Friedrich-Alexander-Universität
Erlangen-Nürnberg

Crowdsourcing License and Copyright Scanner Corrections

Master Thesis

Handed in by: Eldrin Sanctis

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

Submission date: 18.10.2024

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

Declaration of originality

I confirm that this thesis is my original work, written independently and without external assistance. I have credited all sources where the work of others has been referenced. This thesis has not been previously submitted for examination or published. Additionally, the electronic version of the thesis is identical to the printed version.

Erlangen, 18.10.2024

License

Erlangen, 18.10.2024

Abstract

In today's software development world, open-source software is a big player, but it presents a number of challenges in managing and complying with a sea of licenses. Tools like ScanCode are used everywhere for detecting the licenses within the code, but they regularly fail at newer or modified licenses since the open-source landscape keeps on changing. The result is often missed or wrong licenses, putting organizations at legal risk.

This thesis also proposes a new approach that will make identification of licenses more accurate by adding a crowdsourcing feature to SCA Tool. Users can suggest licenses that might have been missed by the scanner, endorse the finding of an existing license, and finally provide comments to support decisions made on both suggestion and endorsement activities. Upon reaching the number of endorsements, the license is flagged as concluded but takes only full approval upon review and confirmation of the admin himself in the database.

By doing so, the methodology pools collective knowledge of the community in a quest towards keeping pace with a fast-changing world of OSS licenses. Human judgment along with automated scanning achieves a more accurate process and also makes sure that organizations can confidently stay compliant.

Contents

List of Figures

1 Introduction

1.1 Problem statement and the business value of accurate open-source license scanning

In today's world of software development, open-source components are found everywhere. They are crucial in making robust and effective applications. However, with such wide usage comes the big responsibility of management and following the rules laid down by different open-source licenses as described in (Wolter et al. 2022). Each license has its set of permissions, requirements, and restrictions that a developer should observe. Following are some of the reasons why open-source license scanning is important:

Legal compliance: To avoid legal issues, it is important to follow all the rules of open-source licenses. The failure to follow these may lead one to lawsuits resulting in fines or worse, the forced release of proprietary source code. These problems also will have serious financial implications on both individuals and organizations as mentioned in (Duan et al. 2017).

Security: As stated in (Domar Bolmstam et al. 2020), various licenses have specific requirements with respect to the handling and notification in cases of vulnerabilities. For such licenses, if they are misidentified, important requirements may be overlooked, which might compromise the security of the entire software. Proper or precise identification allows the security measures placed to be followed correctly.

Reputation management: Following open-source license rules helps an organization keep up its reputation among the developer community and its customers (HU et al. 2024); it alludes to ethical practices for respecting intellectual property and enhances elements of trust and credibility.

Intellectual property management: License findings helps in the management of intellectual property, including the correct scanning of the licenses, to ensure the proprietary code remains protected and that contributions towards open-source projects are properly credited as clearly explained in (Harutyunyan et al. 2019).

However, considering the fact that the most advanced tool, like ScanCode, identifies correctly known licenses in source code and gives hints, fast evolution is an issue in open-source licensing. New licenses appear, and updates to existing ones occur quite often, making the process of automated identification hard. So many diverse licenses are present, with each in its own complexity, requiring human intervention and interpretation of license texts. This indicates partial limitations of the automated tools themselves. This is why human judgment is typically necessary to accurately ascertain the set of licenses applying to the open-source components.

1.2 How crowdsourcing can improve license identification accuracy

In open-source license identification, crowdsourcing will take the accuracy on a completely different level. Due to the collective knowledge and contribution of a large pool of people, the crowdsourced license scanning will be much more accurate and extensive than anything performed previously. Following are some of the reasons supporting that:

Tapping into collective insight:

Diverse insights: Crowdsourcing by definition involves a wide community of users. It draws upon the wide range of experiences emanating from diverse users and thus is more apt to correctly identify licenses.

Expertise: Some contributors might have deep knowledge concerning particular licenses or codebases that could add value to the identification process in terms of precision.

Real-Time updates:

Real-time contributions: The community can update new licenses or even modify existing ones to help maintain the license database up to date.

