path): ExtractionResult? {
  return extractFiles(input, listOf("requirements.txt"))
}

// This is called from other Python Extractors
fun extractExactly(
  input: Path, versionHint: String? = null
): ExtractionResult? = extractFiles(
  input, listOf(input.fileName.toString()), versionHint
)

// This is the actual implementation that runs pip install --dry-run --report
fun extractFiles(
  input: Path, fileNames: List<String>, versionHint: String? = null
): ExtractionResult? {
  // logic
}
```

Listing 11: PipExtractor actual `extract()` interface

The `parseRequirementsFile` method (like the `parseLockFile` found in listing 7) includes logic to handle the `versionHint` parameter. Running the `pip install --dry-run --report` command now looks like listing 12, where the `getPythonExecutable` method is used find the correct Python executable based on the detected version.

```
val installProcess = runPipCommand(
  getPythonExecutable(workingDir, versionHint),
  workingDir.toAbsolutePath(),
  ...
)
```

Listing 12: PipExtractor `runPipCommand()` with Python version handling

Under the hood the full command executed by the `runPipCommand`, with a Python version hint of `3.12`, would look like Listing 13, where the `python` command is replaced with the full path to the Python executable that matches the detected version.

```
.../libs/python-3.12/python install -r /path/to/requirements.txt
```

Listing 13: Example of executed pip install command with Python version

Python uses the same analyzer-libs Docker volume as Java, so the same script that downloads the Java runtimes, shown in listing 6, can be extended to download Python versions as well as newly supported package managers like uv[8]. The script is extended to include Python versions 3.9 through 3.13 which are the most commonly used versions in modern Python projects. Versions below 3.9 are not included as pre-built binaries are increasingly difficult to obtain due to the end-of-life status of these versions (*Status of Python Versions*, n.d.).

Overriding Python Version using scatool.yaml

For projects where the Python version cannot automatically be determined, the `scatool.yaml` configuration file allows users to specify a custom Python version. This is particularly useful for projects when the automatic detection fails due to unusual project structures or configurations as well as for pip projects as pip does not support versioning in its requirements files. The file has to be placed in the root directory of the project and contains a `python` section with a `version` field, as shown in listing 14.

```
# Python-specific configuration
python:
  # Specify the Python version to use (e.g., "3.9", "3.10", "3.11")
  # If not specified, the system default Python will be used
  version: "3.9"
```

Listing 14: Example of `scatool.yaml` configuration for Python version

As stated in listing 14, the `version` field allows users to specify the desired Python version, which will be used during analysis tasks. If not specified, the system default Python version will be used. The introduction of a `scatool.yaml` file allows for future extensions and customizations, such as specifying version overrides for other languages or package managers, making it a flexible configuration option for users.

[8] https://github.com/astral-sh/uv

The logic for reading and parsing the `scatool.yaml` file is implemented in a `AnalysisConfig` class. The Python-specific configuration is read within the `getPythonExecutable` method of the `PipExtractor` class, which was first introduced in listing 12. The settings within an `AnalysisConfig` object are prioritized over automatically detected versions, allowing users to override the default behavior when necessary.

Enhancing found Dependencies with Metadata

A critical aspect of supporting diverse dependency ecosystems lies in the ability to acquire comprehensive metadata beyond basic dependency information. The Python ecosystem demonstrates this challenge and solution particularly well, as packages distributed through the Python Package Index (PyPI) contain rich metadata that significantly enhances the quality of SCA analysis.

Python's centralized package distribution model through PyPI provides a unique advantage for metadata acquisition. The `PyPiResolver` implements an API integration that leverages PyPI's JSON API endpoints (`https://pypi.org/pypi/{package_name}/{version}/json`) to systematically retrieve detailed package information. This centralization enables comprehensive metadata queries that extend far beyond what can be extracted from local dependency files like `requirements.txt` or `pyproject.toml`.

The metadata service significantly enhances license detection capabilities by accessing PyPI's standardized license information.

