Data Curation in SCA Tool for Improved Software Composition Analysis Results

MASTER THESIS

Lukas Nehrke

Submitted on 11 October 2025

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

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

Declaration of Originality

I confirm that the submitted thesis is original work and was written by me without further assistance. Appropriate credit has been given where reference has been made to the work of others. The thesis was not examined before, nor has it been published. The submitted electronic version of the thesis matches the printed version. Where artificial intelligence tools were used in the preparation of this thesis, such use was undertaken with the explicit approval of my supervisors, and all AI-generated content has been reviewed, verified, and appropriately acknowledged.

Erlangen, 11 October 2025

License

Erlangen, 11 October 2025

Use of Artificial Intelligence

In the preparation of this thesis, large language models (LLMs) have been used for proofreading, improving clarity and readability of text, translations, code completion, and code review.

Abstract

Open source components dominate modern software applications, making license compliance a critical challenge for organizations. While automated processes in Software Composition Analysis (SCA) tools are essential for managing this complexity, they face inherent limitations in accuracy when detecting licenses and copyright statements, managing package metadata, and identifying dependencies. This master's thesis addresses these limitations by designing and implementing comprehensive data curation capabilities for SCA Tool, enabling human-in-the-loop workflows that combine automated analysis with manual review. The implementation introduces two primary curation workflows: package metadata correction and scanner finding curation. The metadata correction system allows users to override incomplete or incorrect package information at multiple levels. The scanner finding curation functionality provides an intuitive interface for reviewing and correcting license and copyright findings, with support for auditing, bulk operations, hotkeys, and automatic reuse of decisions across identical files. The resulting system enables organizations to achieve the accuracy required for license compliance by efficiently combining automated analysis with targeted human expertise.

Contents

List of Figures

Acronyms

API	Application Programming Interface
DBMS	Database Management System
DOS	Double Open Server
DTO	Data Transfer Object
HTTP	Hypertext Transfer Protocol
JVM	Java Virtual Machine
ORM	Object-Relational Mapping
ORT	OSS Review Toolkit
OSSRA	Open Source Security and Risk Analysis
PRD	Product Requirements Document
PURL	Package URL
REST	Representational State Transfer
RLS	Row Level Security
SCA	Software Composition Analysis
SBOM	Software Bill of Materials
SPDX	Software Package Data Exchange
SQL	Structured Query Language
SWHID	SoftWare Hash IDentifier
UI	User Interface
UML	Unified Modeling Language
URL	Uniform Resource Locator

UUID Universally Unique Identifier

VCS Version Control System

WAL Write-Ahead Log

1 Introduction

Open source components have become essential to modern software development. According to the recent OSSRA report by Black Duck Software, Inc. (2025), 97% of commercial codebases contain open source software, with the typical application incorporating approximately 911 such components. To avoid legal risks, organizations must maintain compliance with all applicable licenses for any software they distribute, including all of its transitive dependencies. Non-compliance can result in severe consequences, including intellectual property disputes, financial penalties, and reputational damage (Helmreich & Riehle, 2012).

Despite its importance, achieving license compliance remains a complex challenge. Automated tools have become essential for managing this complexity: scanning source code, extracting license information, and identifying copyright statements. However, even sophisticated scanners such as ScanCode Toolkit[1] cannot guarantee the accuracy needed for legal compliance (Wolter, 2019; Wolter et al., 2023). This accuracy gap necessitates human intervention to achieve the confidence level required for legal certainty. Manual review can identify missing dependencies, verify correct package metadata, disambiguate complex licensing scenarios, and make informed decisions about edge cases that require domain knowledge. However, the sheer volume of dependencies and scanner findings makes comprehensive manual review impractical without proper tooling support. Organizations need efficient workflows that combine the speed of automation with the accuracy of human judgment.

This thesis addresses this challenge by focusing on the human-in-the-loop aspect, defining *curation* as the manual process of reviewing and potentially correcting the results of automated processes in software composition analysis. Closely related, the term *license clearing* is used for examining all dependencies for a particular project and curating license and copyright findings from each dependency's source code, with the objective of enabling the generation of accurate Software Bill of Materials (SBOM) and third-party legal notices with correct licenses and attestations.

[1] ScanCode Toolkit - https://github.com/aboutcode-org/scancode-toolkit

The goal of this thesis is to enhance the existing SCA Tool application by introducing curation functionality. This addition enables users to manually correct incomplete or invalid data resulting from dependency graph generation, package metadata retrieval, and the detection of license and copyright findings. The key challenge is designing workflows that are intuitive and simple to learn, yet powerful enough to minimize the time spent on curation tasks, while maintaining the accuracy necessary for compliance.

This thesis is divided into seven chapters. Following this introduction, Chapter 2 discusses existing solutions in the field and offers a more in-depth look at SCA Tool. Chapter 3 specifies the functional and non-functional requirements for the system. Chapter 4 provides an overview of the system architecture and core concepts. Chapter 5 details the key design decisions and their subsequent implementation. Chapter 6 evaluates the implemented system against the previously established requirements. Finally, Chapter 7 summarizes the results and provides an outlook for future work.

2 Related Work

This chapter examines existing approaches and solutions in the field of Software Composition Analysis (SCA) and license compliance. Section 2.1 analyzes related tools that provide curation and license-clearing capabilities. Section 2.2 gives more insights about SCA Tool, the target platform for this thesis.

2.1 Related Tools

Three open source tools were selected for analysis based on their adoption in industry and relevance to the curation workflows being developed in this thesis. The focus is on their curation mechanisms, user workflows, and architectural decisions that inform the design choices made in this thesis.

2.1.1 OSS Review Toolkit

The OSS Review Toolkit (ORT)[1] is an open source compliance automation platform under the Linux Foundation that helps organizations systematically manage legal and security risks in their software supply chains. Notable adopters include leading companies such as Porsche, Volkswagen, Bosch, and ZEISS. ORT consists of multiple tools that are typically run together in a toolchain (The ORT Project Authors, n.d.):

- **Analyzer**: Resolves the dependency graph of a repository and retrieves metadata about the included packages.
- **Scanner**: Extracts license and copyright findings from the source code of project dependencies.
- **Advisor**: Identifies security vulnerabilities in included packages using external vulnerability databases.
- **Evaluator**: Applies customizable "Policy as Code" rules based on the previously gathered data to flag policy violations.

[1] OSS Review Toolkit - https://oss-review-toolkit.org/

- **Reporter**: Generates results in various formats, e.g. CycloneDX or SPDX SBOMs or customizable reports (HTML, Markdown, PDF).

Built in Kotlin, ORT follows a highly modular architecture that supports both programmatic use and command-line execution via standalone binaries. To simplify setup, Docker images bundled with common programming languages and package managers are available. A key strength of ORT is its extensible plugin architecture. ORT already ships with numerous plugins for tasks such as dependency analysis, source code scanning, vulnerability advisories, and report generation, allowing organizations to adapt ORT specifically for their needs.

The ORT-Server[2] is a related project, which extends ORT as a scalable service. It transforms the traditionally command-line-based ORT tools into a cloud-deployable service with a web interface and REST API endpoints. It is currently in incubation phase at the Eclipse Foundation.

Package Curations

Package Curations enable users to correct or enhance package metadata. These curations are applied during the analyzer step of the ORT pipeline and can be specified by providing YAML files via the command line when running the binary.

Figure 2.1 illustrates a typical curation example in which the description and Version Control System (VCS) information for a package is corrected. Rather than targeting individual packages only, curations also support Apache Ivy-based pattern matching for version specifications, which can significantly reduce duplication and simplify maintenance.

```yaml
- id: "Maven:foo:bar:1.0.0"
  curations:
    comment: "Example package curation."
    description: "This is a curated description metadata entry."
    vcs:
      type: "Git"
      url: "https://example.com/repositories/foobar.git"
      revision: "e0a3880087aee1286a1ab90055de7b38cbc0f6cf"
      path: "lib/foobar"
```

Figure 2.1: Example package curation in YAML format.

[2]ORT Server - https://eclipse-apoapsis.github.io/ort-server/

A large set of metadata entries can be modified, such as the Package URL (PURL), homepage, description, or source code location of the package. In addition, a `concluded_license` can be specified that overrides the license detected by source code scanners.

Beyond file-based curations, ORT supports integration with other package curation providers. ORT ships built-in plugins for ClearlyDefined[3] and SW360[4], allowing seamless integration into these tools.

Package Configurations

Package Configurations allow users to correct and exclude license findings. Like package curations, they can be defined using YAML files. Figure 2.2 shows an example of a package configuration that excludes test files using a regular expression and corrects the license classification of a license finding present in the `LICENSE` file of a package.

```yaml
package_configurations:
  - id: "NPM::foobar:1.0.0"
    source_code_origin: VCS
    path_excludes:
      - pattern: "test/**"
        reason: "TEST_OF"
    license_finding_curations:
      - path: "LICENSE"
        start_lines: "4"
        reason: "DOCUMENTATION_OF"
        concluded_license: "MIT-0"
```

Figure 2.2: Example package configuration in YAML format.

False-positive license findings can be rejected by setting the `concluded_license` to `NONE`, preventing their propagation to subsequent pipeline stages. For false-negatives, however, adding additional findings is not supported.

Because license headers often generate many similar findings across a codebase, a single license finding curation can cover them by glob-matching file paths. When needed, multiple start lines can be listed to target specific occurrences.

While copyright findings cannot yet be corrected in the same manner as license findings, users can define lists of copyright texts or regular expressions to filter out unwanted content during the evaluator and reporter stages (*copyright garbage*). Additional authors can also be included through package curations.

[3]ClearlyDefined - https://clearlydefined.io/
[4]SW360 - https://eclipse.dev/sw360/

Since manual curation through YAML file editing can be cumbersome, Double Open Server (DOS) provides a graphical UI for more convenient creation and management of finding curations.

2.1.2 Double Open Server

Double Open Server (DOS)[5] is an open source companion server for the OSS Review Toolkit (ORT) with focus on license compliance. Governed by Double Open Oy[6], it integrates ScanCode Toolkit to identify license findings in software components and provides a user-friendly interface for reviewing the results. The software is still in early development and lacks comprehensive documentation; therefore, workflows were analyzed primarily from source code. All curation workflows were verified with ORT 66.0.3 and the DOS development environment at commit 0d7775e.

DOS integrates into the ORT toolchain through package configuration and scanner plugins, which ORT ships by default. During the scanner stage, the scanner plugin uploads all package sources to the DOS backend, which schedules a ScanCode worker to scan the provided sources. Scan results from previous invocations are reused, making the process highly efficient.

Once scanning completes, users can review the results in the Clearance UI. This interface displays all source files for a package and allows users to create *license conclusions*, which assign SPDX license expressions to specific source files. To support decision-making, the UI displays both declared and discovered license information for each file. Users can also bulk-create license conclusions using glob patterns to match file paths. Additionally, *path excludes* allow users to ignore files irrelevant to distribution by specifying glob patterns and selecting appropriate exclusion reasons (e.g., documentation or test files).

By default, both license conclusions and path excludes are shared across packages for identical files based on SHA256 content hashes, significantly reducing curation effort when processing multiple versions or similar packages. Users can alternatively mark decisions as *local*, restricting them to affect only the current package.

The package configuration provider requests path excludes and license conclusions from DOS to apply them in the evaluator and reporter stages. Similar to ORTs limitations, the curation of copyright findings is currently unsupported.

[5]Double Open Server - https://github.com/doubleopen-project/dos
[6]Double Open - https://doubleopen.io/

2.1.3 FOSSology

FOSSology is a toolkit for license compliance supported by the Linux Foundation. It was initially developed by HP and released as an open source project in 2007 (Gobeille, 2008). At its core, FOSSology provides tools to scan codebases for license, copyright and export control (ECC) information, presents the results for manual review and allows the data to be exported in various formats such as SPDX, CSV or HTML. For advanced workflows, FOSSology provides a comprehensive REST API allowing an integration in other tools such as SW360.

In a basic license-clearing workflow, the user first uploads the source code of a dependency to FOSSology, either directly via the UI or via automated tooling. The code is then analyzed using a set of *agents*. Once finished, the user can view the results file by file, showing the source code of the file with findings visually highlighted. The user can delete, modify or manually add missing license findings to a file. The clearing process is complete once all files containing license findings have been reviewed.

Bulk Recognition in FOSSology applies license decisions to all files containing a specified text fragment. This feature is particularly useful for license headers, where many source files require the same license classification. Rather than processing files individually, a single bulk operation can be executed to apply the decision across all matching files.

FOSSology offers several solutions for reusing previous decisions. Similar to DOS, it enables clearing decisions to be automatically applied to future occurrences of files with identical contents through the *Clearance Scope* feature. For more advanced reuse capabilities, users can configure the *Reuser* agent, which allows copying selected decisions from a previous upload.

Although the application allows the removal and additions of licenses and copyrights, the workflow is still not optimized for crowd-sourcing and scanner improvement. License findings with the same classification are aggregated on file-level into a single suggestion, making it hard to trace decisions back to the original source. The issue is even clearer with copyright findings, which are aggregated on upload-level. It is also not possible to provide positional information when editing existing or creating new findings.

2.2 SCA Tool

SCA Tool is a cloud-based solution for Software Composition Analysis (SCA) and license compliance. The software is developed at the Professorship of Open Source Software[7] at the University of Erlangen-Nuremberg, with contributions from the core SCA Team and university students engaged in development, design, and research activities. The tool is currently in private beta while gaining feedback from users. SCA Tool offers four core solutions:

- **SBOM Management**: This solution allows users to generate and maintain a Software Bill of Materials (SBOM), providing a complete inventory of used components and their relationships.
- **Open Source Governance**: This solution enforces user-defined policies on acceptable open source usage. By monitoring SBOM data, SCA Tool notifies users when unwanted licenses are detected.
- **License Compliance**: SCA Tool ensures that products comply with the legal obligations of open source licenses by enabling the generation of legal notices with all necessary attestations.
- **Vulnerability Management**: SCA Tool continuously scans components for security vulnerabilities. When new issues arise, SCA Tool alerts users, communicates risks and provides remediation guidelines.

By providing an all-in-one solution with zero setup, SCA Tool simplifies workflows, eliminating the complexity of tool integration, reducing context switching between applications, and providing teams with a single source of truth for all their software composition analysis needs. Another important strength comes from SCA Tool's multi-tenant approach. This not only allows efficient reuse of data storage and compute, but also allows sharing insights such as vulnerability exposure or data curations between users.

2.2.1 Limitations

The implementation of these four solutions requires extensive automation. However, these automated processes face significant limitations that can result in incomplete or incorrect results:

1. **Dependency Graph Generation**: The internal SBOM represents the most critical data structure, as all four core solutions depend on its accuracy. Dependencies and their relationships may be absent from the generated graph in several scenarios: when package managers are unsupported,

[7] https://oss.cs.fau.de/

when dependencies embedded directly in the source code, or when they are downloaded during build or runtime.