Quick fixes: Mistakes brought to light by the automatic tools could be quickly fixed by the community, which means a higher quality.

Quality control:

Voting mechanism: The license identifications can be voted on by any user; this in turn creates a self-regulating environment whereby the most reliable piece of information rises to the top.

Admin review: Provides administrators with the ability to review and verify contributions for added validation.

Engaging the community:

Engagement encouragement: Platforms allow users to contribute and create a sense of community with shared responsibility which leads to increased activity.

Incentives and recognition: The contribution of more users brings an increase in quality and quantity for data by recognition or rewarding the top contributors.

Scalability:

Better volume management: Crowdsourcing can handle an increased workload much better when the number of open-source components go up than relying on automated tool competency with a meager team.

Building on the idea that crowdsourcing can significantly improve license identification accuracy, it is also important to consider other additional aspects of open-source compliance that pose challenges. Specifically, issues related to copyright corrections and user limitations in code and package analysis are crucial to address for a comprehensive solution.

1.3 Copyright corrections

While the focus of this thesis is on crowdsourcing license findings, it should be mentioned that copyright statements constitute yet another integral part of OSS compliance. Copyright statements (Fontana et al. 2008), or those which declare ownership over intellectual property, can sometimes be as painful to find and determine as the licenses themselves. Like licenses, these statements can be missed or incorrectly identified by automated scanners owing to inconsistencies in how they are formatted or embedded within the code.

Although the current SCA tool does not have a user interface for copyright statements and therefore cannot support crowdsourcing of corrections for the same, but the same principles being used as for license findings and validation can be extended to address copyright statements in the future. Correct identification is important in that such a copyright without accompanying license might indicate restrictions on reuse or modification in the code (Morris and Martin 2004). It would allow the community to develop this database, verify, and check these copyright statements for correctness of attributions.

1.4 User limitations in code and package analysis

One current limitation of SCA tool is that it does not have direct provision for the user to look at the source code or packages within the tool in order to analyze the license or copyright statements. Instead, it requires the user to track various components outside the tool, review them manually, and then perform the necessary license information analysis. This includes manual steps such as verifying whether a license detected is correct and then using SCA tool to submit input for example, add or vote on licenses.

Although this functionality is out of the scope of this thesis, the developers of SCA tool are currently working on integrating features into the tool whereby it will allow users to view and review code and packages within it. This will facilitate the process for license analysis, adding more efficiency for the user in contributing toward license corrections.

In the following chapters,

- A thorough **literature review** is conducted to explore existing research on open-source license scanning, crowdsourcing techniques, and the integration of community-driven approaches in software compliance tools.
- The functional and non-functional **requirements** of crowdsourcing system are established.
- The **architectural** framework is provided.
- **Design principles** that guided the crowdsourcing system is discussed which ensures the system is scalable and maintainable.
- Technical details of the **implementation** is explained comprehensively along with the use of modern technologies.
- System's performance is **evaluated** against the requirements and desired outcomes set earlier.
- Finally a **conclusion** with reflections on findings and suggestions for some future work is provided.

2 Literature review

2.1 Introduction to open-source license management

Today's software development relies heavily on open-source software (OSS), which offers advantages including quicker innovation, cost savings, and collaborative growth. Nonetheless, the difficulty of maintaining compliance with numerous, frequently intricate open-source licenses accompanies this extensive use. Accurately recognizing these licenses is necessary to maintain legal compliance, safeguard intellectual property, and steer clear of costly legal conflicts (Wolter et al. 2022). With these licenses becoming more complicated and dynamic, it's critical to have sophisticated tools and procedures for efficient license administration.

2.2 Existing software solutions for license scanning

To help developers and organizations manage open-source licenses; several tools and platforms are available. These tools generally use automated scanning to identify licenses within codebases. Here's a look at some of the most notable solutions:

2.2.1 ScanCode toolkit

- **Description:** ScanCode is an open-source toolkit widely used for analyzing and detecting licenses, copyrights, package manifests, and dependencies in software codebases. It supports comprehensive license detection across a broad range of open-source licenses. This tool plays a critical role in ensuring compliance with license obligations, making it indispensable in open-source projects (Al-Samman 2020, Phipps and Zacchiroli 2020).