```
val declaredLicense = pypiPackage?.info?.licenseExpression?.let {
    SpdxExpression.parse(it).normalize().toString()
}
```

Listing 15: License detection using PyPI metadata

This approach uses the `license_expression` field, which follows SPDX license expression specifications and provides standardized, machine-readable (see listing 15) license information. When no license information is available, the system defaults to "NOASSERTION" following SPDX conventions. While PyPI also provides `license` and `license_files` fields that could serve as additional fallback sources, the current implementation does not yet utilize these fields, representing a potential area for future enhancement to improve license detection coverage. This metadata-driven approach complements file-based license scanning, providing authoritative license information that may not be detectable through source code analysis alone, particularly when packages have missing license files in their source distributions.

Beyond license information, the PyPI metadata includes cryptographic checksums (SHA256, MD5, BLAKE2b) that enable integrity verification of downloaded artifacts. The system preferentially selects .tar.gz source distributions over wheel files to ensure complete source code availability for scanning, while maintaining checksum validation for future integrity checks.

The metadata acquisition integrates seamlessly with the `Analyzer` service through the `MetadataService` class, which orchestrates ecosystem-specific resolvers and combines their results with ORT analysis output. This creates `FullComponent` objects containing

both dependency relationship information and enriched metadata, providing users with complete visibility into their software supply chain dependencies and their associated legal and security implications. This implementation provides a blueprint for extending metadata acquisition to other ecosystems while allowing easy adaptation.

5.2 Scanner

The `Scanner`, first introduced in section 4.3, is a key component of the *SCA Tool* architecture, responsible for license and copyright analysis of software packages. It operates as a RabbitMQ consumer, processing tasks one after another. The `Scanner`'s primary function is to obtain the source code of a software package based on a PURL, analyze it using ScanCode, and provide the results in a structured format allowing for further processing and integration into the overall SCA pipeline.

The `Scanner` service implements an event-driven architecture pattern, where the `EventHandler` class serves as the central entry point for processing `ScanMessage`'s. When a scan request is received, the service creates a temporary directory and follows a structured workflow to ensure effective processing. Each scan task goes through the following steps:

1. **Cache Check:** First, it checks whether the package has been recently scanned to avoid redundant analyses
2. **Progress Notification:** The service notifies that scanning has begun, providing progress updates visible in the web UI
3. **Source Code Download:** The `DownloaderService` obtains the source code based on the PURLs and stores it in the temporary input directory
4. **File-Level Cache:** At the file level, it checks which files have already been analyzed to skip previously processed content
5. **ScanCode Analysis:** Only new or changed files are analyzed through ScanCode, with results generated in a structured format
6. **Result Upload:** The scan results are uploaded to MinIO storage for persistent access
7. **Completion Event:** A `ScanCompletedMessage` is sent via RabbitMQ to notify other services of the finished analysis
8. **Cleanup:** Temporary directories are removed to maintain system cleanliness, even in case of exceptions

This implementation enables efficient processing of large volumes of software packages while optimally utilizing system resources through intelligent caching at both the package and file levels.

5.2.1 Downloader

During the initial testing phase of *SCA Tool* (commit `dea3c420e5a7da9a5653647b097d9acb796ae03b`) it became clear that the existing downloader implementation was not sufficient to handle the diverse range of packaging formats. The original downloader yet lacked real-world robustness and extensibility, leading to frequent failures when processing packages from Java and Python ecosystems also due to poor data quality from the `Analyzer` service.

Before testing could start though, two profiles (configuration options) were introduced:

- `always-rescan`: This profile forces the downloader to always download the source code, even if it has been scanned before. This is useful for development and testing purposes, where changes to the source code or the scanning process need to be reflected immediately without relying on cached results.
- `keep-dirty`: This profile allows the downloader to keep the downloaded source code in a temporary directory after the scan is completed, enabling further inspection and debugging if needed. This is particularly useful during development and testing phases, where understanding the downloaded content is crucial for diagnosing issues.

The core challenge lies in the diversity of package ecosystems - each has its own conventions for versioning, repository hosting, and source code distribution. The downloader must interpret PURLs correctly and select the appropriate download mechanism while maintaining high reliability and data integrity.

The first change was introduced to the `ArchiveUtils`, which are responsible for extracting source code from various archive formats. Support for Maven artifacts was added, allowing the downloader to handle `-sources.jar` files which behave like ZIP files and contain the source code of Java packages. Though the quality and existence of these artifacts varies widely from repository to repository and package to package, they are often the only available source code for Java packages. Later revisions also include support Python `.whl` artifacts which also just like Maven artifacts are ZIP files containing the source code of Python packages.

The contents of these archives are not guaranteed to be complete, as they may not include all files necessary for a full analysis, such as `LICENSE.md`, `README.md`, or other development configurations. Having this in mind, this variant of the downloader is considered a fallback option, providing a basic level of source code retrieval when no other options are available. Further current limitations include lack of checksum verification and authentication support for private repositories.

5.2.2 Tiered Quality Downloading of Source Code

While PURLs provide a standardized format for package identification, the reality of package naming and versioning is far more complex. Not all PURLs contain version information, leading to "latest" scenarios, or version names may mismatch with actual versions in online registries, requiring sophisticated matching algorithms. This creates different quality levels where one can be confident that the downloaded code actually corresponds to the intended software package. These quality labels are crucial feedback

for the user to understand the reliability of the downloaded source code and its suitability for further analysis.

The system implements a hierarchical quality taxonomy based on the download priority:

1. **VCS Exact**: The highest quality level where one has precise repository information including exact revision/commit hash. This provides the most complete information and ensures *SCA Tool* was able to get the exact code that was used to build the package.
2. **VCS Best Guess**: When revision information is unavailable but a version is present, a version matching algorithm attempts to find the closest matching tag or branch in the remote repository.
3. **Source Artifact**: Direct download of source archives (ZIP, TAR.GZ) may contain the exact code used, but repository-specific files like LICENSE.md, README.md, or development configurations might be missing from the distributed archive.
4. **VCS Last Commit**: If VCS information is present and all other methods have failed, fallback to the latest available revision when version matching fails. This provides source code but with low confidence that it matches the intended package version. License and copyright information may be missing or outdated.

In later revisions of the downloader, these strict quality levels were replaced by a score-based system that assigns numerical quality scores (0-100 scale) to multiple download variants for each dependency (see table 3). The system creates concurrent download options: repository source URLs receive the maximum score of 100 points, while source artifact URLs are assigned 50 points. Additionally, fallback strategies are configured globally with `SOURCE_GUESS_REVISION` at 80 points, which attempts to infer the correct revision from version hints when primary download variants fail, and `SOURCE_USE_LATEST` for using the most recent repository revision as a last resort which is currently not used.

Download Strategy	Score	Description
Source URL	100	A dependency with a URL to the source code repository or archive
Artifact URL	50	Direct download to an artifact archive (e.g. Maven -sources.jar)
Fallback: Guess Revision	80	Used when primary URLs fail and revision must be guessed from version hints

Table 3: Download strategies and their quality scores

This multi-variant approach enables the `Scanner` to systematically attempt downloads in quality-descending order, starting with the highest-scored source and progressively falling back through lower-quality alternatives. Unlike the previous categorical system, the numerical scoring allows for algorithmic decision-making and quality threshold enforcement.

5.2.2.1 Matching Algorithms for Packaged Dependencies

The version matching problem represents one of the most complex challenges in the downloader implementation. Software packages are named and versioned according to inconsistent formats and conventions, as developers follow their own preferences rather than universal standards. The system must handle this diversity while maintaining high accuracy - ideally ensuring that matches have 99.99% confidence while still capturing as many packages as possible.

The version may not match 100% of the time, due to discrepancies in version naming conventions between the package and the repository. Tagged releases may use different formats, such as "v1.0.0", "release-1.0.0", or "v1_0_0", which can lead to mismatches even when the underlying code is the same. The downloader must handle these variations and ensure that the correct source code is retrieved. This level is less reliable than `VCS Exact`, as it relies on heuristics and may not always find a perfect match, but sophisticated matching provides high confidence.

A first draft implementation (seen in listing 16) of the matching algorithm is implemented in the `AbstractVcsDownloader` class, which serves as a base class for all VCS downloaders.