2. **Package Metadata Retrieval**: Package metadata is directly resolved from various package registries. This metadata contains essential information such as source code locations, which are required for subsequent license and copyright extraction. However, this metadata is frequently incomplete, incorrect, or in case of private packages entirely absent.
3. **License and Copyright Detection**: For license scanning, SCA Tool utilizes ScanCode Toolkit, which provides extensive license detection rules and a comprehensive license database. Despite its capabilities, the scanning process faces fundamental challenges:
   (a) License and copyright texts often appear in modified, incomplete, or non-standard formats, leading to potential misidentifications.
   (b) License declarations may reference external resources such as websites or file paths that the scanner cannot sufficiently resolve.
   (c) The entire source of a package is analyzed without distinguishing between files intended for distribution and those for build, test or documentation.
   (d) Natural language ambiguities and complex licensing constructs such as dual licensing or license exceptions usually require human interpretation.
4. **Vulnerability Detection**: Vulnerability detection suffers from high false-positive rate. The cross-referencing process relies solely on package identifiers without considering which specific code paths or symbols are actually used by the application. Since SCA Tool already provides basic curation capabilities by flagging vulnerabilities as false-positive, this area falls outside the scope of this thesis.

For the first three categories of limitations, SCA Tool currently lacks any mechanism for human intervention. Users must either accept potentially incomplete or incorrect results, or resort to external manual processes or different tools. Therefore, this thesis aims to close this gap by designing and implementing a curation system that enables users to correct and supplement these automated analysis results directly within SCA Tool.

3 Requirements

This chapter defines the software requirements, covering the requirements analysis methodology and user personas (Section 3.1), functional requirements (Section 3.2), and non-functional requirements (Section 3.3).

3.1 Requirements Analysis

Requirements were derived based on three primary sources:

- **Existing requirements**: The Product Requirements Document (PRD) of SCA Tool defined high-level functional requirements prior to this thesis.
- **Reverse-engineering of related tools**: The workflows of FOSSology, Double Open Server, OpossumUI, and the OSS Review Toolkit were analyzed to identify standard features and opportunities for innovation.
- **Domain knowledge from stakeholders**: Through weekly supervisor meetings and feedback from SCA Team coworkers, requirements were refined and prioritized throughout the thesis period.

All requirements were collectively evaluated at the end of the implementation. The comprehensive results of this evaluation are presented in Chapter 6.

3.1.1 Proto-Personas

Four proto-personas were developed to represent typical user profiles and usage patterns in license compliance workflows. While not based on empirical research, these personas synthesize insights from above requirement sources to capture key user needs and priorities.

> **Persona A** is a solo developer building a commercial smartphone game. They want to generate third-party legal notices to ship with the app and prefer to rely entirely on automated findings. Although they understand they are legally responsible for their software, they

consider extensive manual reviews disproportionate to the perceived risk and available time and budget.

Implications: manual curation should be optional and not disrupt existing workflows; improved automation can be more valuable than extensive curation features

Persona B is part of a small software startup with only technical founders. Due diligence requirements suddenly made license compliance critical. B needs a tool to quickly identify and resolve potential legal issues in their software. Because they need to move fast and lack in-house expertise, they are considering paying an external company to perform license clearing for them.

Implications: intuitive and easy-to-learn workflows; offer filters to focus on a subset of packages/files; support for external license-clearing services

Persona C is a Tier-1 automotive supplier whose OEM customers require ISO/IEC 5230 (OpenChain) conformance. Their current open source maturity is low: inconsistent tooling is used, SBOM quality varies, and no centralized compliance process is in place. C wants to advance their processes by using a single tool, combining automated processes with manual reviews, to meet the standard's requirements and deliver auditable compliance evidence. Additionally, C wants to reuse license-clearing decisions across multiple products, and doesn't mind sharing this data with other companies.

Implications: license-clearing workflows with decision reuse across multiple projects/products; strong audit trails and evidence retention; support for cross-organization data sharing

Persona D is responsible for the open source usage of a large software company. D needs their products to be legally compliant by ensuring all third-party code is permissively licensed and providing SBOM and legal notices with correct attestations. D is already using FOSSology for license clearing, but wants a more integrated tool with more efficient workflows. While developers are responsible for tracking components and providing correct source code locations, license-clearing workflows are performed by a dedicated clearance team.

Implications: At least feature parity with FOSSology; migration/import for existing clearing data; clear separation of roles (developer vs. curator); support for many projects with potentially large codebases; strong data isolation

3.2 Functional Requirements

Functional requirements define the behaviors, functions, and services that the software must provide. All requirements follow the sentence templates described by SOPHIST GmbH (2016) and include unique identifiers for reference throughout this work. The requirements are presented in an arbitrary order that does not reflect their priority.

3.2.1 Curation of Dependency Graphs

F-01: The system should provide the user with the ability to filter dependencies by including or excluding them based on package names and package manager scopes.

F-02: The system should provide the user with the ability to add an additional dependency by entering its Package URL and, optionally, package metadata.

F-03: If a user adds a dependency without providing package metadata, the system should attempt to resolve the metadata from the package registry.

3.2.2 Curation of Package Metadata

F-04: The system shall provide users with the ability to update package metadata for a specific package within a single project version.

F-05: The system shall provide users with the ability to update package metadata for a specific package across all projects in an organization.

F-06: The system shall provide users with the ability to override the source code location of a package by specifying a public URL to the source repository or archive.

F-07: The system should provide users with the ability to override the source code of a package by uploading a zip file.

F-08: As soon as a user updates the source code of a package, the system shall schedule a scan for the new source code.

3.2.3 Curation of Scanner Findings

F-09: The system shall provide users with the ability to accept, reject or modify scanner findings.

F-10: The system shall provide users with the ability to add additional copyright and license findings for a source file.

F-11: If two or more codebases reference an identical file, the system shall store the associated curation data only once and make it available to all affected codebases.

F-12: If a finding is accepted, added or modified, the system shall include the curated version in SBOM files and legal notices.

F-13: If a finding is rejected, the system shall not include it in SBOM files and legal notices.

File Reviews

F-14: The system should provide users with the ability to exclude directories or individual files within a codebase.

F-15: The system should provide users with the ability to explicitly skip an entire codebase, specific directories, or individual files.

F-16: The system should provide users with the ability to exclude or explicitly skip files based on regular expressions that match file paths.

F-17: The system should provide users with the ability to explicitly skip all files that do not contain scanner findings.

F-18: As soon as a user completes the review of a file, the system shall update the discovered license for that file and all codebases containing the file.

License-Clearing UI

F-19: As soon as the user opens a source file for review, the system shall display all detected copyright and license findings with their metadata in a structured list format.

F-20: The system shall provide users with the ability to navigate between packages, files, and findings.

F-21: The system shall provide users with the ability to view curated findings and base findings in a single merged view overlaid on the source file.

F-22: The system should provide users with the ability to view curated findings and base findings in two different views (split view).

F-23: As soon as the user clicks on the highlighted finding text in the source code, the system shall display the finding's details.

F-24: The system should provide users with the ability to configure a finding's position by selecting text in the source code.

3.3 Non-Functional Requirements

Non-functional requirements establish boundary conditions for the previously described functionality of the software. These are subdivided below according to the main characteristics of the ISO/IEC 25010:2023 specification.

3.3.1 Security

NF-01: The system shall record the user ID, timestamp, and specific changes made for every operation related to curation.

NF-02: The system shall ensure that only users with the appropriate permissions can curate data.

NF-03: The system shall ensure that curation data is only accessible by the tenant that owns the data.

3.3.2 Reliability

NF-04: The system shall have at least one fallback provider for providing source code.

NF-05: The system shall recover curation data with a maximum data loss of 30 minutes in case of database failure.

3.3.3 Interaction Capability

NF-06: The visual consistency of the UI should be maintained by reusing components and styles from the existing application.

NF-07: The UI should allow users to execute all core actions using at most one mouse click.

NF-08: The system shall require confirmation for all destructive operations with a clear warning message.

NF-09: The preview functionality for bulk operations shall display all affected items before execution.

NF-10: The system shall provide loading indicators for all operations expected to be longer than one second.

NF-11: The color scheme consistency of the UI shall be maintained across all curation related components.

NF-12: The error messages of the system shall include meaningful descriptions and suggested corrections.

NF-13: The UI shall provide meaningful tooltips for all buttons and links that just use an icon.

3.3.4 Compatibility

NF-14: The system shall reuse the technologies, communication patterns and database infrastructure of the existing services.

3.3.5 Maintainability

NF-15: The curation logic should use flexible algorithms that can be improved in future versions without requiring complex data migrations.

3.3.6 Performance Efficiency

NF-16: The system should be able to handle at least 100 concurrent user sessions without degradation in performance.

4 Architecture

This chapter outlines the system's overall architecture and key concepts. Section 4.1 examines the existing SCA Tool architecture and its core components. Section 4.2 describes the architectural modifications needed for the scanner service. Section 4.3 presents the role of the new curation module. Finally, Section 4.4 explains the intended license-clearing workflow.

4.1 Existing Architecture

In order to better understand the architectural changes introduced in this thesis, this chapter briefly highlights the most relevant architectural decisions and styles of the existing SCA Tool application.

4.1.1 Logical Structure

SCA Tool uses a modern web development stack where the frontend and backend are independently deployed applications that communicate over a RESTful interface. The frontend uses the Next.js[1] framework, which extends React with performant routing, caching, and server-side rendering capabilities. API conformance is established through an API-first approach. Developers maintain an OpenAPI specification YAML file, which is used to generate API clients and server stubs.

The backend uses the Spring Boot[2] framework and follows an architectural style known as a Modular Monolith. While traditional monoliths organize code in layers (e.g., presentation, business, data), modular monoliths organize code into domain-specific modules (e.g., inventory, orders, billing). This architecture is implemented using Spring Modulith[3], which ensures that internal symbols remain hidden from other modules, prohibits cyclic dependencies, and provides utilities for asynchronous communication between modules.

[1]Next.js - https://github.com/vercel/next.js
[2]Spring Boot - https://github.com/spring-projects/spring-boot
[3]Spring Modulith - https://github.com/spring-projects/spring-modulith

Figure 4.1 presents the logical composition of the SCA Tool backend relevant to this thesis:

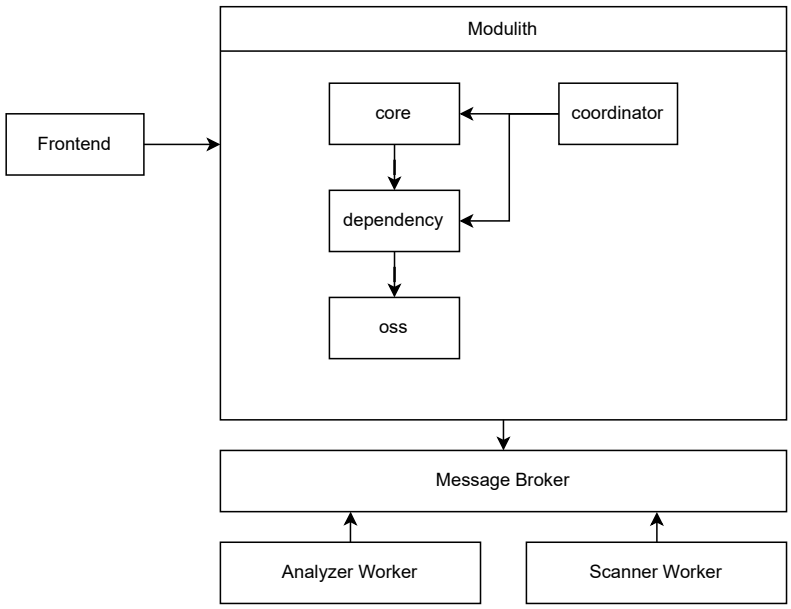

Figure 4.1: Logical structure (simplified) of the SCA Tool backend prior to this thesis.

- **core**: Manages projects and project versions. A project typically represents a version-controlled codebase of a user, while a project version represents a specific state of this codebase in time, typically tracked with a release name.
- **dependency**: Manages the internal SBOM data structure of a project version. This includes which packages are used, their scopes (build, test, development, etc.), and the relationships between them.
- **oss**: Manages package-related data independent of any project. This includes package metadata, license and copyright findings, and security advisories. This data is mostly public and shared between all SCA Tool tenants.
- **Analyzer Worker and Scanner Worker**: Two microservices communicating asynchronously with the modulith via a message broker. Both services analyze codebases as part of a project analysis.

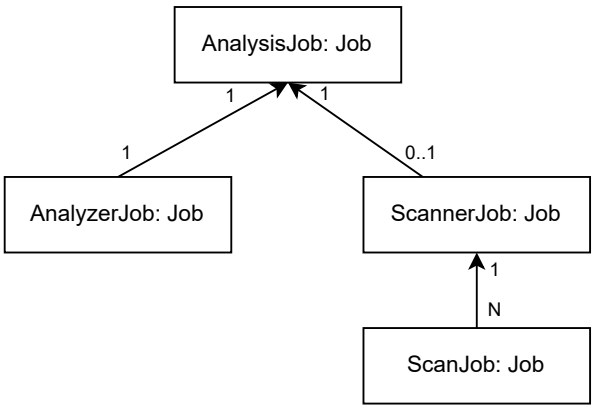

Figure 4.2: Hierarchy of project analysis jobs.

4.1.2 Analysis Pipeline

When a user pushes a new version of a project to SCA Tool, a project analysis job is enqueued. Jobs are managed by the **coordinator** module and organized hierarchically, with independent jobs running in parallel. The `AnalysisJob` represents the entire pipeline of one project analysis and consists of the `AnalyzerJob` and the `ScannerJob` as its direct children.

The analyzer job extracts the dependency graph from the user's codebase, which is typically provided via a Git repository location. It uses the analyzer module of the OSS Review Toolkit (ORT) to perform the extraction, falling back to a custom implementation where ORT has limited support. The process typically completes within seconds to a few minutes, returning the data to the modulith.

The scanner job is enqueued after the analyzer job succeeds. This job bundles a scan job for every unscanned package detected by the analyzer. By bundling these jobs and waiting for a free scan queue, the system ensures fairness across multiple tenants. Once a queue becomes available, the scanner job is unpacked and all individual scan jobs are sent to that queue.

Scan workers pull jobs from all scan queues in a round-robin fashion. The actual scanning is performed by invoking ScanCode Toolkit on the downloaded files. Once completed, the data is sent to the modulith and persisted. For projects with many dependencies, the scan process can take multiple hours. SCA Tool addresses this through a large worker pool, job deduplication, and a multi-tiered cache of scan results.

4.2 Scanner Adaptations

The scanning architecture was modified to support package metadata corrections. This was necessary due to several issues with the existing interfaces:

1. The scanner worker received package data as input and attached all findings directly to that package. When a package was rescanned due to a user's metadata correction, the results would overwrite the data for all tenants.
2. Both the input and resulting scan data were treated as public, making it impossible to support direct source code uploads.
3. The scanner worker relied on fallback mechanisms. For example, when unable to resolve a specified VCS revision, it automatically scanned the latest revision instead. However, this behavior is inappropriate when users have explicitly specified the source location as part of a metadata correction.

While issue (3) was fixable by making the applied fallbacks configurable, issues (1) and (2) required greater adaptation of the exposed interfaces.