- **Features:**
  - **License and copyright detection**: ScanCode can detect and identify licenses and copyrights in codebases, ensuring that software components comply with licensing regulations (Wagner 2023)
  - **Package manifest scanning**: The tool is capable of scanning package manifests to ensure the proper attribution of licenses (Kemppainen 2023)
  - **Rule-based engine**: ScanCode employs a sophisticated rule-based engine to detect variations in license texts, making it an effective solution for comprehensive software compliance audits (ScanCode-Wiki 2024)
- **Limitations:**
  - **Newly introduced or modified licenses**: The toolkit may not detect new or highly modified licenses promptly, as the open-source landscape evolves rapidly.
  - **Predefined patterns and rules**: Since ScanCode relies on predefined patterns and rules, it may miss variations in license texts that do not conform to standard templates (ScanCode-Wiki 2024)
  - **Legal interpretations**: While ScanCode can detect and identify licenses, legal interpretation is required to ensure full compliance. The tool does not replace the need for legal expertise (ScanCode-Wiki 2024)

2.2.2 FOSSology

- **Description:** FOSSology is another open-source license compliance toolkit that helps users scan and analyze open-source licenses within a codebase. It offers a web-based interface for managing and reporting on license compliance (FOSSology-Wiki 2024)

- The below **features** are mentioned in (**FOSSology-Wiki 2024**)
  - Scans and detects licenses
  - Generates and analyzes SPDX(Software Package Data Exchange) reports
  - Provides a web-based user interface for managing licenses
- **Limitations:**
  - Like ScanCode, FOSSology can struggle with licenses that don't fit predefined patterns or those that are more or less common (FOSSology-FAQ 2024)
  - The tool may require significant setup to get optimal results (FOSSology-Setup 2024)

2.2.3 Black Duck by synopsys

- **Description:** Black Duck is a commercial software composition analysis (SCA) tool that offers comprehensive license compliance and security vulnerability management for open-source components (Black-Duck-Team 2023)
- The below **features** are mentioned in (Black-Duck-Team 2023)
  - Automatic license identification and manages associated risks.
  - Provides continuous monitoring for license compliance.
  - Integration with DevOps pipelines for seamless operations.
- **Limitations:**
  - As a commercial product, it comes with a high cost, making it less accessible to smaller organizations or individual developers.
  - Being a closed-source solution, it offers limited customization and transparency

2.3 Use cases for license scanning tools

License scanning tools are used in various scenarios to ensure compliance with open-source licenses:

- **Enterprise software development:** Large companies often integrate tools like Black Duck into their development workflows to maintain compliance throughout the entire software lifecycle. This ensures that all components meet licensing requirements from development through to deployment (Black-Duck-Team 2023)
- **Open-Source projects:** Open-source projects that are managed by tools like ScanCode or FOSSology help in verifying that all contributions adhere to the project's licensing rules (Fendt and Jaeger 2019)

2.4 Concluded license and the role of SPDX

2.4.1 Understanding concluded licenses

A key concept in managing open-source licenses is understanding "concluded licenses". This term refers to the final licenses determined for a piece of software after considering both the licenses declared by the author and those discovered through automated scanning.

- **Declared licenses:** These are the licenses explicitly stated by the software's author or distributor (Samman et al. 2024)
- **Discovered licenses:** These are identified by tools like ScanCode or FOSSology based on the analysis of the codebase (Samman et al. 2024)
- **Concluded licenses:** After evaluating both declared and discovered licenses, and incorporating any additional input from users or administrators, the concluded license represents the final legal license governing the software (SPDX-Concluded-License-Info 2023). Getting this right is crucial, as mistakes can lead to legal issues and intellectual property disputes.

2.4.2 The role of SPDX in license management

SPDX (Software Package Data Exchange) is a standardized way to document the license of open-source software. It helps in recording, exchanging, and analyzing license information across different platforms.