```
protected boolean tagCouldMatchVersion(String tag, String version) {
  // normalize inputs by trying ignore-case match
  tag = tag.trim().toLowerCase(Locale.ENGLISH);
  version = version.trim().toLowerCase(Locale.ENGLISH);
  if (tag.equals(version)) return true;

  // normalize separators
  tag = tag.replaceAll("[._ ]", "-");
  version = version.replaceAll("[._ ]", "-");

  // handle 'v' prefix variations
  if (tag.startsWith("v") && tag.substring(1).equals(version)) return true;
  if (version.startsWith("v") && version.substring(1).equals(tag)) return
true;

  return tag.equals(version);
}
```

Listing 16: Matching algorithm implementation in `AbstractVcsDownloader.java`

The algorithm is designed to handle various cases:

- **Case Normalization**: Both tag and version are converted to lowercase for case-insensitive matching.
- **Separator Normalization**: Dots, underscores, and spaces are replaced with dashes to create a consistent format.
- **Prefix Handling**: Common prefix like 'v' for version tags is handled
- **Exact Matching**: After normalization, a final exact string comparison is performed to determine a match.

Though this implementation provides a solid foundation, it is not exhaustive and does not cover all edge cases. The algorithm must balance precision (avoiding false positives) with recall (capturing legitimate matches). The current implementation prioritizes precision to maintain trust in the system while continuously expanding coverage through iterative improvements.

Further improvements could include:

- **Prefix Handling**: More sophisticated handling of common prefixes like 'release-', 'rel-' to improve matching accuracy
- **Suffix Handling**: Improved handling of version suffixes like '-beta', '-alpha' to enhance matching capabilities
- **Regular Expressions**: Using regex patterns to match complex version formats, especially for projects with unconventional versioning scheme
- **Fuzzy Matching**: Implementing algorithms like Levenshtein distance or Jaro-Winkler distance to handle minor variations in version strings
- **Configuration Options**: Allowing users to configure matching heuristics based on their specific needs or project conventions
- **Logging and Metrics**: Enhanced logging to capture matching attempts and outcomes, providing insights into algorithm performance and areas for improvement

Overall one has to strike a balance between precision and recall in the matching algorithm design. Every improvement should be carefully evaluated against the potential for false positives or negatives, ensuring that the system remains reliable and trustworthy while expanding its capabilities to handle a wider range of versioning conventions and formats. In future iterations of *SCA Tool* a form of user feedback mechanism or manual curation / review could be implemented to gather insights on matching accuracy and relevance as well as fix wrongly matched packages.

An evaluation of the matching algorithm's performance is available in chapter 6, where the algorithm is tested against a set of known package versions and their corresponding repository tags to measure its accuracy and effectiveness in real-world scenarios.

5.2.2.2 VCS Downloaders

As already mentioned in the sections above, the downloader not only supports direct downloads but also is able to retrieve source code from VCS's. Currently only Git is supported, though the architecture is designed to be extensible for other VCS systems like Mercurial or Subversion in the future if demand arises. The downloader adds specialized support for popular Git platforms like GitHub and GitLab, which provide extensive, well-documented, endpoints for repository metadata and source code retrieval. This allows the downloader to efficiently access repository information, including tags, branches, and commit history, without needing to clone the entire repository.

The downloader architecture uses a strategy pattern where specific downloaders are registered and selected based on applicability. The `DownloaderService` class (see listing 17) manages a list of registered VCS downloaders, which are initialized at

runtime. Each downloader implements the `VcsDownloader` interface, providing methods for downloading source code based on the `ScanMessage` containing scan URLs, fallback methods and an optional version hint.