4.2.1 Scan Identifiers

To address problem (1), scanner responsibilities were reduced from scanning packages to scanning directories. Instead of providing a Package URL when requesting a scan, clients now provide instructions for accessing the target directory. These instructions are modeled by the `ScanInput` class, which has three variants for different source code locations (Figure 4.3):

- `VcsScanInput`: References a public repository under version control. Currently, only Git repositories are supported.
- `ArchiveScanInput`: Downloads public file archives (.zip, .tar, etc.) or public repositories from GitHub and GitLab.
- `LocalScanInput`: Specifies a path to a local directory. Used exclusively for testing and debugging.

While this model effectively represents inputs, it is poorly suited for indexing scan results. The problem is that multiple scan inputs can reference the same source code, and a scan input's contents may change over time. This leads to redundant rescanning and duplicated storage of scan data.

Codebase identifiers are the solution to this problem, which are represented by SoftWare Hash IDentifiers (SWHIDs) as defined in ISO/IEC 18670:2025. Before scanning, the system calculates a `swhid:1:dir:` hash for the downloaded contents. This hash is then used to skip already-processed jobs and index the scan

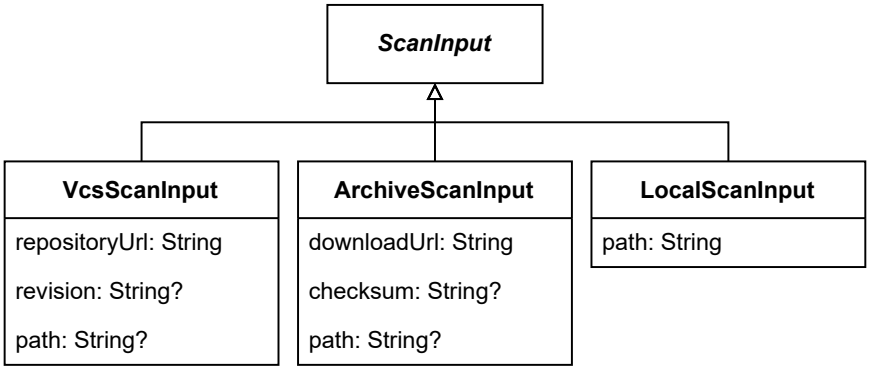

Figure 4.3: Class diagram visualizing different types of scan inputs.

results. By maintaining a mapping table between scan input values and codebase identifiers, either values can be used to retrieve the scan results.

4.2.2 Multi-tenancy

To allow scanning of private resources like the user's codebase or direct source code uploads, SCA Tool requires a sophisticated multi-tenancy concept to correctly isolate the data of multiple tenants. The insights from migrating the scanner to a multi-tenant service can later be applied to other parts of the application. Based on Beardsley (2010), three common architectural approaches for implementing a multi-tenant system can be implemented when using PostgreSQL as DBMS:

1. **Database per tenant**: Each customer gets their own dedicated database. While this offers highest isolation of the customer's data, it also has the highest complexity and maintenance cost.
2. **Schema per tenant**: Uses a single database but each customer gets their own schema/tables. This approach trades some isolation with lower costs due to resource-sharing, but still requires a complicated setup at application level.
3. **Discriminative field**: All customers share the same database and tables with data separated only by a partition key. This simplifies management but may not meet strict isolation requirements of customers.

For the scanner architecture, the third option with discriminative fields was implemented, as this is the only feasible approach with a small team and a large number of tenants. This solution also enables the system to easily return both public and private data within single SQL queries.

All exposed interface methods now require a `tenant_id` parameter, and the system strictly enforces that only data from the specified tenant or public data will be returned. The 'Nil' UUID (all bits set to zero) serves as a special identifier to designate public data.

To provide an additional layer of security, PostgreSQL Row Level Security (RLS) policies were configured. These policies act as a safeguard at the database level, ensuring that no data from other tenants can be accessed or modified, even if application-level queries are misconfigured.

4.3 Curation Module

To implement the main requirements of this thesis, the **curation** module was introduced to the modulith. This module acts as a middleware between the dependency and oss modules, as illustrated in Figure 4.4. Previously, the dependency module queried the oss module directly for package metadata and scanner findings. With the new architecture, the dependency module requests this data from the curation module instead, which retrieves the information from the oss module, applies any relevant corrections, and returns the modified results.

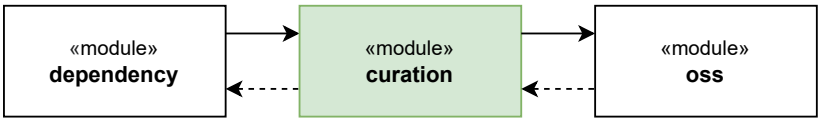

Figure 4.4: Flow of package data with the curation module as an intermediate layer for applying corrections.

The curation module exposes its functionality through multiple interfaces:

- `MetadataCurationService`: Manages the creation, modification and deletion of package metadata corrections (Section 4.3.1).
- `FindingCurationService`: Manages the creation, modification and deletion of finding curations (Section 4.3.2).
- `FileCompletionService`: Handles the creation and deletion of file completions (Section 4.3.3) and provides methods for calculating the package clearance progress.
- `CuratedPackageService`: Proxies the `PackageService` interface of the oss module to return package metadata with corrections applied.
- `CuratedFindingService`: Proxies the `FindingService` interface of the oss module to return license and copyright findings with corrections applied.

Similarly to the other modules, the implementation of these services resides in the `internal` package nested within the curation module. The internal structure of the curation module is described in detail in Chapter 5.

4.3.1 Package Metadata

As specified by requirements F-4 and F-5, metadata corrections can be applied at the organization level and at the project version level. The more specific level inherits unset fields from the more general level, as shown in Figure 4.5.

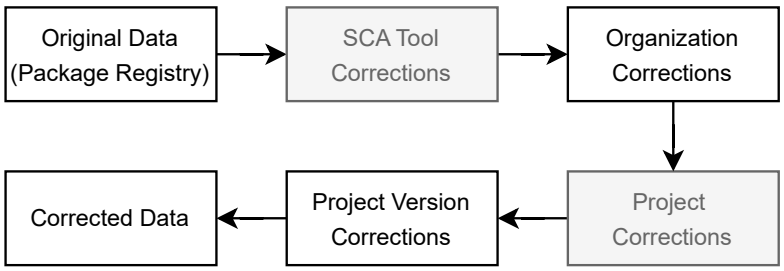

Figure 4.5: Data flow through metadata correction levels. Gray levels are not implemented.

The SCA Tool level is conceptual and can be implemented later to apply corrections to all tenants of the application. Metadata corrections for invalid or missing entries in package registries are typically made at the organization level, where they are automatically shared with all projects within that organization. The project version level serves two purposes. First, it stores project-specific overrides that should not be shared with other projects, such as download locations unique to a particular project version. Second, it enhances future SBOM upload functionality. When an uploaded SBOM contains metadata attributes, the system can offer to preserve this information through metadata corrections at the project version level rather than discarding it.

All package attributes except the Package ID and Package URL can be corrected. The discovered license cannot be corrected directly but is automatically updated through the license-clearing process.

4.3.2 Finding Curations

A *finding curation* is a data structure that, depending on the curation action, either accepts, corrects, or rejects an existing scanner finding, or creates a new one. The different actions and their effects are visualized in Figure 4.6. In the context of curation, scanner findings are also called *base findings*. Approved base findings, edited base findings with their corrections applied, as well as newly

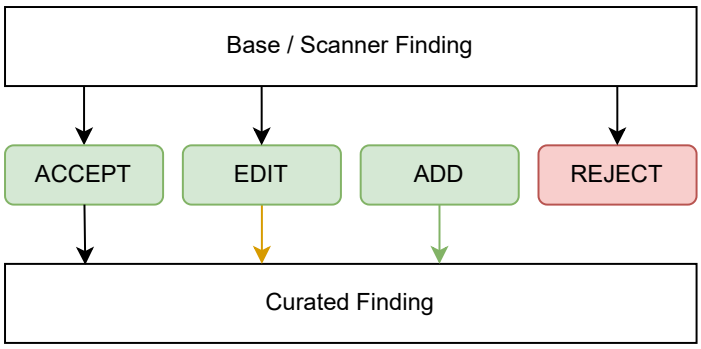

Figure 4.6: Relationship between base findings and curated findings based on the curation action.

added findings, are called *curated findings*. When requesting findings, for example for generating an SBOM or legal notices, base findings get replaced by their curated variants if a matching finding curation exists.

However, finding curations are independent entities that can also be applied without their base findings. As a result, base findings can be added, removed or updated, without breaking any existing curation data, allowing the scanner to improve. This also makes the curation actions flexible: for example, when an improved scanner detects a finding that was previously added manually, the system automatically reclassifies the `ADD` curation as an `ACCEPT`.

Reusing decisions

Because license clearing can be time-consuming, the system provides capabilities for automatically reusing decisions where possible. This mechanism leverages the fact that most findings are context-independent and all necessary information is immediately clear from the finding text itself. When a user curates a context-independent finding, this decision is automatically applied to all other codebases containing the identical file. This behavior is consistent with related tools such as DOS and FOSSology. This approach significantly reduces clearing effort in two scenarios. First, when clearing newer versions of the same package, only changed files require review. Second, when working with monorepos that typically share many files across projects, shared files need to be cleared only once.

However, not all findings are context-independent, for example when they reference other files in the codebase or external resources. In these cases, the actual license information depends on the content of the referenced file or URL, which may differ between codebases or change over time. For such context-dependent findings, the system offers local decisions, which are not automatically reused

across different codebases, ensuring that each context-dependent finding is reviewed within its specific environment.

4.3.3 File Completions

File completions indicate that a file's review has been completed. This concept is also essential for calculating the package clearance progress, which is determined by dividing the number of completed files by the total number of files in the codebase. The specific behavior of a file completion depends on its completion type:

- **Review**: Indicates that the user has fully reviewed the file, including the identification of all false negatives. A review is only allowed when all base findings in the file have been reviewed.
- **Exclude**: Indicates that all findings within the file should be disregarded. These findings are omitted from legal notices and SBOM exports and are not included in the final license calculation.
- **Blind Accept**: Indicates that the file has not been examined. All current and future base findings in this file are automatically accepted.

Reviews are the standard operation when curating findings. Excludes serve to ignore files irrelevant to the package because they are not distributed, such as test or documentation files. Typically, excludes are applied early in the license-clearing process to reduce the number of findings requiring review.

Files can be blind-accepted to explicitly skip them. This enables workflows where false negatives are accepted as a trade-off, allowing users to focus only on files that contain findings.

A file that has not (yet) been completed behaves the same way as a blind-accepted file. This maintains consistency with the behavior that existed before the changes introduced in this thesis. The only difference is that a blind-accept is explicit and increases the completion progress.

Additionally, file completions create a snapshot of the current curation state to account for scanner findings changing over time. This prevents new base findings from unexpectedly appearing in SBOM files or third-party legal notices without review. As a result, modifying curations in completed files requires temporarily removing the completion status before reapplying it.

4.4 Clearing Workflow

Based on the concepts introduced in Section 4.3, this section describes the intended user workflow for license clearing. For clarity, the workflow is structured into three hierarchical levels. Users work through each package systematically (Section 4.4.1), reviewing all relevant files (Section 4.4.2), and for each file, addressing all findings (Section 4.4.3).

4.4.1 Package-level Workflow

The package-level workflow describes the process of clearing a single dependency. This involves verifying package metadata, excluding irrelevant content, and systematically reviewing all relevant files. The workflow is illustrated in Figure 4.7.

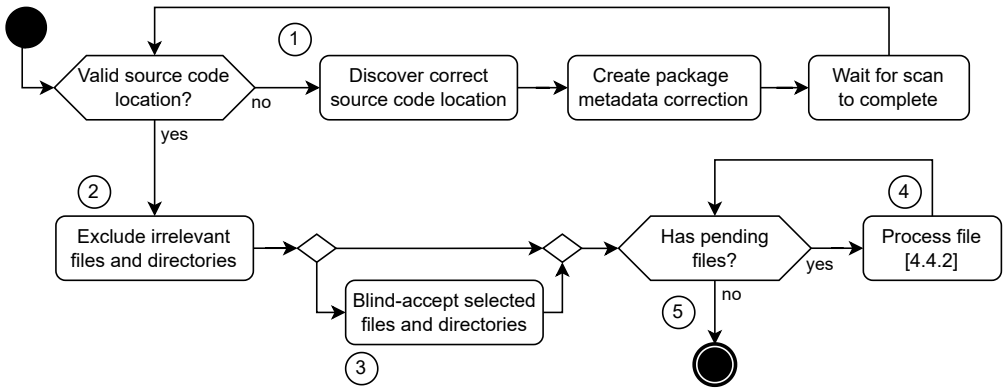

Figure 4.7: Package-level clearing workflow showing metadata verification, file exclusion, and iterative file processing.

The workflow consists of the following phases:

(1) If the source code location is invalid, the user must manually discover the correct repository URL and revision. This process typically involves using external search engines to find the repository, then examining releases or git tags to identify the correct revision. Once discovered, the user sets this information using a metadata correction, which triggers a scan of the source code. The user can either wait for the scan to complete or continue clearing other packages in the meantime.

(2) The user excludes files and directories that are not relevant to the package being cleared. This includes test files, documentation, build artifacts, or other subprojects in case of monorepos. Performing this step early is important to reduce the number of findings requiring review.

(3) For large codebases, the user can optionally select files to blind accept. This marks files as complete without manual review, automatically accepting all findings. While this approach accepts the risk of false negatives, it significantly reduces clearing effort by allowing users to focus only on files containing scanner findings or files of particular interest.

(4) The user iterates through all pending files in the package that have not been excluded, blind-accepted or completed. For each file, the file-level workflow (Section 4.4.2) is applied to review findings and mark the file as complete. This loop continues until all files in the package have been cleared.

(5) Once all files are completed, the package is considered cleared. The user can now view the accurate discovered license on the governance page and export SBOMs and legal notices that reflect all applied curations for this package.

4.4.2 File-Level Workflow

The file-level workflow describes the process of clearing an individual source code file of a package. This involves reviewing all scanner findings, checking for false negatives, and marking the file as complete. The workflow is visualized in Figure 4.8.

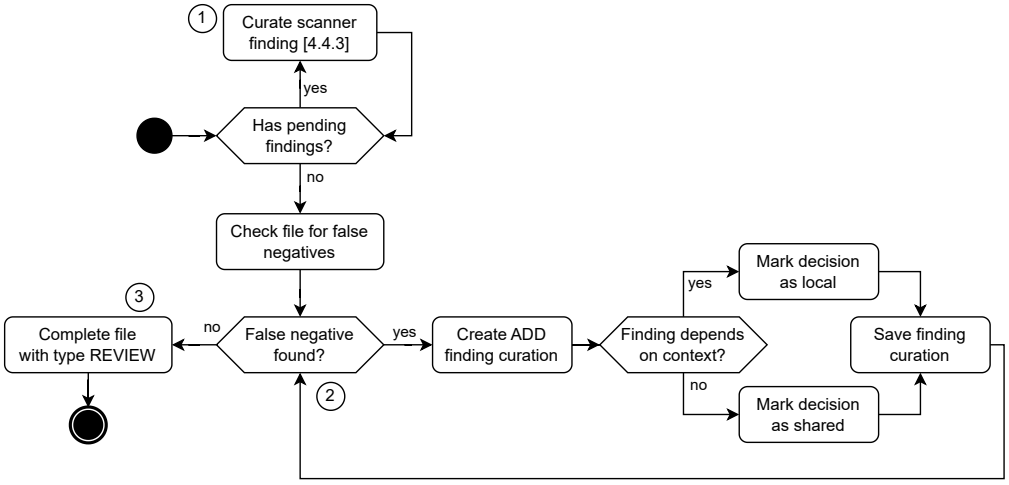

Figure 4.8: File-level clearing workflow showing the review process for scanner findings and false negatives.