- **SPDX license list:** This is an up-to-date list of recognized open-source licenses, each with a unique identifier to simplify identification and reporting (SPDX-License-List-Info 2022)
- **SPDX documents:** These provide a standardized format to describe licenses, copyrights, and other relevant details about software packages, aiding in compliance reporting and due diligence (SPDX-Document-Info 2022)
- **Concluded license field:** This field in the SPDX specification records the final license determination after considering all the relevant data. It is vital for compliance workflows as it represents the legal status of the software package (SPDX-Concluded-License-Info 2023)

2.5 Importance of Crowdsourcing License and Copyright Scanner Corrections

Addressing limitations of automated tools

Current license scanning tools, while useful, often struggle with the complexities of modern open-source licenses, especially as new ones emerge and existing ones evolve. This thesis proposes a crowdsourcing approach to complement these tools, bringing in human expertise to address these limitations.

- **Human input:** Automated tools may miss context-specific nuances in license texts. By allowing users to contribute and verify license information, this approach enhances accuracy.
- **Dynamic license landscape:** Crowdsourcing helps keep the license database up-to-date, leveraging community knowledge to address the constantly evolving open-source ecosystem.

Enhancing license identification accuracy

Accurate license conclusion is essential for legal compliance. By incorporating crowdsourced input where users can vote on and contribute to license identification, this solution improves the reliability of concluded licenses. This helps ensure the final determination reflects a broad base of knowledge and is less prone to errors. This process also includes mechanisms for flagging and managing troll users who contribute false or malicious license information, ensuring the integrity of the crowdsourced data.

Use cases for the proposed solution approach

- **Small and medium enterprises (SMEs):** For SMEs that may not afford expensive compliance tools, crowdsourcing offers a cost-effective way to achieve accurate license compliance.
- **Open-source project maintainers:** These projects benefit from community contributions to maintain compliance, especially when dealing with multiple contributors.
- **Compliance teams:** Integrating crowdsourced data with existing tools adds an extra layer of verification, helping legal teams manage compliance more effectively.

2.6 Problem-solving and relevance

Solving the incomplete identification problem

The proposed approach addresses the issue of incomplete or inaccurate license identification by combining automated scanning with crowdsourced human input. This hybrid approach fills the gaps left by existing tools and provides a more comprehensive solution.

Relevance in the current landscape

Growing OSS adoption: As more organizations adopt open-source software, the need for accurate license compliance becomes increasingly important. The proposed solution approach is timely, helping companies use open-source components confidently while staying compliant.

Legal risks and compliance: With tighter regulations around software use, precise license identification is crucial. The proposed solution offers a solution to these challenges, helping organizations navigate the complexities of open-source licensing with greater assurance.

2.7 Drawbacks of crowdsourcing approach

Reliance on user contributions:

- The accuracy of the system depends heavily on active community participation.
- A lack of contributions or malicious input could reduce the system's effectiveness.

Potential for bias or errors:

- Crowdsourcing can introduce bias if users overwhelmingly agree without sufficient validation.
- Incorrect contributions or votes might still influence the outcome until reviewed by an admin.

Admin workload:

- Admins must review and approve/disapprove licenses, which could become time-consuming with large numbers of contributions.

Examining the state of open-source licensing management today and the tools available for license scanning shows that although these solutions provide useful features, they are severely constrained. Automated solutions alone are unable to adequately handle the issues posed by the complexity of modern licenses and the quick evolution of the open-source ecosystem. By including the community's collective knowledge, crowdsourcing has emerged as a promising method to improve license identification accuracy. But there are some disadvantages to this approach as well, namely the dependence on user input and possible biases. To move forward, it is important to define the specific requirements that a new solution must meet to effectively improve upon existing methods.

3 Requirements

In this chapter, we will go into more detail about the functional and non-functional requirements, taking into account the positives and negatives found in our literature review. This will lay the foundation for developing a system that not only improves the license findings through crowdsourcing but also mitigates the challenges or difficulties involved, strengthening open-source compliance procedures in the process.

3.1 Functional requirements

The functional requirements will explain what kind of behaviors and functions the system must have in order to accomplish the goals of the thesis. These are required by SCA Tool to crowdsource and maintain the licensing information efficiently.

- **User contributing a discovered license**

  Title: Add a detected license

  As a: User

  I would like to add a license that the scanner missed to the discovered licenses

  So that: I can contribute to the community.