```
@Service
public class DownloaderService {

  private final List<VcsDownloader> vcsDownloaders = new ArrayList<>();

  @PostConstruct
  public void init() {
    vcsDownloaders.add(new GitHubDownloader());
    vcsDownloaders.add(new GitLabDownloader());
    vcsDownloaders.add(new GitDownloader());
  }

  private DownloadResponse download(
    ScanMessage message,
    ScanMessage.ScanVariant variant,
    Path tempDir
  ) {...}

}
```

Listing 17: DownloaderService with registered VCS downloaders

The registration order in listing 17 implements a priority system where specialized platform downloaders are preferred over generic implementations, optimizing for both performance and feature richness. The `download()` method is responsible for selecting the appropriate downloader based on the `ScanMessage` and its variants.

It first bisects the `ScanMessage` to extract the relevant information about the package source such as VCS information and PURL, and then selects the appropriate downloader based on the parsed information. For packages that include VCS information in their qualifiers, the service gets a `VcsInfo` object containing the repository details. It then searches through the registered VCS downloaders to find the first one that declares itself applicable for the specific VCS information using the `isApplicable()` method shown in listing 18.

This approach allows for intelligent downloader selection where specialized implementations (such as the GitHub downloader referenced in listing 18) can handle platform-specific features and optimizations, while falling back to generic VCS downloaders when no specialized implementation is available. The system ensures that the most appropriate downloader is selected based on the source type and platform, maximizing both download efficiency and feature support for different package ecosystems.

Platform-Specific Downloaders

Large Git platforms like GitHub and GitLab receive native support through their URL based endpoints, which provide structured metadata and source code archives. These downloaders implement platform-specific logic to handle repository metadata retrieval,

tag processing, and source code downloading efficiently. This approach is relatively straightforward to implement and works reliably in most cases.

The GitHubDownloader implementation in listing 18 is a concrete example of a platform-specific downloader that extends the `AbstractVcsPlatform` class.

```
public final class GitHubDownloader extends AbstractVcsPlatform {

  List<Pattern> patterns = List.of(
    Pattern.compile(
      "https?://github\\.com/(?<owner>[^/]+)/(?<repository>[^/]+?)(?:\\.git)?"
    ),
    Pattern.compile(
      "ssh://git@github\\.com/(?<owner>[^/]+)/(?<repository>[^/]+?)(?:\
\.git)?"
    )
  );

  // Platform specific URL patterns
  String REVISION_URL = "https://github.com/%s/%s/archive/%s.tar.gz";
  String HEAD_URL = "https://github.com/%s/%s/archive/HEAD.tar.gz";
  String TAGS_URL = "https://api.github.com/repos/%s/%s/git/refs/tags";

  @Override
  public boolean isApplicable(VcsInfo info) {
    return parseRepositoryInfo(info.url()).isPresent();
  }