The workflow consists of the following phases:

(1) The user iterates through all scanner findings detected in the current file. For each finding, the finding-level workflow (Section 4.4.3) is applied. This loop continues until all base findings have been curated.

(2) After all scanner findings have been processed, the user manually scans the file to identify any license or copyright information that the scanner missed (false negatives). For each false negative found, a new finding curation with type `ADD` is created. If the finding depends on context, the decision is made local so it is not shared between packages.

(3) Once all base findings are curated and no false negatives remain, the file is marked as complete with type `REVIEW`. The user can now continue with the next file.

4.4.3 Finding-Level Workflow

For each finding detected by the scanner, the user must determine the appropriate curation action and decide whether the decision should be reused across other codebases. This workflow is illustrated in Figure 4.9.

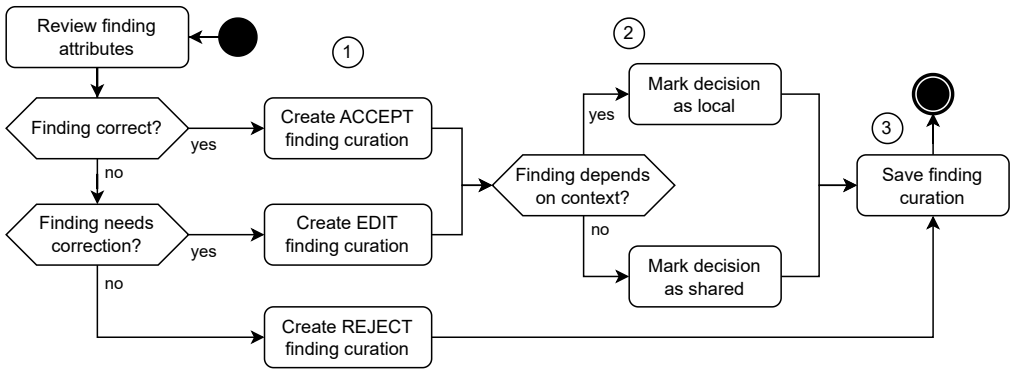

Figure 4.9: Finding-level clearing workflow with decision points for curation actions and reusability.

The workflow consists of three core phases:

(1) The user examines the finding text and its location in the source code to determine whether it is accurate, requires corrections, or is a false positive. Based on the assessment, the user selects the appropriate curation action.

(2) The user determines whether the finding is context-dependent. For context-dependent findings, the curation is marked as local and will not be automatically reused in other packages. Otherwise it is marked as shared.

(3) Finally, the finding curation is saved with the selected action and scope.

5 Design and Implementation

Building on the architectural concepts introduced in Chapter 4, this chapter presents the concrete design choices and implementation strategies. Section 5.1 provides an overview of the key frameworks and libraries used for implementation. Section 5.2 details the correction of package metadata, followed by Section 5.3, which describes the approach to scanner finding curation and license clearing. Finally, Section 5.4 explains the auditing system that tracks all entity modifications.

5.1 Frameworks and Libraries

This section outlines the key frameworks and libraries used for implementation. Since the curation logic builds upon the existing SCA Tool application, most choices align with the established technology stack. Feature-specific libraries that address particular requirements are introduced later in their respective sections.

5.1.1 Backend

The SCA Tool backend uses Java 21 with the Spring Boot[1] framework. Database interactions are managed through Spring Data JPA, which integrates with Hibernate[2] as an Object-Relational Mapping (ORM) solution for PostgreSQL.

Spring identifies and manages classes based on their declared roles within the application. These roles are typically defined through annotations. For clarity, illustrations throughout this work utilize custom UML stereotypes such as «Service» to visually represent these annotations. The primary roles include:

- **Controller**: Marks web controllers that handle HTTP requests. Because the curation controllers are mainly forwarding requests to services, they are mostly omitted in this thesis.

[1]Spring Boot - https://github.com/spring-projects/spring-boot
[2]Hibernate - https://github.com/hibernate/hibernate-orm

- **Service**: Marks classes that implement business logic. Exposed methods typically open a (potentially read-only) transaction and can be fully cached using Spring Data Redis.
- **Repository**: Encapsulates persistence logic and provides clean, domain-oriented access to data.
- **Entity**: A domain object that is mapped to a database table and is managed by a Repository.

For passing data between module boundaries, entities are converted into Data Transfer Objects (DTOs). DTOs are simple, serializable objects that carry only the data needed for a specific operation, without exposing any internal persistence details or business logic.

5.1.2 Frontend

Despite low functional cohesion, the new license-clearing interfaces were integrated into the existing Next.js-based frontend to avoid the operational overhead of maintaining a separate application. Next.js effectively supports large and heterogeneous codebases through file-based routing and automatic code-splitting. If required, a micro-frontend architecture can be considered in the future.

The user interface reuses components from the internal SCA Tool component library wherever possible to ensure visual and behavioral consistency across the application. This library builds on shadcn/ui[3], which offers customizable components based on Radix UI primitives and styled with TailwindCSS.

State management initially relied on the React Context API, but this approach became difficult to maintain as the application grew in complexity. To address this, the curation state management was migrated to zustand[4]. The new global state is not only easier to read and reason about but also more performant, as only the relevant components re-render when changes occur.

To remain consistent with the rest of the application, backend communication uses the swr-openapi library[5]. This library generates a fully typed API client from the manually maintained OpenAPI specification and integrates with the SWR[6] data-fetching library. This approach ensures build-time type safety for HTTP requests, minimizing runtime errors.

[3]shadcn/ui – https://github.com/shadcn-ui/ui
[4]zustand - https://github.com/pmndrs/zustand
[5]swr-openapi – https://github.com/openapi-ts/openapi-typescript
[6]SWR - https://github.com/vercel/swr

5.2 Correction of Package Metadata

This section describes the design and implementation of the metadata correction logic. This includes the recursive data structure that enables correction inheritance, the mechanisms for exposing correction information to other modules, and the user interface for managing corrections at different levels.

5.2.1 Data Structures

Figure 5.1 illustrates the recursive data structure required to support corrections at multiple levels. Each level consists of a `CuratedPackageInternal` object containing both a reference to its parent level and a `MetadataCorrections` object, which stores the corrected values for this level. When determining an attribute's final value, the object returns the correction value if present; otherwise, it retrieves the value from the parent.

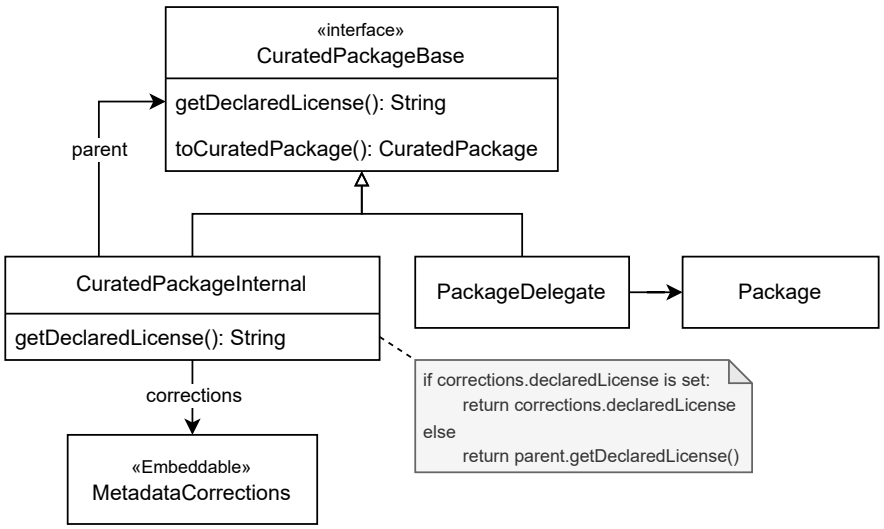

Figure 5.1: Simplified class diagram highlighting the recursive data structure for applying metadata corrections.

The base level is represented by a `PackageDelegate` object that wraps the underlying `Package` DTO, forwarding all method calls to it. This wrapper is necessary because the `Package` class resides in the oss module and cannot be modified for inheritance. This approach maintains a clear separation of module responsibilities.

At the persistence layer, the `CuratedPackageEntity` serves as the common superclass for all correction level entities (see Figure 5.2). By using the `SINGLE_TABLE` inheritance strategy, all correction data is stored in a single database table, enabling efficient retrieval of all relevant corrections. The corrections themselves are persisted using a `MetadataCorrections` embeddable object, with all attributes maintaining the same data types and constraints as the original data. A null value for any attribute indicates that no correction exists at that level.

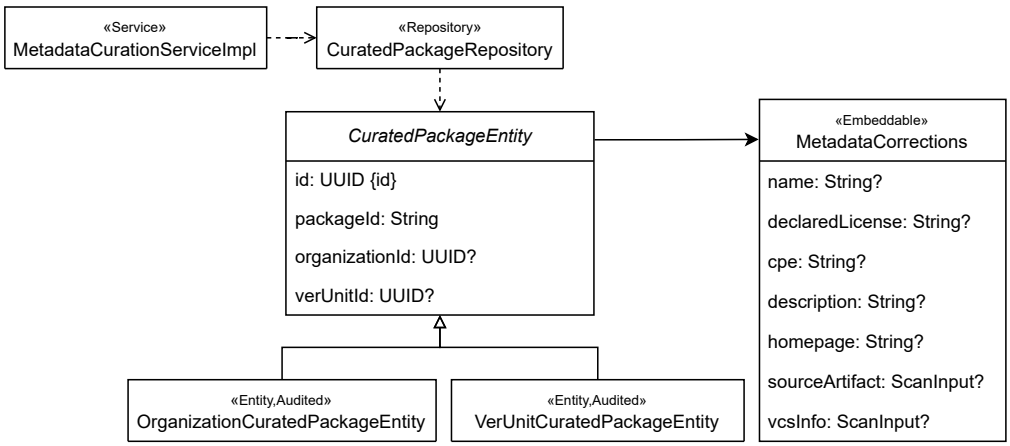

Figure 5.2: Class diagram illustrating the persistence layer for metadata correction functionality.

5.2.2 Metadata Resolution

The `CuratedPackageServiceImpl` is responsible for providing the corrected package metadata information. Because the metadata now depends on context, the service offers two retrieval methods with different levels of correction specificity:

1. getPackagesByIds(Organization org, List<String> packageIds)
   Retrieves package metadata with organization-level corrections only. This method is used when the caller has no project version available.
2. getPackagesByIds(ProjectVersion version, List<String> packageIds)
   Retrieves package metadata with all applicable correction levels (organization and project version). This method should be preferred whenever the project version is available, as it provides the most accurate metadata.

Callers should always use the most specific method available for their context to ensure the most accurate corrections are applied.

Correction Attributes

When resolving package metadata, the service provides not only the corrected data but also supplementary information about the applied corrections. This functionality is realized through the `CuratedPackage` DTO, which extends the `Package` DTO from the oss module. This polymorphic design ensures that application components uninterested in correction metadata, such as the governance or security module, continue to function without modification, while components that benefit from this information, including the SBOM module and the frontend, can utilize the additional data.

For each correctable field, the `CuratedPackage` class includes an additional attribute of type `MetadataCorrection`. This object contains metadata about the most specific correction applied, including both the correction level and the original base value that was modified. When an attribute remains uncorrected or when the corrected value matches the base value, this correction attribute remains `null`. Figure 5.3 provides an example showing a package instance with its declared license corrected from `MIT` to `MIT-0`.

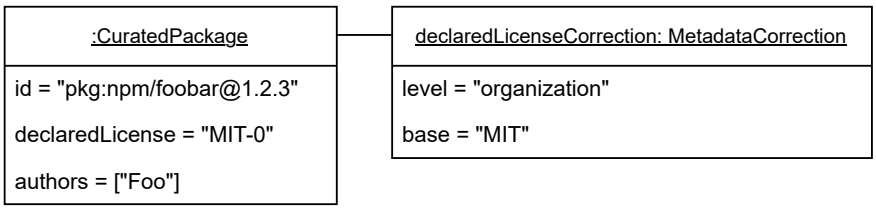

Figure 5.3: Simplified instance diagram showing a curated package object with a corrected declared license.

5.2.3 User Interface

The user interface provides a package details page, accessible through the actions column in the SBOM table. This screen presents editable form fields for each correction level, enabling users to modify individual attributes. Values inherited from the parent level appear as placeholders within the form fields. Tab headers above the form fields allow navigation between the different correction levels. A dedicated dialog allows users to edit the source location. For type VCS, users can provide a repository URL, revision, and an optional path parameter; for type Public URL, users provide the URL and an optional checksum. Screenshots of the package details page and source location dialog are provided in Appendix A.

5.3 Curation of Scanner Findings

This section details the design and implementation of the license-clearing functionality, highlighting advanced concepts, data models, and user interface.

5.3.1 Finding Curations

The data model of finding curations is visualized in Figure 5.4. Finding curations are represented by the abstract superclass `FindingCurationEntity<T>`, which is implemented for both license and copyright findings. Each implementation stores type-specific data based on the curation action. For `EDIT` actions, the entity stores override values, while for `ACCEPT` and `ADD` actions, it stores the finding values themselves. When the action is `REJECT`, these attributes remain null. The `Position` attributes of the curation follow the same pattern. Java Generics ensure type safety by constraining abstract methods to operate on the correct finding type (e.g., license finding curation with license finding).

The `CurationContext` captures all available contextual information when the curation is created. Although some values are currently unused, storing them upfront eliminates the need for data migration when enhancing the matching algorithms in the future. The `BaseFindingContext` stores attributes of the base finding targeted by the curation. For new findings without a corresponding base finding, this attribute is null.

As mentioned in Section 4.3.2, the action of a finding curation may change when base findings are modified. The `determineAction(T baseFinding)` method calculates the effective action based on the matched base finding. The logic follows these rules: `REJECT` actions always remain unchanged; when no base finding is provided (null), the action is determined to be `ADD`; and for distinguishing between `ACCEPT` and `EDIT`, the utility methods `isDataOverride(T baseFinding)` and `isPositionOverride(T baseFinding)` check whether the curation provides modifications to the finding.

5.3.2 File Completions

File completions are implemented through the `FileCompletionEntity` as shown in Figure 5.5. The `CompletionType` attribute captures the user's action: Review, Exclude, or Blind Accept. To reduce storage overhead, the default completion type (Pending) is modeled implicitly by the absence of the entity. Each completion also records a `CompletionContext`, which captures a snapshot of the context at creation time, and a `CompletionScope` (detailed in Section 5.3.3), which determines whether the completion can be shared across codebases.