  Acceptance criteria:

  – Users can proceed with the licenses found section.

  – The user can add a new license and insert an optional comment explaining the license.

  - The additional license gets added and is considered temporary until it obtains a net difference of 10 contributions.

- **User confirming a discovered license**

  Title: Confirming a license found

  As a: User

  I would like to verify a detected license that I agree with

  So that: It may achieve the threshold to become a temporary concluded license.

  Acceptance criteria:

  - Users can view the list of discovered licenses.
  - Users can click an upvote button next to a license they've discovered.
  - When upvoting, a comment can be included supporting the reason for the positive vote.
  - The system registers the confirmation (upvote) and updates the vote count along with the net difference.
  - If the net difference of confirmation minus disapproval reaches 10, the license status becomes temporarily concluded.

- **User rejecting a discovered license**

  Title: Disapproves a found license

  As a: User

  I would like to disapprove or reject a detected license that I disagree with

  So that: The community knows my opinion about the license.

  Acceptance criteria:

  - Users can view the list of discovered licenses.
  - Users can click a downvote button next to a discovered license.
  - Users can add a comment explaining the reason for the downvote.
  - The system registers the disapproval and updates the vote count and net difference.

- **Admin reviewing and approving temporary concluded licenses**

  Title: Review and approve temporary concluded licenses

  As an: Admin

  I would like to review licenses that have reached the upvote threshold and are marked as temporarily concluded.

  So that: I can ensure that the concluded license is accurate and valid.

  Acceptance criteria:

  - Admin can view licenses that have a net difference of 10 upvotes.
  - Admin can approve or disapprove the license.
  - Approved licenses are marked as concluded.
  - Disapproved licenses are flagged with a status "D" for Disapproved.

- **Admin marking a user as a troll**

  Title: Mark user as troll

  As an: Admin

  I would like to identify and flag users who add nonsense licenses or malicious input

  So that: I can maintain the integrity of the system.

  Acceptance criteria:

  - Admin can review the history of a user's contributions.
  - Admin can flag a user as a troll if they repeatedly add nonsense licenses.
  - Flagged users are restricted from adding new licenses or voting.

- **User viewing and adding concluded licenses**

  Title: View and add concluded licenses

  As a: User

  I would like to view concluded licenses or accept suggested conclusions one by one or all together

  So that: I can use accurate, community-validated license information.

  Acceptance criteria:

  - Users can navigate and access the concluded licenses section.
  - Users can view and accept from a list of licenses that have been marked as suggested licenses for conclusion.

- **User commenting on a license during interaction**

  Title: Comment on a license

  As a: User

  I want to add comments when I add, approve, or disapprove a license

  So that: I can provide additional context or information about the license.

  Acceptance criteria:

  - Users can add a comment when contributing a new license.
  - Users can add a comment when confirming a discovered license.
  - Users can add a comment when disapproving a discovered license.

  - Users can add a comment when confirming a concluded license.
  - Comments are visible to admins.
  - Comments are sanitized to remove any personal information, such as email addresses.

- **Admin removing a license contribution**

  Title: Remove a license contribution

  As an: Admin

  I would like to remove a license contribution that is deemed inappropriate or incorrect

  So that: The integrity of the license data is maintained.

  Acceptance criteria:

  - Admin can view a list of user-contributed licenses.
  - Admin can mark a license contribution with a status "D" meaning Disapproved.
  - Disapproved licenses are no longer visible to regular users.