  // Further inherited methods
}
```

Listing 18: GitHubDownloader implementation

Further future enhancements could include support for GitHub's GraphQL API to retrieve repository metadata more efficiently, as well as handling private repositories through OAuth2 authentication. The downloader can also implement platform-specific rate limiting and retry strategies to handle API limits imposed by GitHub and GitLab.

Generic Git Downloader

Git serves as a universal protocol supporting virtually all platforms, making it an essential fallback mechanism. However, downloading the correct code revision is more complex and potentially less efficient than platform-specific APIs.

The main performance hit comes from the need to clone the entire repository history, which can be significantly larger than the source code archive. A shallow clone does not come with all revisions, may leave out tags and branches. This is problematic when trying match a version from a PURL to a specific commit or tag, as it may miss an existing revision that just hasn't been fetched yet.

Implementation wise the `GitDownloader` makes use of the `JGit` library, which provides a comprehensive API for interacting with Git repositories. The downloader simply has

to call `Git.cloneRepository().setURI(url).setDirectory(tempDir.toFile()).call();` for the initial clone operation. Once a repository is cloned, the downloader can use `repo.getRefDatabase().getRefs()` to retrieve all references, including branches and tags, and then resolve the requested revision based on the `ScanMessage` and its variants using the code in listing 16.

5.2.3 ScanCode Integration

The `Scanner` service integrates ScanCode for license and copyright detection through a flexible runner system that supports both local and containerized execution. The integration uses a strategy pattern with two implementations: `LocalScanRunner` for development environments with ScanCode installed locally, and `DockeredScanRunner` for production deployments using Docker containers.

ScanCode executes with optimized flags including `--only-findings` for reduced output size, `--processes` for parallel processing based on CPU cores, and a 300-second timeout per file. The system ignores cached files marked with `.scaignore` suffixes and outputs structured JSON results containing license detections, copyright statements, and file metadata.

The integration transforms ScanCode's output into structured `ScanResult` objects containing file-level license detections with confidence scores, copyright holders associated via line number correlation, and comprehensive metadata. Results are cached at both job and file levels using Redis, with SHA256 hashes preventing redundant scans of identical files across different packages.

This architecture enables language-agnostic license scanning while providing deployment flexibility and efficient caching for the broader SCA pipeline.

6 Evaluation

This chapter evaluates the implementation of *SCA Tool's* dependency ecosystem support capabilities against the requirements defined in chapter 3. The evaluation encompasses both existing functionality (F-01 through F-06) to verify no regression has occurred, and newly implemented features (F-07 through F-14) to assess their integration and effectiveness within the multi-ecosystem dependency analysis framework.

Testing was conducted across a repository of diverse software projects where possible, though not all features could be evaluated within the complete pipeline and were instead assessed as standalone components. While this approach has limitations, positive results and improvements in individual components should contribute to better overall pipeline performance across the supported ecosystems.

Rather than providing extensive empirical benchmarking data, this evaluation focuses on reasoned analysis of the implemented capabilities, documenting which requirements have been successfully fulfilled and identifying limitations or areas requiring future development. This approach acknowledges that while most targeted capabilities have been successfully added to the system, certain implementation caveats and incomplete features remain that warrant discussion for complete transparency regarding the current state of the extended *SCA Tool*.

6.1 Functional Requirements

The functional requirements defined in chapter 3 were iteratively refined during development as implementation challenges and ecosystem-specific complexities became apparent, reflecting the exploratory nature of extending SCA capabilities.

Existing SCA Tool Functionalities

- **F-01:** Version matching using PURL information to establish correspondence between package versions and remote artifacts/source code not only remains intact but has been strengthened with an enhanced matching algorithm that effectively handles version naming variations between package registries and source repositories. This requirement is **fulfilled**.
- **F-02:** This requirement has been **fulfilled**. Testing on **644** valid GitHub repositories shows the baseline algorithm (commit `dea3c420e5a7da9a5653647b097d9acb796ae03b`) achieved **55.4%** accuracy, while the initial thesis implementation (commit `59215405c03c9e69b94136fc2dc90fa51789fff0`) reached **55.7%**. Implementing the first two enhancements introduced in section 5.2.2.1 (further support for prefix/suffix handling) achieved **58.5%** accuracy, representing a **5.6%** relative improvement over baseline with **20** additional correct matches. Allthough the total number of unmatched cases remains high at **41.5%**, the improvements demonstrate the effectiveness of the matching algorithm in capturing a wider range of version naming conventions and formats. Future enhancements through regular expressions, fuzzy matching, and per-project configuration could potentially address the remaining 41.5% of unmatched cases, as discussed in section 5.2.2.1.

- **F-03:** Support for downloading source code from Git repositories before the thesis was constrained to GitHub and GitLab, but has been extended to support generic Git repositories using the `JGit` library. This allows the downloader to handle a wider range of Git platforms, though performance may vary based on repository size and structure. This requirement is considered **fulfilled**.
- **F-04:** This requirement continues to be **fulfilled** through the existing platform-specific optimizations for GitHub and GitLab repositories, with no regression observed during the implementation of extended ecosystem support.
- **F-05:** This requirement remains fully operational with existing transitive dependency resolution capabilities preserved across all supported ecosystems (JavaScript, Java & Python), ensuring no degradation in dependency tree analysis functionality.
- **F-06:** This requirement is implicitly **fulfilled** through the utilization of ScanCode by the `Scanner` service as the primary license and copyright analysis tool. The integration remains intact, with no regression observed in the ability to analyze license and copyright information across all supported ecosystems.

New Functionalities for Extended Ecosystem Support