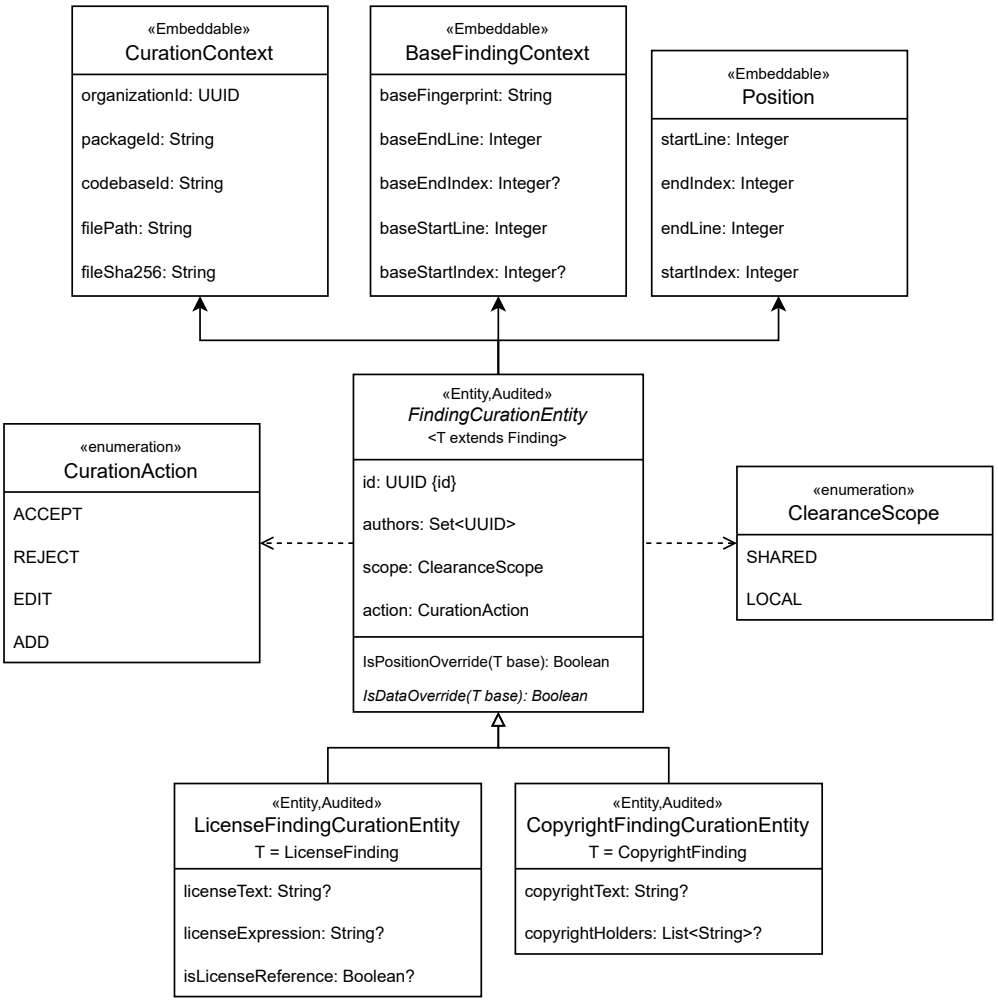

Figure 5.4: Data Model of finding curation entities.

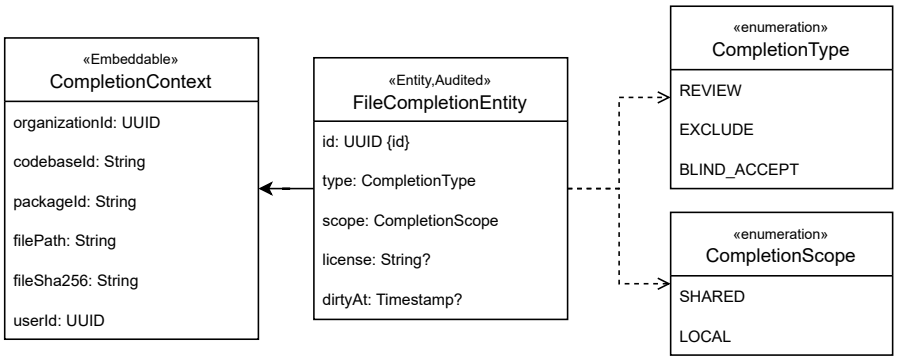

Figure 5.5: Class diagram of file completion entities.

The `FileCompletionServiceImpl` provides the `completeFiles` endpoint for creating new file completions. When `Pending` is requested as type, the existing file completion is removed. For all other types, new file completion entities are created accordingly. Data retrieval is performed by the `FileCompletionRepository` using native PostgreSQL queries.

When the completion status of a file changes, the system triggers asynchronous precomputations such as the calculation of the clearance progress. These performance optimizations are detailed in Section 5.3.6.

5.3.3 Clearance Scope

The *clearance scope* is the set of attributes in the `CurationContext` that determine whether a finding curation applies in a given context. It implements the local decision rules introduced in Section 4.3.2. A broader clearance scope applies a single curation to multiple findings across different codebases, reducing license-clearing effort. However, a broader scope also reduces flexibility, which is needed for context-specific findings. The appropriate scope must therefore be chosen for each situation.

Two clearance scopes have been implemented: *shared* and *local*. The shared scope is used by default and operates based on the `fileSha256` hash, applying the finding curation to identical files across all codebases within the organization. This scope is appropriate for license texts and unambiguous license references that do not depend on other files in the codebase. It provides high reuse, as identical files only need to be reviewed once.

The local scope is used when shared scope is not applicable. Instead of the file hash, it includes a combination of `packageId`, `codebaseId`, and `filePath`. Consequently, the finding curation is tied to a specific package and is not automatically reused between different packages.

Additional scopes were considered but not implemented. One option would omit the organization identifier from the scope attributes, causing curations to affect all tenants of SCA Tool. This could enable new tenants to benefit immediately from aggregated, manually reviewed curation data. Another option would remove file-level information entirely and rely only on the matched finding text, maximizing reuse but risking incorrect automatic application when legally important surrounding text is missed by the scanner. Such a scope might nevertheless be useful later for generating suggestions.

The clearance scope concept likewise applies to file completions, where it is referred to as `CompletionScope`. For completions of type `REVIEW`, a file completion is classified as *shared* only when every finding curation for that file uses the shared scope; otherwise, the completion is *local*. Completions of type `EXCLUDE` and `BLIND_ACCEPT` are always local.

Multiple scopes significantly increase implementation complexity, requiring complex SQL queries for efficient data retrieval and careful handling of edge cases when multiple decisions apply to the same item. The benefits of this feature should be weighed against this overhead in future evaluations.

5.3.4 Bulk Operations

Bulk operations significantly improve the user experience by applying updates to multiple entities as a single unit based on a specific criteria. These operations can target file completions, finding curations, or a combination of both.

While bulk operations could be implemented in the frontend, a backend implementation offers several important advantages:

- **Performance**: Executing bulk operations close to the database ensures optimal performance. The backend can also leverage caching mechanisms more effectively than client-side implementations.
- **Atomicity**: When execution is expected to be quick, operations can run within a single transaction, guaranteeing all-or-nothing behavior.
- **Reliability**: Operations continue executing even if the user navigates away or loses connectivity, preventing partial completions.
- **Locking**: The backend can ensure that only one bulk operation executes at a time, preventing unexpected behaviors.

For maintainability, each bulk operation algorithm is implemented in its own Java class, as shown in Figure 5.6. The `BulkActionContext` encapsulates all parameters shared between operation types. The `dryRun` parameter allows operations to run without performing modifications, enabling the frontend to preview

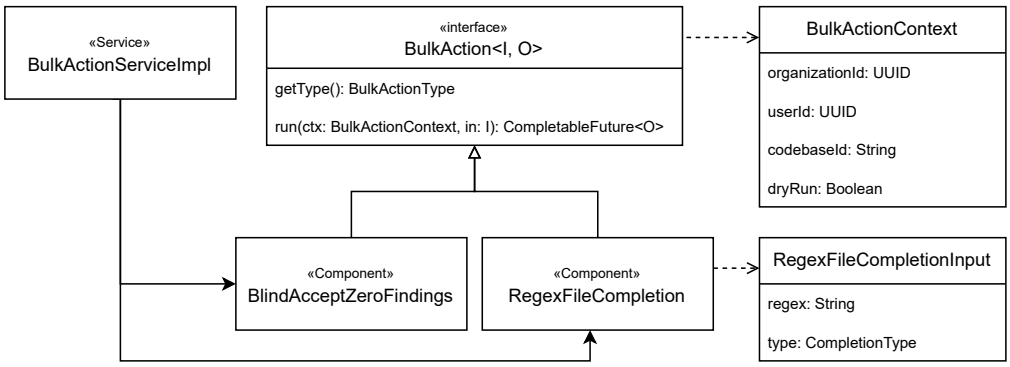

Figure 5.6: Class collaboration for realizing bulk actions.

affected files before execution. Types can also define additional input and output parameters if required. By default, operations return the number of affected files.

Two bulk operation types have been implemented:

- `BlindAcceptZeroFindings`: Sets the completion status to *blind-accepted* for all files without scanner findings. This enables workflows where only scanner findings instead of all files are reviewed, ignoring false negatives.
- `RegexFileCompletion`: Updates completion status for all files whose paths match a specified regular expression. A common use case is excluding test files or documentation directories.

5.3.5 Source Code Fetching

To display source files in the Clearance UI, the application requires a method for retrieving source code even after the scanning process has completed. Rather than persistently storing source code, the backend relies on the APIs provided by external source code providers, eliminating the need to manage data storage and implement efficient retrieval algorithms. This approach, however, creates dependencies on external service availability, as discussed in Section 6.3.2.

The `CurationController` calls the `SourceProviderService` from the oss module to retrieve the source code of a file. All source handlers implement the `SourceProvider` interface and are sorted by priority. Each provider determines whether it can provide the source for the given input. When a source handler cannot provide the source code, for example, due to an incompatible input or a request error, the handler with the next highest priority acts as a fallback. Fetched source files are cached in Redis to speed up subsequent requests and mitigate rate limit constraints from external providers. Figure 5.7 visualizes the static structure of the relevant classes.

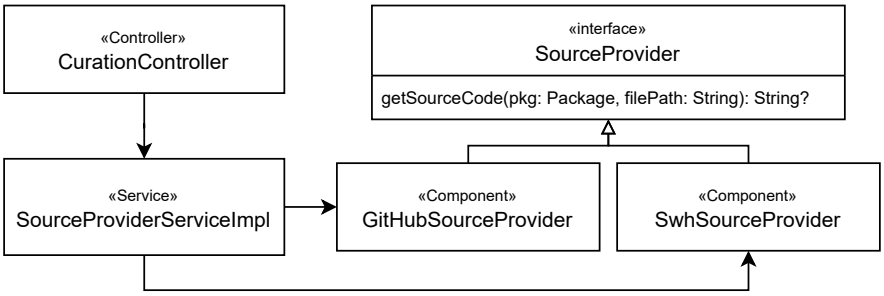

Figure 5.7: Classes involved in source code fetching.

Two source providers have been implemented. The primary provider is GitHub, as it hosts the majority of open source repositories. The GitHub REST API offers several advantages: low response times, high rate limits (5,000 requests per hour), and the ability to fetch individual files. The handler makes HTTP requests to the API using the repository owner, repository name, and file path as parameters.

The second provider is the Software Heritage[7], an initiative that aims to preserve all publicly available source code. Like GitHub, the API supports fetching individual files. However, response times are significantly slower (though not formally measured), and endpoints were occasionally unresponsive during development. The rate limit is also much lower at 1,200 requests per hour for authenticated users. For fetching source code, only the SHA256 hash of the file contents is needed as parameter.

5.3.6 Performance

While performance was not the priority of this thesis, initial deployments caused significant slowdowns across the application, even without any finding curations or file completions present. This issue stems from the shared clearance scope, which requires SHA256 file hashes to retrieve curations and completions. When calculating the clearance progress for an entire project or generating SBOMs and third-party legal notices, the database must perform numerous index reads on the files table. Combined with HDD storage, this resulted in unacceptable loading times.

To address this issue, a precomputation strategy was implemented that caches relevant data at package level. The system now computes and persists the discovered license, clearance progress, and a list of reviewed and excluded files for each package whenever a file's completion state changes. The implementation

[7]Software Heritage - https://www.softwareheritage.org/

uses dirty flags (implemented as timestamps) on both packages and files, in combination with two cron jobs. The workflow is illustrated in Figure 5.8.

When the completion status changes, the system determines which packages need updating based on the `CompletionScope`. If the scope is `LOCAL`, the completion affects only the current package, which is marked dirty immediately. If the scope is `SHARED`, the completion affects all packages containing this file. Since this could potentially include a high number of packages and slow down the operation, only the file itself is marked as dirty in this phase.

The first periodic job propagates dirty file markers to the packages that contain them. It loads dirty files in batches, groups them by organization, resolves the containing packages and marks them as dirty. Finally, it removes the dirty flag from the processed files. This job currently runs every 30 seconds.

The second periodic job recalculates the cached data for dirty packages. For each dirty package, the job fetches file data and file completions, then computes the discovered license, the clearance progress, and a set of file completions where the completion type equals `REVIEW` or `EXCLUDE`. Completions of type `BLIND_ACCEPT` do not need to be persisted because their behavior, by definition, equals that of incomplete files. Upon successful completion, the dirty flag is reset. This job also runs every 30 seconds, independently of the first job.

As a result of this optimization, read-heavy operations such as SBOM generation can perform simple lookups against the precomputation table, eliminating the performance bottleneck. However, the approach also introduces several tradeoffs. First, the implementation is complex, requiring careful maintenance. When file completions, codebases, or packages are modified without accounting for precomputations, the cached data becomes invalid. Second, the asynchronous nature causes temporary data staleness for up to 60 seconds until both cron jobs complete. This can lead to unexpected behavior when users generate an SBOM or third-party legal notices immediately after license clearing, as the changes may not yet be reflected in the precomputed data.

Due to the urgency of the performance problems, alternative approaches were not thoroughly explored, leaving room for future improvement. Optimizations could include incremental updates that compute only changed data, or implementing a synchronous fast path for small changes, especially once the hardware situation improves.