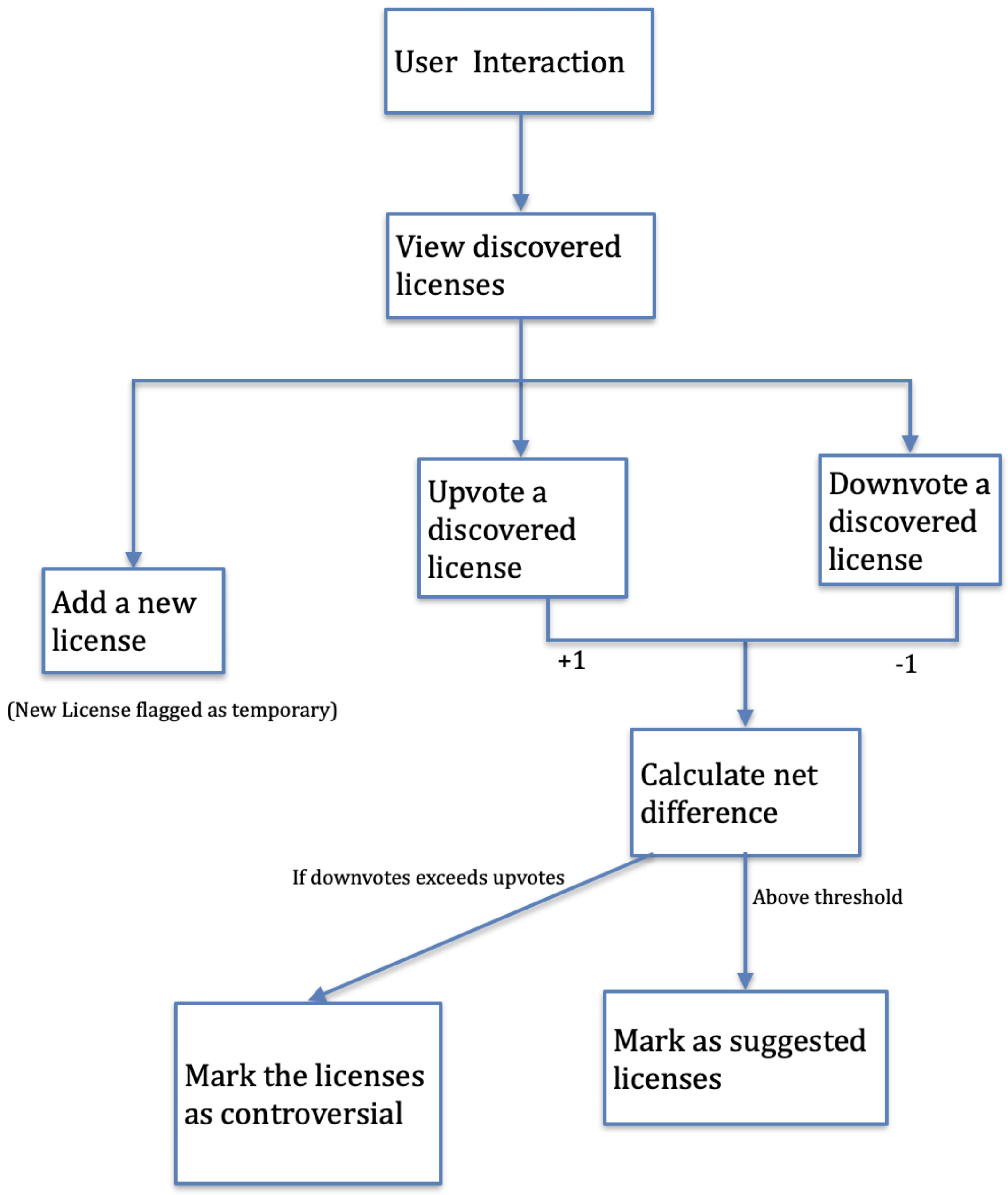

User Interaction
View discovered licenses
Add a new license
Upvote a discovered license
Downvote a discovered license
(New License flagged as temporary)
+1
-1
Calculate net difference
If downvotes exceeds upvotes
Above threshold
Mark the licenses as controversial
Mark as suggested licenses