- **F-07:** This requirement is considered **fulfilled**, as the new `JavaVersionDetector` and `OrtConfigWriter` classes with the analyzer libraries (JDKs in different versions) integrate seamlessly into the existing architecture and provide a more satisfactory analysis result compared to the vanilla ORT implementation.
- **F-08:** This requirement is **fulfilled** through the implementation of the `PipExtractor` class, which supports Python-specific dependency extraction from `requirements.txt` files. The extractor is capable of handling both direct dependencies and transitive dependencies, ensuring comprehensive analysis of Python projects and replaces the *python-inspector* component within ORT.
- **F-09, F-10 & F-12**: These requirements are **partially fulfilled** through the implementation of custom extractors for every package manager, which convert the respective package manager's dependency specifications into a `requirements.txt` format that is further processed by the `PipExtractor`. The requirement can only be considered partially fulfilled, as the current implementation does only regard the lock files and not use the manifest files, even though this is intended. A project that does not include a lock file within its project directory will not be analyzed correctly. A further limitation lies in the fact that uv's export functionality does not include custom index URLs in the generated `requirements.txt` format, making packages from private indexes unresolvable during the conversion process.
- **F-11:** This requirement is not **fulfilled**. Conda support has not been implemented because Conda focuses on binary packages and its own package ecosystem with pre-compiled binaries, while other Python package managers primarily use PyPI. Projects using Conda for dependency management cannot be analyzed correctly, though most Conda users maintain pip-based requirements files that can be processed through existing Python extractors.
- **F-13 & F-14:** These requirements are **fulfilled** through the addition of multiple scan URLs and fallback methods per `ScanMessage` within the `Analyzer`. The downloader

makes use of the `ScanMessage`'s by supporting multiple download variants for each dependency, allowing the system to attempt downloads in quality-descending order. This multi-variant approach enables the `Scanner` to systematically try different download options, starting with the highest-scored source and progressively falling back through lower-quality alternatives.

6.2 Non-Functional Requirements

The following evaluation summarizes the fulfillment of these requirements:

- **NF-01:** Performance considerations have been maintained with minimal conversion overhead in the `Analyzer` and unchanged pip resolution performance. The downloader's multiple download tiers enhance accuracy without significant performance degradation, while the primary bottleneck remains generic Git repository cloning, which is a pre-existing limitation affecting all ecosystems equally. This requirement is **fulfilled**.
- **NF-02:** The implemented Python ecosystem support demonstrates excellent maintainability through consistent architectural patterns and proper abstraction. New Python extractors follow established directory structures and interface implementations, with higher-level tools (Poetry, uv, Pipenv) elegantly delegating to `PipExtractor` to minimize code duplication. The Spring-native design with `@Component` annotations enables automatic discovery, making future ecosystem additions straightforward. While some architectural documentation gaps exist, the codebase's pattern consistency and clear separation of concerns provide a solid foundation for future development and ecosystem expansion making this requirement **fulfilled**.
- **NF-03:** The system implements comprehensive error handling including configurable timeouts (5-10 minute timeouts for processes, 3-5 second HTTP timeouts), retry mechanisms with exponential backoff for HTTP requests, and multi-URL fallback strategies for missing repositories. Python extractors handle malformed manifests gracefully, and the system continues analysis when individual dependencies fail. However, incomplete analysis results may require project rescanning in cases where critical dependencies cannot be resolved making this requirement **largely fulfilled**.
- **NF-04:** The system demonstrates fault tolerance through component-level error isolation, message queue retry mechanisms with dead letter handling, and partial result merging capabilities. Individual dependency resolution failures do not halt the entire analysis process, and the system can produce meaningful results from successfully processed components. However, projects with significant dependency resolution issues may produce incomplete analysis results requiring manual intervention or rescanning making this requirement **partially fulfilled**.
- **NF-05:** Checksum validation is implemented for critical system components (analyzer-libs service validates SHA256 checksums for Java/Python runtimes), and Git operations benefit from Git's native integrity mechanisms. However, the downloader service does not validate checksums for artifact downloads despite collecting checksum metadata from package registries, creating a security gap for non-Git downloads while relying solely on HTTPS transport security. This requirement is considered

not fulfilled as this is a security-critical aspect that must be addressed in future iterations of the system.

- **NF-06:** External communications properly use HTTPS for package registries (PyPI, npm) and APIs (GitHub, GitLab). However, internal service communications default to HTTP in development configurations, and critical security settings like SSH host key checking are disabled. Production security depends on proper environment configuration, as key service URLs are configurable with HTTP defaults that could compromise security if not properly set. This requirement is considered **largely fulfilled** as it is a configuration issue that is not inherent to the system design, but rather a deployment concern that must be addressed in production environments.

7 Conclusion

This chapter summarizes the key learnings, insights, and implications of the research conducted on supporting new dependency ecosystems in *SCA Tool*. The following sections present the primary learnings obtained, acknowledge the limitations encountered during the study, and outline potential directions for future developments in this domain.

7.1 Learnings

Throughout the development and evaluation of enhanced dependency ecosystem support in *SCA Tool*, several key insights emerged that contribute to our understanding of SCA's challenges and opportunities.

The thesis revealed fundamental patterns in how modern dependency ecosystems operate and highlighted the complexities inherent in extending *SCA Tool* capabilities across diverse package management systems. Most significantly, the thesis demonstrated that ecosystem-specific characteristics heavily influence the feasibility and accuracy of dependency detection approaches. Java's mature Maven repository infrastructure with structured metadata contrasts sharply with Python's fragmented package manager landscape, where inconsistent implementations and limited metadata quality standardization create substantial technical challenges.