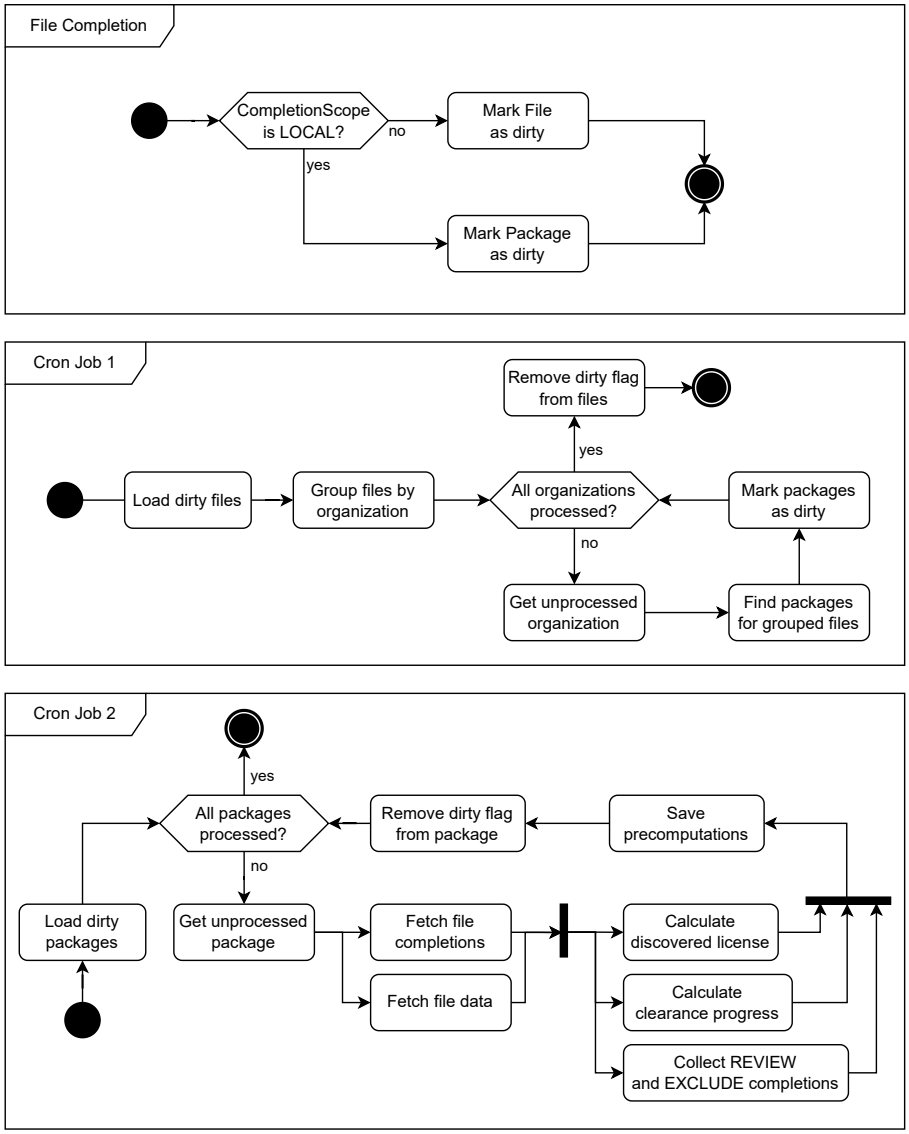

Figure 5.8: Workflow of precomputing discovered license, clearance progress and completion lists.

5.3.7 Finding Resolution

The `CuratedFindingServiceImpl` provides the corrected license and copyright data for packages by applying file completions and finding curations to scanner findings. This service is used when generating SBOM files or third-party legal notices. Figure 5.9 illustrates the process for a single package.

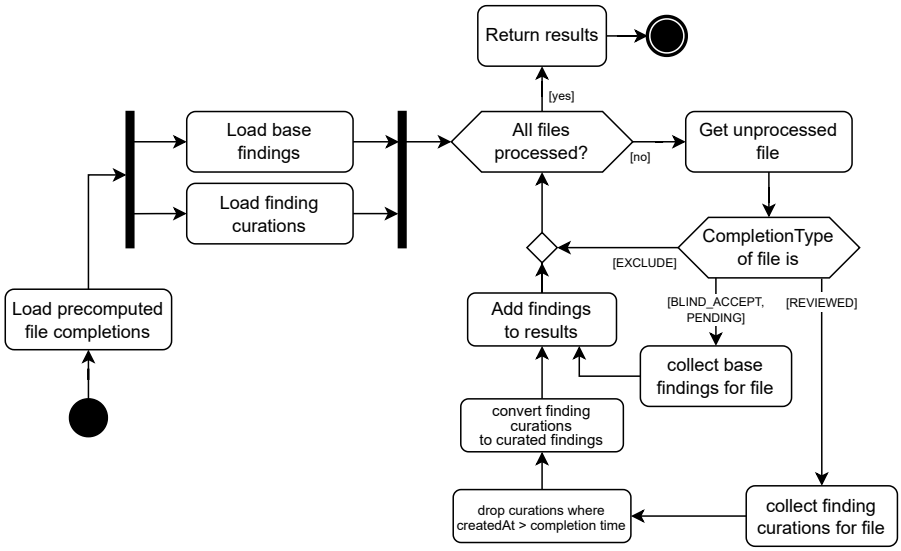

Figure 5.9: Process of resolving corrected findings.

The resolution process begins by loading the precomputed file completions (see Section 5.3.6), scanner findings, and finding curations from the database. File completions are loaded first to determine which curations are relevant and to account for different clearance scopes (Section 5.3.3).

The service iterates over all files. The file set is determined by merging file information from base findings and finding curations; files without findings are not processed. For each file, the service performs the following actions based on its completion type:

1. `EXCLUDED`: No findings are added to the results, and the process immediately continues with the next file.
2. `BLIND_ACCEPTED` and `PENDING`: Only base findings are added to the results without applying any corrections.
3. `REVIEWED`: Finding curations (`REJECT` curations excluded) are added to the results and base findings are ignored. Curations can be easily converted to findings as the entity already contains all necessary data.

5.3.8 User Interface

For the license-clearing frontend, two new pages were added: the Package Overview and the License-Clearing UI. The latter is divided into three panels: Files, Source Code, and Actions.

Users can access the new curation interfaces in two ways: through the SBOM page by clicking on the clearance progress of a package, or through the Governance page by clicking on a package's license. Both routes lead to the Package Overview, with the Governance route additionally applying a license filter to source files.

Package Overview

The Package Overview page displays the detailed clearance progress for a package, along with metadata about the package, source code, and latest scan. It provides a list of source files showing each file's completion status, license, and number of scanner findings. Users can enter the Clearance UI by clicking on any file in the list or by clicking the *Start Curation* button. A screenshot of the page is provided in Appendix B.

Files Panel

Located on the left-hand side of the License-Clearing UI, the Files Panel displays a tree view of the package's source files. Each file shows its finding count and completion status, similar to the information provided in the Package Overview. Files can be opened in the Source Code Panel by clicking on them, while right-clicking provides a context menu with exclude and blind-accept options.

Source Code Panel

The Source Code Panel displays the source code of the currently active file with all scanner and curated findings highlighted. CodeMirror[8] is used to display the code in a read-only editor instance. Multiple custom CodeMirror extensions were developed to support the curation features:

- **KeywordHighlighter**: Highlights keywords based on regular expressions such as `license` or `copyright`. Helps users identify false negatives and can be deactivated if needed.
- **SelectionTracker**: Extracts the current text selection from the editor when modifying finding positions or adding new findings.
- **FindingScroller**: Enables scrolling to specific findings, for example when a finding is clicked in the finding list of the overview tab.

[8]CodeMirror - https://codemirror.net/

- **FindingHighlighter**: Highlights scanner and curated findings in the editor using different background colors for different actions. Scanner and curated findings use distinct patterns to differentiate overlapping findings.

Additionally, users can switch between four different views. A visual comparison between the Merged View and Split View is provided in Appendix B.

- **Base Only**: Shows only base findings, hiding all curated findings.
- **Curated Only**: Shows only curated findings, hiding all base findings.
- **Merged View**: The default view, displaying both base and curated findings in a single editor.
- **Split View**: Displays two editors with synchronized scrolling. The left editor shows base findings, while the right editor shows curated findings.

Action Panel

The Action Panel is used for viewing finding details and managing finding curations. Located on the right-hand side of the Clearance UI, it contains multiple tabs that appear contextually based on the currently active finding:

- **Overview Tab**: The default tab, displaying a list of all findings in the current file. Clicking on a finding selects it, producing the same result as clicking the finding directly in the Source Code Panel. Once all findings have been reviewed, the tab offers the option to complete the file.
- **Details Tab**: Displays detailed information about the currently active finding. Base findings show Accept and Reject buttons, with the Accept button changing to Edit when fields are modified. For curated findings, the tab offers options to delete or modify the curation.
- **Diff Tab**: Compares the matched text of the currently active license finding with standard license texts from the ScanCode LicenseDB[9] and shows a similarity percentage to help identify the correct license classification. This tab is disabled when no license finding is selected.

Keybindings

Keybindings were implemented to accelerate curation workflows and fulfill requirement NF-07, covering file and package navigation, file completion, and view selection. The system uses a global key event listener with an extensible architecture that makes it easy to add new shortcuts in the future. All available keybindings are documented in a table accessible through the curation settings dialog.

[9]ScanCode LicenseDB - https://github.com/aboutcode-org/scancode-licensedb

5.4 Auditing and Activities

Auditing involves tracking all changes made to an entity over time. While Spring provides basic auditing capabilities through Spring Data JPA, its functionality is limited, allowing only the storage of author and timestamp information. To address these limitations, the implementation uses Hibernate Envers[10] instead, an official extension of Hibernate ORM. Envers logs all entity changes and provides query mechanisms for retrieving historical versions. Setting up Envers is straightforward: auditing is enabled by adding the `@Audited` annotation to entities that require tracking. When persisting or deleting an audited entity through a repository, Envers automatically inserts a new row into the corresponding revision table.

To track why changes were made, *curation activities* were introduced. For every package metadata correction, finding curation, file completion, or bulk action, the system creates a curation activity. By configuring a custom revision entity in Envers, each new revision is automatically linked to the currently active activity, enabling full traceability of which actions resulted in which changes. Figure 5.11 illustrates how curation activities are linked to revisions. At the beginning of a curation operation, the `CurationService` stores the activity identifier in the `AuditContext`, an object scoped to the current user request. Later, when Hibernate persists the changes, the `CustomRevisionListener` retrieves the identifier from the `AuditContext` and attaches it to the revision, creating the link shown in Figure 5.10.

A drawback is that Envers supports only a single revision entity per application. This limitation does not align well with the modulithic architecture of the backend. To address this, the auditing logic was centralized in a dedicated *audit* module, introducing more coupling than desirable.

Currently, these auditing capabilities remain largely unutilized. They will prove valuable for implementing future event logs and undo functionality, particularly for bulk operations.

[10] Hibernate Envers - https://hibernate.org/orm/envers/

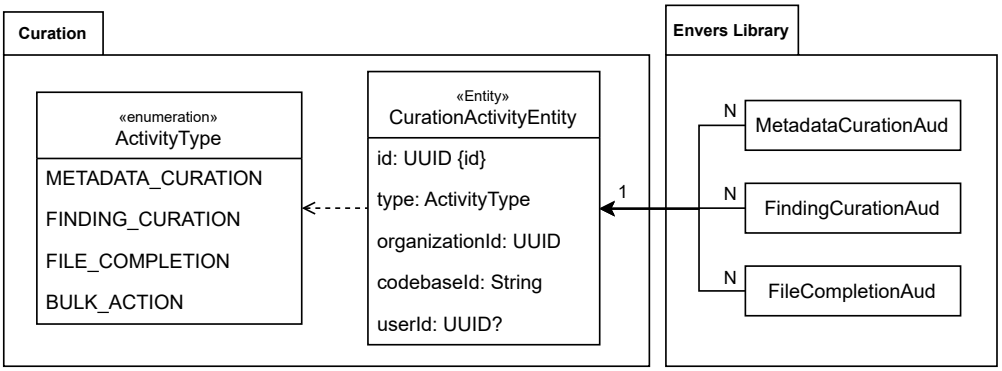

Figure 5.10: Dependency between curation activities and Hibernate Envers revisions.

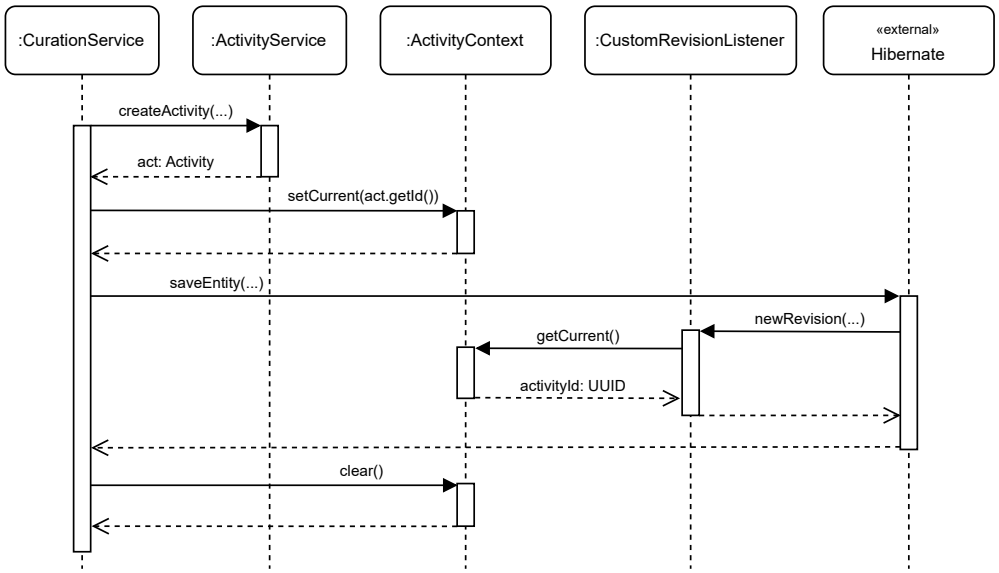

Figure 5.11: Sequence diagram (simplified) highlighting the communication between the curation service, activity service and JPA/Hibernate.

6 Evaluation

This chapter evaluates the implementation against the previously defined requirements. It first describes the evaluation process (Section 6.1), then presents results for functional (Section 6.2) and non-functional requirements (Section 6.3), and concludes with known limitations (Section 6.4).

6.1 Evaluation Process

Multiple approaches were employed to assess both the functional correctness and the compliance with the non-functional requirements. Throughout the development process, extensive manual testing and iterative testing with Playwright[1] ensured that basic curation workflows functioned correctly, with critical issues being resolved as they emerged.

The final evaluation, conducted after completing implementation, involved license-clearing an SCA Tool project consisting of the 30 most-dependent packages from the deps.dev dataset provided by Google (2025). This approach aimed to identify edge cases that the application could not sufficiently handle. Additionally, artificial test cases were constructed to trigger expected problem scenarios.

The performance evaluation for requirement NF-16 was conducted using k6[2], a load testing tool developed by Grafana. The test suite was adapted from existing end-to-end curation tests to simulate realistic user interactions. Testing was performed against the SCA Tool staging environment, which has a hardware configuration similar to the production instance to ensure representative performance metrics. Performance data was collected through Spring Boot Actuator and Prometheus for analysis.

[1] Playwright - https://playwright.dev/
[2] Grafana k6 - https://k6.io/

6.2 Functional Requirements

This section evaluates the implementation against each functional requirement specified in Section 3.2. Requirements are classified as either fully met (complete implementation), partially met (limited functionality), or not met (absent or non-functional). Results follow the original requirement order.

6.2.1 Evaluation of Dependency Graphs Corrections

Due to the time constraints of this thesis and unexpectedly high implementation complexity, functional requirements F-1 through F-3 were excluded from the thesis scope, enabling more focused attention on the remaining requirements.