Figure 1: User Interaction

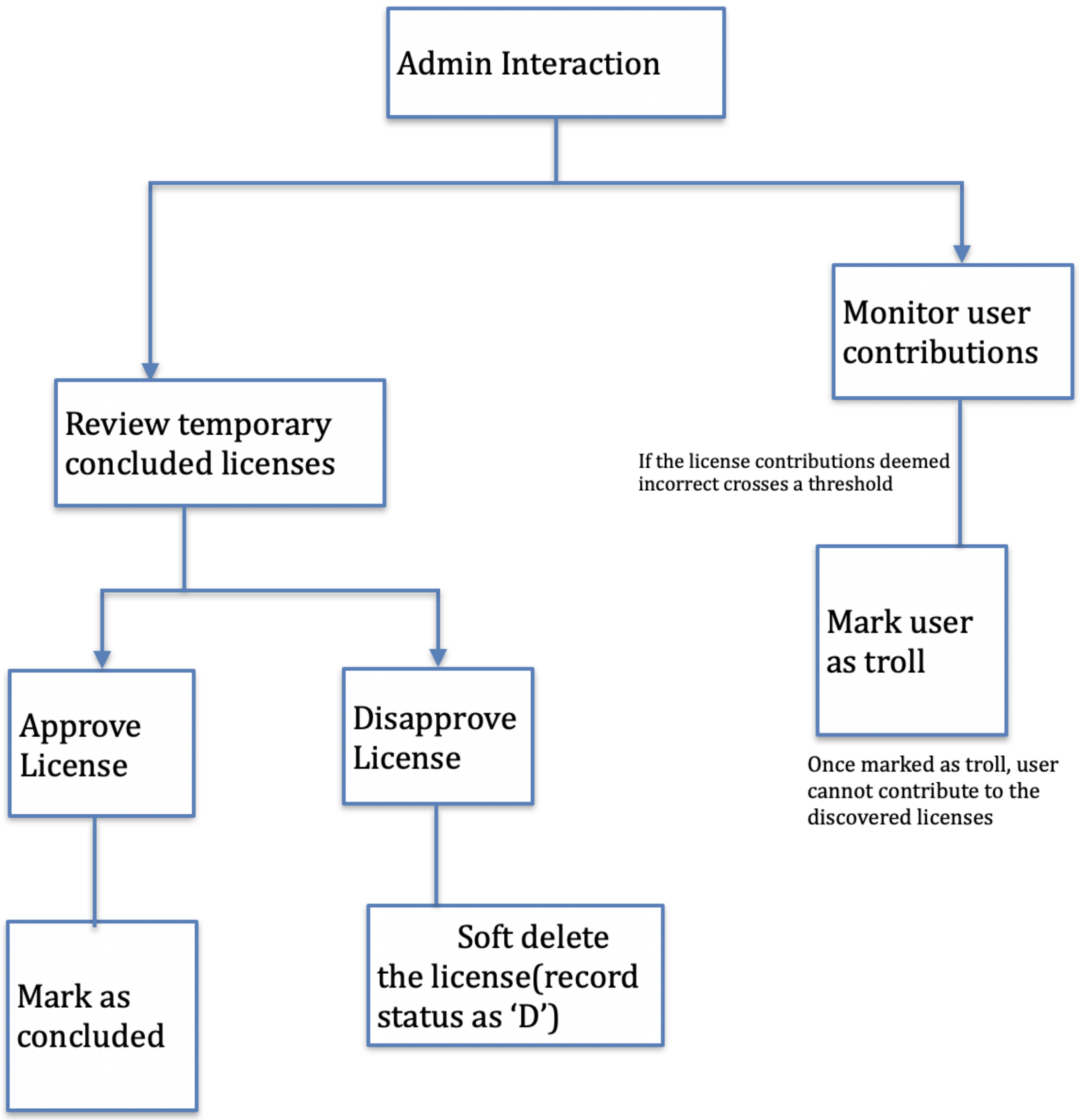

Admin Interaction
Review temporary concluded licenses
Monitor user contributions
If the license contributions deemed incorrect crosses a threshold
Approve License
Disapprove License
Mark user as troll
Once marked as troll, user cannot contribute to the discovered licenses
Mark as concluded
Soft delete the license(record status as 'D')

Figure 2: Admin Interaction

3.2 Nonfunctional requirements

Non-functional requirements explain the quality, performance, and system operational characteristics that will ensure that the designed system runs efficiently, effectively, and securely.

- **System performance and scalability**

  The system should be built to efficiently handle increasing license and user numbers.

  Details:

  - It should allow for more user submissions and votes without substantial delays.
  - For getting the user contributions, votes, basically license information; database has to be optimized.
  - Even with peak traffic, the user interface should stay responsive.