The development of enhanced version matching algorithms yielded measurable improvements, with the third iteration of the algorithm achieving 58.5% accuracy compared to the 55.4% baseline — a 5.6% relative improvement representing 20 additional correct matches across 644 repositories. This validates the hypothesis that sophisticated prefix/suffix handling can meaningfully improve version correspondence accuracy, though the remaining 41.5% of unmatched cases indicates substantial room for future enhancement.

The modular architectural approach proved successful in integrating new ecosystem support while preserving existing functionality. The delegation pattern, where higher-level tools (Poetry, uv, Pipenv) convert their specifications to `requirements.txt` format for processing by `PipExtractor`, demonstrated an effective strategy for minimizing code duplication while accommodating diverse package manager interfaces. This architectural decision contributes valuable guidance for extending SCA tools to additional ecosystems.

7.2 Limitations

Several limitations must be acknowledged that constrain the generalizability and scope of the findings. These limitations primarily stem from resource constraints and methodological choices that were necessary to maintain the feasibility of this bachelor thesis project.

The evaluation framework employed in this study exhibits certain biases that may influence the conclusions drawn. The scope was intentionally limited to Java and Python ecosystems, representing only a fraction of the diverse landscape present in modern

polyglot development environments. This constraint was imposed due to the three-month time frame allocated for this thesis and the collaborative development context where multiple contributors work on *SCA Tool* simultaneously, occasionally limiting testing due to system bugs and integration issues inherent to ongoing development cycles.

The repository-of-repositories evaluation approach, while enabling systematic testing across representative projects, may not capture the full complexity of enterprise-scale deployments or edge cases present in less common dependency configurations. Additionally, the performance characteristics of the foundational *SCA Tool* components revealed scalability concerns when processing large repository collections, suggesting that the current approach may require optimization for enterprise-scale deployments.

The Python ecosystem investigation revealed fundamental challenges beyond simple implementation issues — inconsistent package manager interfaces and metadata availability represent inherent limitations that impact *SCA Tool* effectiveness regardless of implementation quality. These findings contribute to understanding ecosystem extensibility constraints rather than representing implementation failures.

7.3 Outlook on Future Developments

Building upon the foundations established in this thesis, several promising research directions emerge that could significantly advance the capabilities and scope of SCA tools.

Ecosystem Expansion and Scalability

The technical architecture and implementation analysis delivered through this thesis provide a foundation for significantly expanding ecosystem coverage by incorporating additional programming languages and package managers. Future efforts could build upon the modular detection components demonstrated in this work to support emerging ecosystems such as Rust (Cargo), Go (modules), or next-generation package management systems. The established delegation and modular extraction patterns offer a proven framework for integrating diverse package manager parsing strategies while preserving ecosystem-specific optimization opportunities.

Automated Testing Infrastructure

The evaluation framework and private test repository created during this research enable the development of comprehensive automated testing pipelines for individual *SCA Tool* components. Future work could establish systematic test case generation and validation using the repository-of-repositories approach, creating continuous integration workflows that automatically assess detection accuracy across diverse dependency scenarios. This automated infrastructure would facilitate rapid validation of new detection approaches and regression testing as the software development landscape continues evolving.

Advanced Matching and Community Resolution

Future research should focus on developing more sophisticated matching algorithms that could address the remaining 41.5% of unmatched dependency cases through techniques such as fuzzy string matching, machine learning-based pattern recognition, and regular expression optimization. Additionally, exploring community-driven dependency resolution mechanisms could leverage collective knowledge to validate and correct complex dependency relationships that automated systems struggle to resolve accurately. Such approaches could integrate expert human judgment with automated processing to achieve significantly higher accuracy rates.

Enterprise Integration Capabilities

Expanding *SCA Tool* capabilities to support enterprise environments requires implementing private repository access through SSH and OAuth authentication mechanisms. These enhancements would enable seamless integration with corporate development workflows while maintaining security standards required for proprietary codebases. Combined with the improved dependency detection capabilities demonstrated in this thesis, these enterprise features would position *SCA Tool* for broader adoption across diverse organizational software development environments.

References

About SPDX. Retrieved July 30, 2025, from https://spdx.dev/about/overview/

An Introduction to Package Managers. Retrieved July 29, 2025, from https://www.onyxgs.com/blog/introduction-package-managers

Analyzer | OSS Review Toolkit. Retrieved July 29, 2025, from https://oss-review-toolkit.github.io/ort/docs/tools/analyzer

Analyzing projects for dependencies (SCA). Retrieved July 29, 2025, from https://docs.sonarsource.com/sonarqube-server/latest/advanced-security/analyzing-projects-for-dependencies/

Authoritative Guide to SBOM - Implement and optimize use of Software Bill of Materials. OWASP Foundation. Retrieved July 30, 2025, from https://cyclonedx.org/guides/OWASP_CycloneDX-Authoritative-Guide-to-SBOM-en.pdf

Bals, F. (2024, March 17). *What is a software bill of materials?*. https://www.blackduck.com/blog/software-bill-of-materials-bom.html

Black Duck Binary Analysis. (2025, May 29). https://documentation.blackduck.com/bundle/bd-hub/page/Network_Communications/BDBA.html

Black, J. (2024, November 30). *Snippet Scanning, Explained*. https://fossa.com/blog/snippet-scanning-explained/

Build & package management concepts and terminology. Retrieved August 6, 2025, from https://pypackaging-native.github.io/meta-topics/build_steps_conceptual/#terminology

Cofano, S., Benedetti, G., & Dell'Amico, M. (2024, September 2). *SBOM Generation Tools in the Python Ecosystem: an In-Detail Analysis*. arXiv. https://doi.org/10.48550/arXiv.2409.01214

Cycode, T. (2024, April 23). *What Is Software Composition Analysis (SCA)?*. https://cycode.com/blog/what-is-software-composition-analysis-sca/