The complexity stems from the current architecture of the analyzer, which heavily relies on the OSS Review Toolkit (ORT). Currently, the analyzer step integrates both dependency graph extraction and package metadata retrieval into a single process, which complicates the reuse of metadata fetching functionality for manually added packages, especially after the initial analyzer run. Decoupling these two operations would not only address this issue but also facilitate planned SBOM upload functionality. Two GitHub issues exist in the ORT repository that are related to this problem (Schuberth, 2025a, 2025b).

6.2.2 Evaluation of Metadata Corrections

F-04: The requirement has been fulfilled. Version-level metadata corrections enable modifications for specific packages within individual project versions. These modifications only affect the targeted project version, while unchanged attributes are inherited from organization-level corrections and the original registry data.

F-05: This requirement has been fulfilled. Organization-level metadata corrections allow package metadata to be corrected across all projects within an organization. Unchanged attributes are inherited from the original registry data.

F-06: The requirement has been fulfilled. The source location of a package can be configured through a dialog, which provides form fields for VCS information and public URLs.

F-07: This requirement has not been fulfilled. The system currently does not support direct source code uploads for packages because scan results would be shared across all tenants. The multi-tenancy concept described in Section 4.2.2 has not been integrated yet, as the new scanner implementation, which uses a dedicated PostgreSQL database, requires further evaluation of its performance and scalability.

F-08: The requirement has been fulfilled. The system automatically schedules a new scan when the source code of a package is updated. After the scan is complete, the source code can be license-cleared as expected. For sources provided via VCS information or public URL, the system only performs a scan if the new value differs from the original data and has not been scanned before.

6.2.3 Evaluation of Finding Curations

F-09: The requirement has been partially fulfilled. Users can accept, reject and modify license and copyright findings. The system implements each curation action by creating a finding curation instance with a specific action type that references the original finding. For some cases, such as curating license headers, the workflow is not practical yet (see Section 6.4.2).

F-10: The requirement has been fulfilled. Users can add additional license and copyright findings to source files. These new findings are represented as finding curations as well, but without a reference to any existing base finding.

F-11: The requirement has been fulfilled. Finding curations are linked to files and automatically apply to all codebases within the organization that contain those files. As a result, identical files only need to be reviewed once, saving significant time and effort. However, the original requirement did not take ambiguous or context-dependent license references into account. This special case is handled by making these curations local (see Section 5.3.3).

F-12: The requirement has been fulfilled. Once a file is marked as reviewed, curated findings replace their base findings in both SBOM and legal notice documents, and any newly added findings are automatically included.

F-13: The requirement has been fulfilled. Once a file is marked as reviewed, any rejected base findings it contains are hidden from generated SBOM and legal notice documents.

File Reviews

F-14: The requirement has been fulfilled. Users can exclude files through two methods: using the context menu when right-clicking on files or directories in the files panel, or by using a designated hotkey. When files are excluded, their findings are omitted from the generated SBOM and legal notice documents, and they do not influence the package's final license. File exclusions have limitations when files contain both relevant and irrelevant findings. In such cases, the exclusion mechanism lacks the granularity to differentiate between them (see Section 6.4.1).

F-15: The requirement has been fulfilled. Users can mark files as blind-accepted using the same process used for file exclusions. Additionally, the whole codebase can be blind-accepted at once by performing a regex bulk operation that was implemented for F-16.

F-16: This requirement has been partially fulfilled. The system provides bulk operations that allow users to exclude or blind-accept files based on regular expressions (see Section 5.3.4). However, these actions cannot currently be reused when license-clearing newer versions of the same package, as detailed in Section 6.4.3.

F-17: This requirement has been fulfilled. The system provides a bulk operation that allows users to blind-accept all files without a scanner finding.

F-18: This requirement has been fulfilled. When a file is marked as reviewed or accepted, its license is determined by combining all license expressions from its findings. The system then updates the licenses of all codebases containing that file, calculating each codebase license from the combination of all its file licenses. These updated licenses are reflected throughout the application.

License-clearing UI

F-19: The requirement has been fulfilled. The overview tab in the action panel lists all findings found in the currently active source file. Users can view details about a finding by clicking on the respective list item.

F-20: The requirement has been fulfilled. Users can navigate between packages and files using the action bar or by using hotkeys. Navigating between findings is possible by clicking on the finding in the overview panel or by clicking in the highlighted area in the source code panel.

F-21: The requirement has been fulfilled. Users can select a merged view that shows base findings and curated findings in the same panel.

F-22: The requirement has been fulfilled. Users can select a split view that shows the source file in two panels. The base findings are visible on the left, while the curated findings are visible on the right.

F-23: The requirement has been fulfilled. If the user clicks on a base finding or curated finding in the source code panel, the finding details are presented in the action panel.

F-24: The requirement has been fulfilled. Users can configure the position of a finding by highlighting text in the source code panel and clicking the *Import from Selection* button in the action panel.

6.3 Non-Functional Requirements

This section evaluates the implementation against the non-functional requirements from Section 3.3, applying the same assessment approach used for the functional requirements.

6.3.1 Security

NF-01: The requirement was fulfilled. Auditing using the Hibernate Envers library ensures changes of metadata corrections, finding curations and file completions are logged together with the timestamp and user id (see Section 5.4). Additionally, curation activities track why a change was made, which is especially useful for bulk operations that trigger multiple changes.

NF-02: This requirement has been fulfilled. A curation permission and a *curator* role have been added to the existing authorization logic. Only users with the curation permission can manage metadata corrections, finding curations or file completions. For viewing curations, the `organization.view` permission is reused.

NF-03: This requirement has not been fulfilled. The system cannot guarantee that curation data remains visible exclusively to the tenant that owns it. Although the risk is low, a misconfigured query could potentially expose such private data. A multi-tenancy concept has been successfully developed for scanner data based on PostgreSQL RLS policies (see Section 4.2.2). This approach can be extended for curation data in the future.

6.3.2 Reliability

NF-04: This requirement has been partially fulfilled. GitHub serves as the primary source code provider, with Software Heritage functioning as the fallback provider (see Section 5.3.5). Although this setup works for most open source packages, those not hosted on GitHub have no fallback provider. It is also not safe to assume that the Software Heritage can provide all necessary files. The application should implement a persistent storage solution for packages, which will be required for direct user uploads regardless.

NF-05: This requirement has been fulfilled. The curation data is stored in the primary database, which utilizes CloudNativePG[3] with backups and WAL archiving configured. This configuration ensures that data loss remains significantly below the 30-minute threshold in the event of a database failure.

6.3.3 Interaction Capability

NF-06: The requirement has been fulfilled. The frontend uses all basic styles and components such as buttons or dialog components from the internal SCA Tool component library.

NF-07: The requirement has been fulfilled. All core actions are available via keybindings and are therefore immediately accessible. For mouse interactions, many operations are also directly available in the action bar.

NF-08: The requirement has been fulfilled. The UI shows a confirmation dialog for destructive operations such as deleting a curation, performing a bulk operation or changing the completion status of a file. To not disrupt workflows, the confirmation dialog for file completions can be skipped for the current session by clicking a checkbox.

NF-09: The requirement has not been fulfilled. To simplify the user interface, the bulk action dialog shows only the number of affected files, not which specific files will be affected.

NF-10: The requirement has been fulfilled. The UI provides consistent loading indicators for data fetches (files, source code, findings) and form submissions (curations, file completions).

NF-11: The requirement has been fulfilled. A centralized TypeScript file specifies color mappings, which are used by all components, therefore ensuring consistent color usage.

[3] CloudNativePG - https://cloudnative-pg.io/

NF-12: The requirement has been partially fulfilled. Common error messages (e.g., overlapping findings) are descriptive, while others remain generic. Comprehensive error handling was outside the scope of this thesis and requires future improvement.

NF-13: The requirement has been fulfilled. All icon buttons have tooltips that display descriptive text on hover.

6.3.4 Compatibility

NF-14: The requirement has been fulfilled. The curation logic is implemented by the curation module, which is part of the existing Spring Boot backend. The frontend resides under the `/curation/` subpath of the Next.js application and uses UI components from the internal library.

6.3.5 Maintainability

NF-15: The requirement has been fulfilled. When creating finding curations or file completions, all available context is stored along the data item. Therefore, a new algorithm always has all context available and can be implemented without data migrations or making old curation data unusable.

6.3.6 Performance Efficiency

NF-16: The requirement has not been fulfilled. Instead of testing all 100 users simultaneously, the load test used batches of 4, 16, 32, and 44 users, reaching a peak of 70 concurrent users. Performance degradation correlated with the number of concurrent users, though the system remained reliable with no errors. Further investigation is needed to improve performance, particularly regarding the authentication layer.

6.4 Limitations

This section briefly discusses the key limitations identified during the evaluation and explores potential solutions.

6.4.1 Irrelevant Findings

In certain cases, a single file, typically the `LICENSE` file in the repository root, specifies licenses for multiple files or directories. If not all referenced files are relevant to the package being cleared, this file contains a mix of relevant and irrelevant findings.

File exclusions cannot address this situation because their scope is too broad. Rejecting the irrelevant findings would also be incorrect, as these are true positives that may be relevant in other contexts.

A possible solution would involve implementing rejection reasons. When rejecting findings, users could distinguish between false positives and irrelevant findings. While false positive decisions can be shared across packages, irrelevance is context-specific and should only apply to the current package.

6.4.2 License Headers

Many projects include license headers at the top of each source file to specify license and copyright information. While this practice improves clarity and aligns with the REUSE[4] specification, it generates an overwhelming number of scanner findings. For example, running ScanCode Toolkit on Apache Hadoop[5] results in over 13,000 license findings with identical text and position.

Manually curating these findings individually is impractical, making bulk curations essential. A simple approach groups findings by identical text. However, this approach cannot handle false negatives where the scanner fails to detect certain headers entirely.

A more robust solution would implement a bulk recognition feature similar to FOSSology, which searches through all files and applies user-specified decisions. However, this approach is currently not feasible, as it requires fast access to all source files. Future development should enable SCA Tool to store source code directly rather than relying on external providers. This would also address the source code availability issues identified in the evaluation.

6.4.3 Reusing File Completions

Exclude and Blind Accept file completions currently affect only a single package version and cannot be reused. However, these decisions typically remain valid across all package versions. For instance, files with the `.test.js` suffix or entire `test` directories should consistently be excluded across all versions. This limitation causes duplicate effort when updating packages to new versions.

One solution would allow users to copy decisions between packages, similar to FOSSology's Reuser agent. When opening the Clearance UI for a new package version, the system could offer to copy decisions from the previous version. This relatively straightforward enhancement would significantly reduce repetitive work during package updates.

[4]REUSE - https://reuse.software/spec-3.2/
[5]Apache Hadoop - https://github.com/apache/hadoop

7 Conclusion

This thesis addressed the accuracy limitations of automated tasks in software composition analysis by implementing data curation capabilities for SCA Tool. The work targeted three areas where manual intervention improves automated results: dependency graph corrections, package metadata corrections, and the curation of scanner findings.

The first goal focused on supporting manual dependency graph corrections, allowing users to add or remove components as needed. This feature proved largely incompatible with the current analyzer architecture and was therefore dropped to prioritize other targets. Future development should decouple dependency extraction from metadata fetching, which would address this issue while also enabling direct SBOM uploads.

The second target addressed package metadata corrections, particularly for specifying accurate source code locations needed for copyright and license extraction. The solution allows users to view and override metadata attributes at multiple levels, where each level determines the degree of reuse across projects. The evaluation showed that the implementation met most requirements, with unsupported direct source code uploads as the main limitation.

The third target introduced license-clearing functionality to SCA Tool. Users can review all source files of their dependencies, correcting scanner findings and adding new findings when false negatives occur. A modern UI supports this workflow with advanced features such as keybindings, bulk operations and highlighting of findings in the source code. Changes affect other SCA Tool solutions, including open source governance, SBOM export, and legal notice generation. The evaluation demonstrated that basic license-clearing workflows function correctly, while remaining challenges include handling findings irrelevant to the distribution, managing codebases with license headers, and reusing decisions across newer package versions.

Overall, SCA Tool users can now address many of the accuracy issues of automated processes, moving an important step closer to the reliability needed for license compliance.

Appendices

A Package Metadata Page

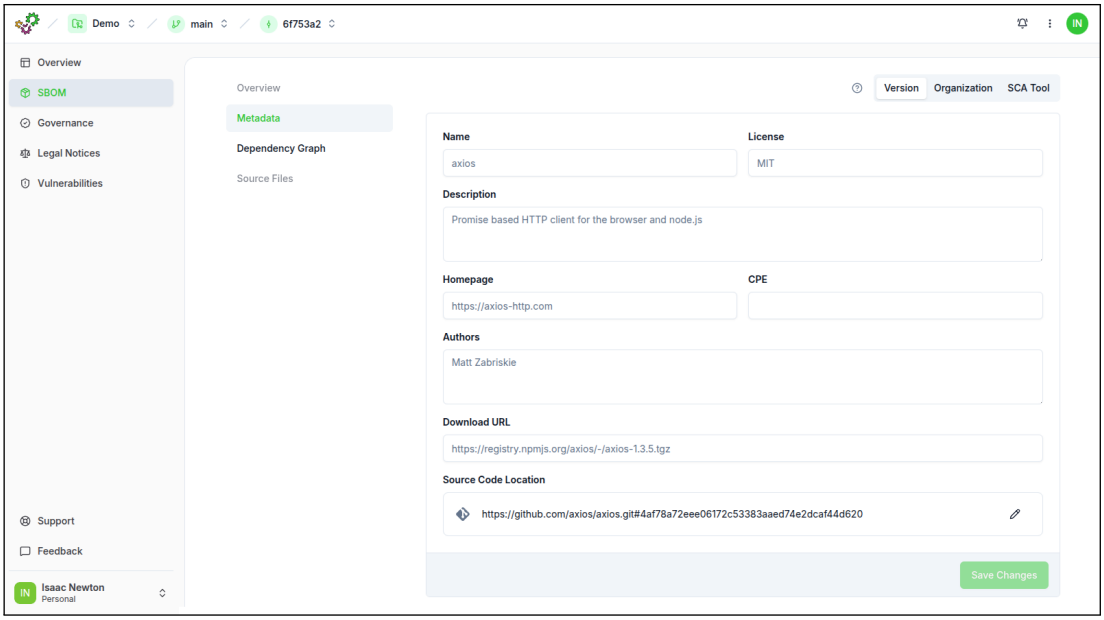

Figure 1: Package details page for viewing and correcting package metadata.