Dietrich, J., Rasheed, S., Jordan, A., & White, T. (2023, October 9). *On the Security Blind Spots of Software Composition Analysis*. arXiv. https://doi.org/10.48550/arXiv.2306.05534

Finkelstein, A. *Software Engineering Governance: a briefing*. https://www.cs.uoregon.edu/events/icse2009/images/postConf/TB-Governance-ICSE09.pdf

Gemmiti, S. (2020, October 21). *Things to consider when choosing a software composition analysis tool*. https://www.blackduck.com/blog/software-composition-analysis-tools.html

GitHub. (2024, July 29). *What is software composition analysis (SCA)?*. https://github.com/resources/articles/security/what-is-software-composition-analysis

Harnik, R. (2023, January 24). *Open Source Licensing Simplified: A Comparative Overview of Popular Licenses*. https://www.endorlabs.com/learn/open-source-licensing-simplified-a-comparative-overview-of-popular-licenses

Haynes, R. (2025, July 22). *How to Evaluate Endor Labs SCA for C/C++ Projects*. https://www.endorlabs.com/learn/how-to-evaluate-endor-labs-sca-for-c-c-projects

Home — ScanCode-Toolkit documentation. Retrieved July 29, 2025, from https://scancode-toolkit.readthedocs.io/en/latest/getting-started/home.html

How Snyk Container works. (2024, December 3). https://docs.snyk.io/scan-with-snyk/snyk-container/how-snyk-container-works

Infographic: Log4Shell Vulnerability Impact by the Numbers. (2022, March 18). https://blog.qualys.com/vulnerabilities-threat-research/2022/03/18/infographic-log4shell-vulnerability-impact-by-the-numbers

ISO 37301:2021(en), Compliance management systems — Requirements with guidance for use. Retrieved July 29, 2025, from https://www.iso.org/obp/ui/en/#iso:std:iso:37301:ed-1:v1:en

ISO/IEC 5962:2021 Information technology - SPDX® Specification V2.2.1. (2021, August). https://www.vde-verlag.de/iec-normen/250220/iso-iec-5962-2021.html

JSON API - PyPI Docs. Retrieved August 6, 2025, from https://docs.pypi.org/api/json/

Kula, R. G., Ouni, A., German, D. M., & Inoue, K. (2017, September 14). *On the Impact of Micro-Packages: An Empirical Study of the npm JavaScript Ecosystem*. arXiv. https://doi.org/10.48550/arXiv.1709.04638

License Detection Updates. Retrieved July 29, 2025, from https://github.com/aboutcode-org/scancode-toolkit/blob/develop/docs/source/reference/license-detection-reference.rst

Licenses. Retrieved July 29, 2025, from https://choosealicense.com/licenses/

Managing dependencies. Retrieved July 29, 2025, from https://go.dev/doc/modules/managing-dependencies

Neil, C. (2024, August 20). *Comparing SBOM Formats: Focus on Component Types in CycloneDX vs. SPDX*. https://sbomify.com/2024/08/20/sbom-component-types/

Nejati, M. *Understanding and Improving Code Review of Changes in Build Systems*. https://rebels.cs.uwaterloo.ca/papers/icse2025ds_nejati.pdf

Ombredanne, P. *Standardizing FOSS package identifiers using Package URL*. https://aboutcode.org/wp-content/uploads/2023-03-30-FOSS-purl.pdf

Open Source License Compliance. Retrieved July 29, 2025, from https://fossa.com/solutions/oss-license-compliance/

Oracle Java SE Support Roadmap. (2025, May 8). https://www.oracle.com/java/technologies/java-se-support-roadmap.html

oss-review-toolkit/ort. (2025, July 29). OSS Review Toolkit. https://github.com/oss-review-toolkit/ort

oss-review-toolkit/ort-ci-gitlab. (2025, March 7). OSS Review Toolkit. https://github.com/oss-review-toolkit/ort-ci-gitlab

Overview — ScanCode-Toolkit documentation. Retrieved July 29, 2025, from https://scancode-toolkit.readthedocs.io/en/stable/reference/overview.html

OWASP Dependency-Check. Retrieved July 29, 2025, from https://owasp.org/www-project-dependency-check/

package-lock.json. Retrieved August 6, 2025, from https://docs.npmjs.com/cli/v11/configuring-npm/package-lock-json

package-url/purl-spec. (2025, May 15). package-url. https://github.com/package-url/purl-spec

package.json. Retrieved August 6, 2025, from https://docs.npmjs.com/cli/v11/configuring-npm/package-json

Pfeiffer, R.-H. (2022, March 3). *License Incompatibilities in Software Ecosystems*. arXiv. https://doi.org/10.48550/arXiv.2203.01634

Plugin Architecture — ScanCode-Toolkit documentation. Retrieved July 29, 2025, from https://scancode-toolkit.readthedocs.io/en/stable/plugins/plugin_arch.html

Ramaswamy, Y. (2022). *Versioning Strategies and Dependency Management in Polyglot DevOps Pipelines*. *20*(12). https://www.neuroquantology.com/open-access/Versioning+Strategies+and+Dependency+Management+in+Polyglot+DevOps+Pipelines_14927

Ren, X., Ho, M., Ming, J., Lei, Y., & Li, L. (2021). *Unleashing the Hidden Power of Compiler Optimization on Binary Code Difference: An Empirical Study*.

Sarwar, M. W. (2024). *A Comprehensive Study on Software Bill of Materials Tools, Challenges and Adoption Barriers*.

SBOM as a legislative requirement. Retrieved July 28, 2025, from https://bell-sw.com/blog/u-s-and-eu-regulations-are-demanding-a-software-bill-of-materials-sbom/

SCA Tool. Retrieved August 4, 2025, from https://scatool.com/

scancode-toolkit/CHANGELOG.rst. Retrieved July 29, 2025, from https://github.com/aboutcode-org/scancode-toolkit/blob/develop/CHANGELOG.rst

SolarWinds Supply Chain Attack. Retrieved July 29, 2025, from https://www.fortinet.com/resources/cyberglossary/solarwinds-cyber-attack

Standardization Process. (2023, November 28). https://cyclonedx.org/participate/standardization-process/

Status of Python versions. Retrieved August 6, 2025, from https://devguide.python.org/versions/

Supported package manifests and package datafiles. Retrieved July 29, 2025, from https://scancode-toolkit.readthedocs.io/en/stable/reference/available_package_parsers.html

The Minimum Elements For a Software Bill of Materials (SBOM). (2021, July 12). The United States Department of Commerce. https://www.ntia.gov/blog/2021/ntia-releases-minimum-elements-software-bill-materials

Thirdparty Licenses Examples – License Maven Plugin. Retrieved August 6, 2025, from https://www.mojohaus.org/license-maven-plugin/examples/example-thirdparty.html

Using Resolution Rules. Retrieved July 29, 2025, from https://docs.gradle.org/current/userguide/resolution_rules.html

Vermeer, B. (2022, October 31). *How to create SBOMs in Java with Maven and Gradle*. https://snyk.io/blog/create-sboms-java-maven-gradle/

Wagner, M. (2023). JavaScript User Interface License Compliance Best Practices. *2023*. https://oss.cs.fau.de/wp-content/uploads/2023/06/Wagner_2023.pdf

Wang, K. (2018, November 29). *Cost/Benefit Analysis: Manual Audits vs Automated License Compliance*. https://fossa.com/blog/cost-benefit-analysis%E2%80%93manual-audits-vs-automated-license-compliance/

What is a package URL? - Amazon Inspector. Retrieved July 30, 2025, from https://docs.aws.amazon.com/inspector/latest/user/sbom-generator-purl-sbom.html

When license field in package.json doesn't match LICENSE. Retrieved July 29, 2025, from https://github.com/npm/feedback/discussions/709

Xia, B., Bi, T., Xing, Z., Lu, Q., & Zhu, L. (2023). An Empirical Study on Software Bill of Materials: Where We Stand and the Road Ahead. *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2630–2642. https://doi.org/10.1109/ICSE48619.2023.00219

Xu, W., Ye, H., Gao, K., & Zhou, M. (2025, July 19). *A first look at License Variants in the PyPI Ecosystem*. arXiv. https://doi.org/10.48550/arXiv.2507.14594