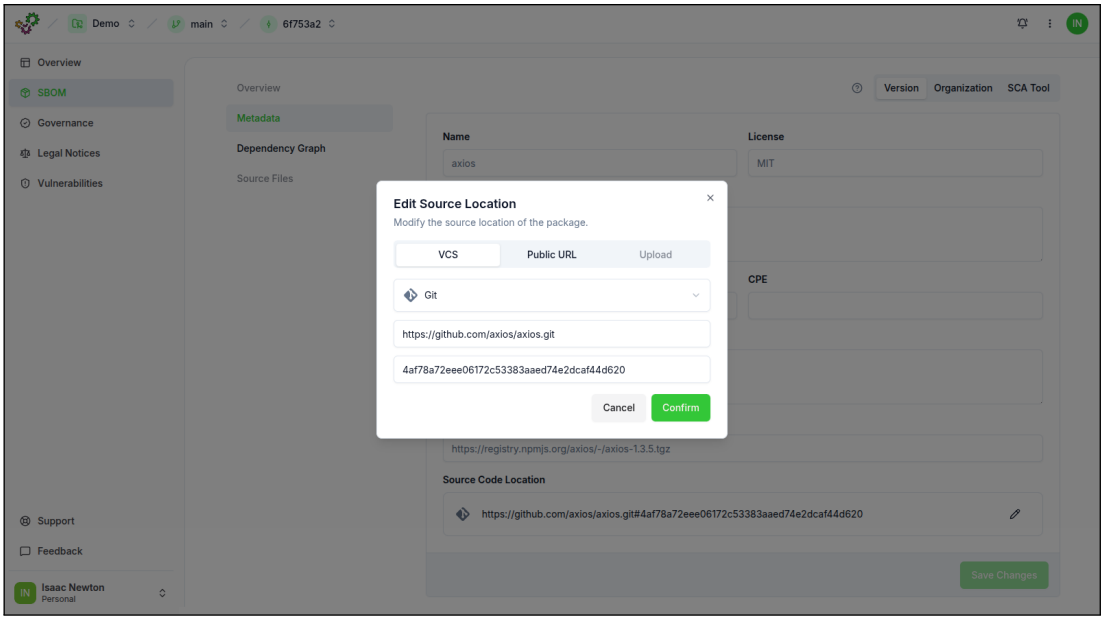

Figure 2: Dialog for correcting the source code location of a package.

B License-Clearing UI

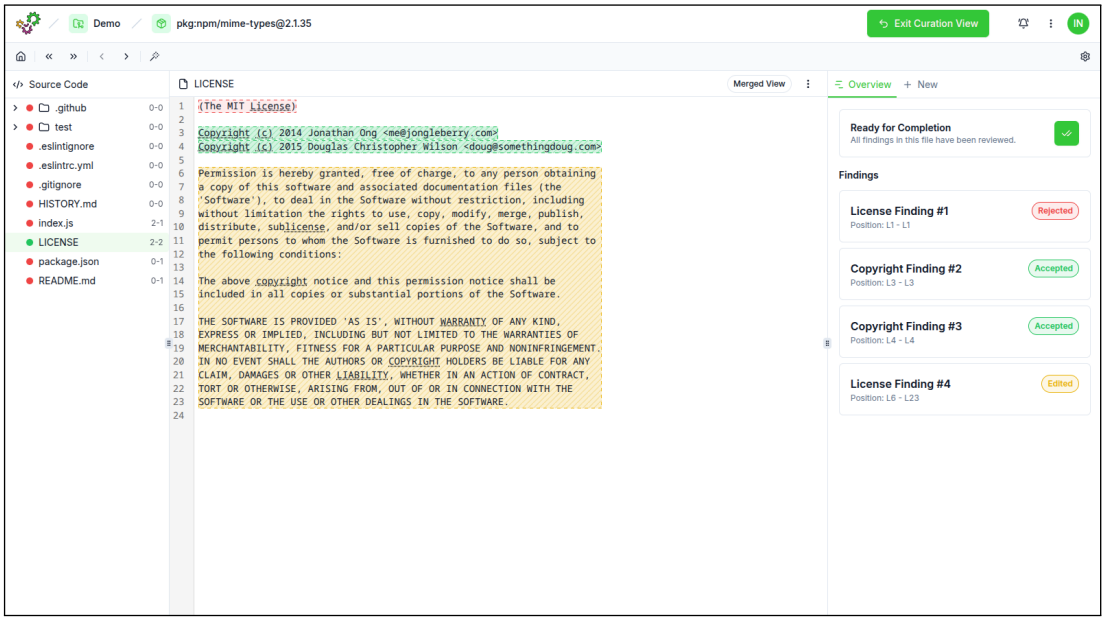

Figure 3: License-Clearing UI of a reviewed source code file (Merged View).

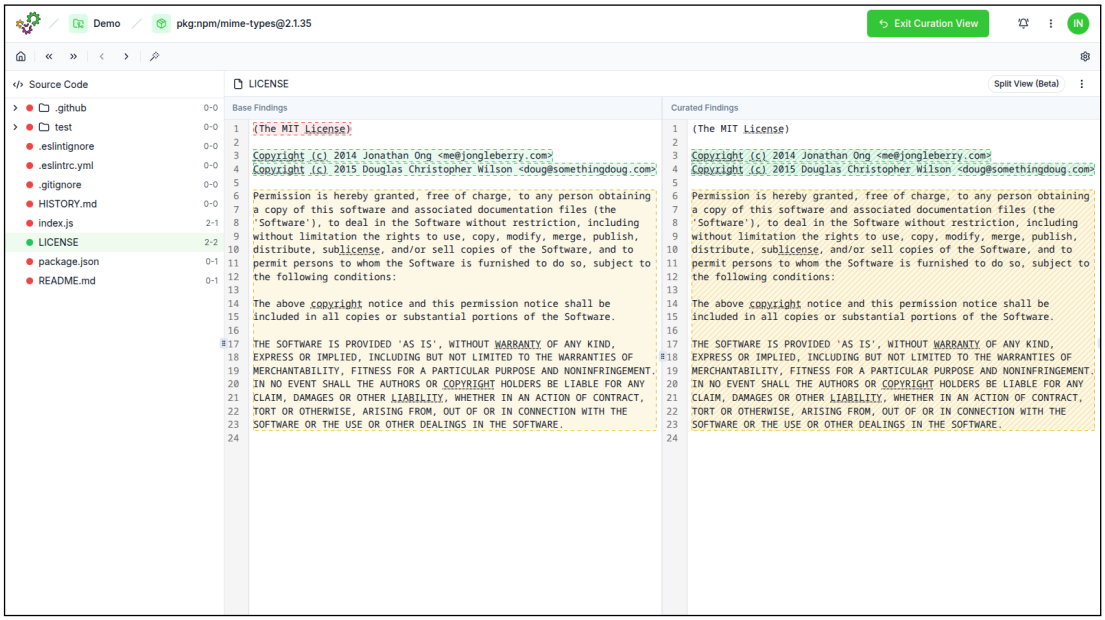

Figure 4: License-Clearing UI of a reviewed source code file (Split View). Action Panel is hidden for better visibility.

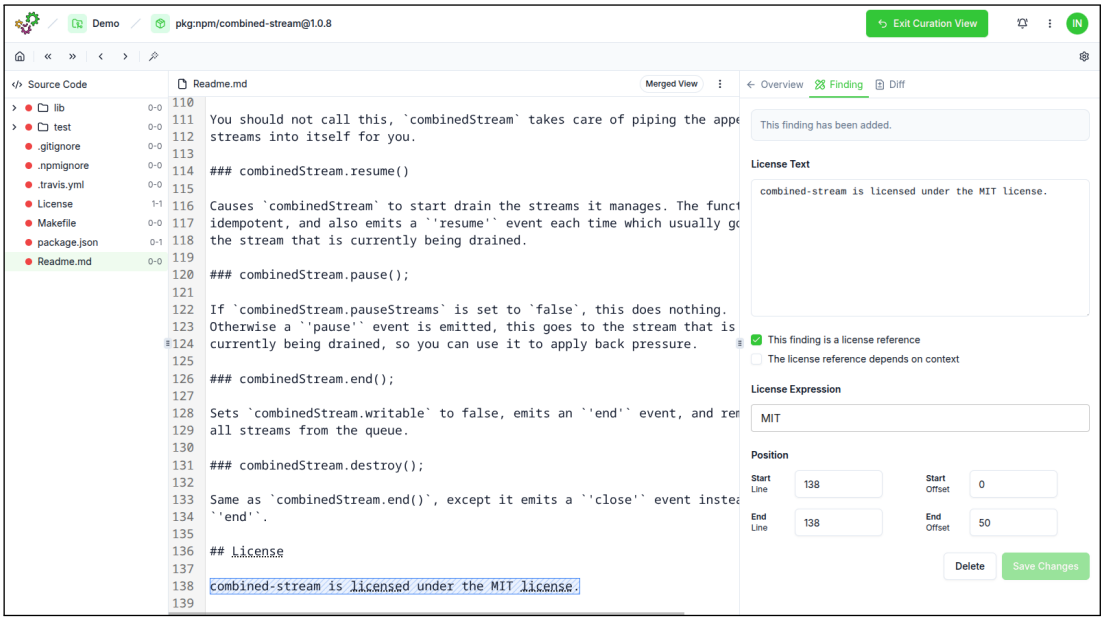

Figure 5: License-Clearing UI showing the details of an added finding.

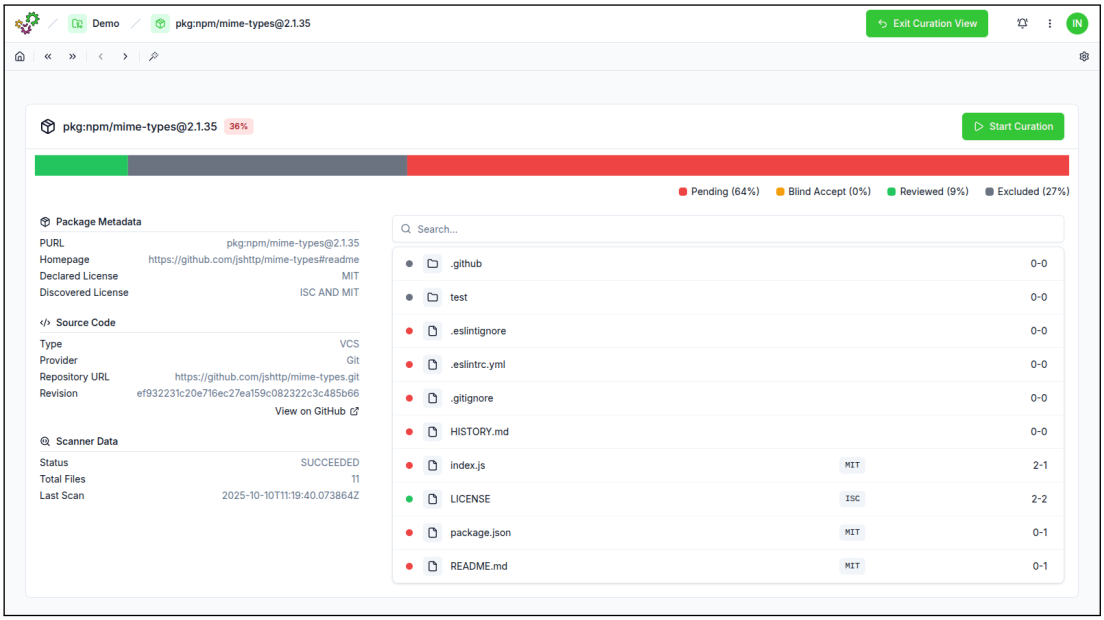

Figure 6: Clearing Overview page of a specific package.

C Packages for Evaluation

Package	Repository
tslib	github.com/Microsoft/tslib
@aws-sdk/types	github.com/aws/aws-sdk-js-v3
@smithy/types	github.com/smithy-lang/smithy-typescript
react	github.com/facebook/react
react-dom	github.com/facebook/react
lodash	github.com/lodash/lodash
chalk	github.com/chalk/chalk
inherits	github.com/isaacs/inherits
axios	github.com/axios/axios
next	github.com/vercel/next.js
commander	github.com/tj/commander.js
debug	github.com/debug-js/debug
inquirer	github.com/SBoudrias/Inquirer.js
@smithy/protocol-http	github.com/smithy-lang/smithy-typescript
safe-buffer	github.com/feross/safe-buffer
uuid	github.com/uuidjs/uuid
vue	github.com/vuejs/core
express	github.com/expressjs/express
numpy	github.com/numpy/numpy
org.slf4j:slf4j-api	github.com/qos-ch/slf4j
org.scala-lang:scala-library	github.com/scala/scala
graceful-fs	github.com/isaacs/node-graceful-fs
semver	github.com/npm/node-semver
requests	github.com/psf/requests
fs-extra	github.com/jprichardson/node-fs-extra
@smithy/node-config-provider	github.com/smithy-lang/smithy-typescript
once	github.com/isaacs/once
glob	github.com/isaacs/node-glob
readable-stream	github.com/nodejs/readable-stream
strip-ansi	github.com/chalk/strip-ansi

D Load-Testing Results

References

Beardsley, M. (2010). *Multi-tenant data isolation with PostgreSQL Row Level Security*. Amazon Web Services. Retrieved July 22, 2025, from https://aws.amazon.com/blogs/database/multi-tenant-data-isolation-with-postgresql-row-level-security/

Black Duck Software, Inc. (2025). *2025 Open Source Security and Risk Analysis Report*. Retrieved September 30, 2025, from https://www.blackduck.com/content/dam/black-duck/en-us/reports/rep-ossra.pdf

Gobeille, B. (2008). *The FOSSology Project: Overview and Discussion*. Retrieved September 30, 2025, from https://www.static.linuxfound.org/sites/default/files/lf_foss_compliance_fossology.pdf

Google. (2025). *deps.dev BigQuery dataset (v1)* (Data set; Version v1). Data set. Retrieved September 10, 2025, from https://docs.deps.dev/bigquery/v1/

Helmreich, M., & Riehle, D. (2012). Geschäftsrisiken und governance von open source in softwareprodukten. *HMD Praxis der Wirtschaftsinformatik*, *49*, 17–25. https://doi.org/10.1007/BF03340659

Schuberth, S. (2025a). *Make CycloneDX / SPDX SBOMs "first class" input to ORT* [Issue #9878 in *oss-review-toolkit/ort*]. GitHub. Retrieved September 17, 2025, from https://github.com/oss-review-toolkit/ort/issues/9878

Schuberth, S. (2025b). *Offer a well-defined but simple way to declare additional packages* [Issue #10182 in *oss-review-toolkit/ort*]. GitHub. Retrieved September 17, 2025, from https://github.com/oss-review-toolkit/ort/issues/10182

SOPHIST GmbH. (2016). *Schablonen für alle fälle*. Retrieved April 30, 2025, from https://sophistgroup.de/fileadmin/user_upload/Bilder_zu_Seiten/Publikationen/Wissen_for_free/MASTeR_Broschuere_3-Auflage_interaktiv.pdf

The ORT Project Authors. (n.d.). *Introduction*. OSS Review Toolkit. Retrieved October 1, 2025, from https://oss-review-toolkit.org/ort/docs/intro

Wolter, T. (2019). *A comparison study of open source license crawler* [Bachelor's thesis]. Friedrich-Alexander-Universität Erlangen-Nürnberg, Department of Computer Science. Retrieved October 1, 2025, from https://osr.cs.fau.de/wp-content/uploads/2019/08/wolter_2019.pdf

Wolter, T., Barcomb, A., Riehle, D., & Harutyunyan, N. (2023). Open Source License Inconsistencies on GitHub. *ACM Transactions on Software Engineering and Methodology*, *32*(5), 1–23. https://doi.org/10.1145/3571852