- **Security and access control**

  The system must safeguard user data from unwanted access.

  Details:

  - Licenses can only be reviewed, approved, or disapproved by administrators and authorized users.
  - Data has to be encrypted regardless whether it is in transit or rest.

- **Usability and user experience**

  Both users and administrators should find the system easy to use.

  Details:

  – The information should be organized/arranged clearly and the interface should be easy to use.

  – Tooltips, help documents, and guidelines should be offered to help users contribute licenses and vote.

- **Reliability and availability**

  The system must be reliable, fault-tolerant, and have little downtime to maintain high availability.

  Details:

  – It should recover fast from failures, ensuring fault tolerance.

  – To avoid data loss, backups should be performed regularly.

- **Maintainability and extensibility**

  The system should be easily maintainable and extensible for future updates and improvements.

  Details:

  – The codebase should be thoroughly documented, with clear comments and developer documentation.

  – The design needs to be adaptable in order to allow upgrades or new features to be added without affecting already existing functionality.

  – Automated testing should ensure that new modifications do not cause bugs or regressions.

In summary, this chapter has covered all the essential functional and non-functional requirements necessary for building our proposed crowdsourcing system. The necessary functions that allow users to add new licenses, validate or reject current licenses, and allow administrators to examine and accept these contributions while preserving the integrity of the system have been outlined. In order to guarantee that the system is reliable and easy to use, we have also highlighted the significance of system performance, security, usability, reliability, and maintainability. These requirements provide a solid basis for creating a system that tackles the issues we've discovered and enhances license findings through crowdsourcing, paving the way for the architectural design in the following chapter.

4 Architecture

In this chapter, we will explore the architecture of the crowdsourcing solution, detailing how the various components will work together to meet the needs that were identified earlier. We'll explore the design decisions, the technologies that will be used, and how we'll address the challenges highlighted in our requirements.

4.1 High-level overview

The architecture includes the following components:

- **User Interface (UI) for license contribution and voting**
- **Crowdsourcing logic layer**
- **Review and validation by admin**
- **Data storage for contributions and voting**

Every element has a distinct function in overseeing and verifying user contributions.

4.2 User Interface (UI) for license contribution and voting

Description:

Users interact with the system using the frontend. It includes options for submitting new licenses, voting on current ones, and providing feedback.

Key features:

- **License contribution form**: Users can add new licenses and leave comments to describe their contribution.
- **Voting interface**: Enables users to accept or reject licenses while adding required contextual remarks.
- **View/Add concluded licenses**: Users can view concluded licenses and accept suggested conclusions individually or collectively.
- **Status indicators**: Displays the current status of licenses (e.g., temporarily concluded, concluded)

Technology:

TypeScript.js: It is used to create an interactive and adaptable user interface.

4.3 Crowdsourcing logic layer

Description:

The backend layer processes user contributions, votes, and manages temporary licenses.

Key functions:

- **License submission handling**: Validates new licenses and labels them temporary.
- **Voting management**: Updates the net difference for each license and tracks the vote counts.
- **Threshold monitoring**: This module determines when a license has received +10 approvals, for example, and modifies its status correspondingly.
- **Comment management**: Saves comments with votes and ensures personal information is secure.

Technology:

Java with Spring framework for backend processing and business logic.

4.4 Admin Intervention for review and validation

Description:

Admins analyze licenses that have been temporarily concluded and control user contributions from within the database.

Key features:

- **License review**: Administrators review temporary licenses based on vote counts and comments.
- **Approval/Disapproval actions**: Admins update the license status in the database, marking them as approved or disapproved.
- **User management**: Admins can edit flagged users' database records and restrict their ability to contribute or vote as needed.

Technology:

Database management system: Administrators utilize SQL queries and database tools to review and manage users.

4.5 Data storage for contributions and voting

Description:

This component stores all data related to user contributions, voting, and license statuses.

Key features:

- **License database**: Stores licenses with current status, vote counts, and comments.

- **User contributions and voting records**: Tracks all user contributions and votes for auditing and review.
- **Audit logs**: Records admin actions to ensure openness and accountability.

Technology:

PostgreSQL: A relational database for structured data storage.

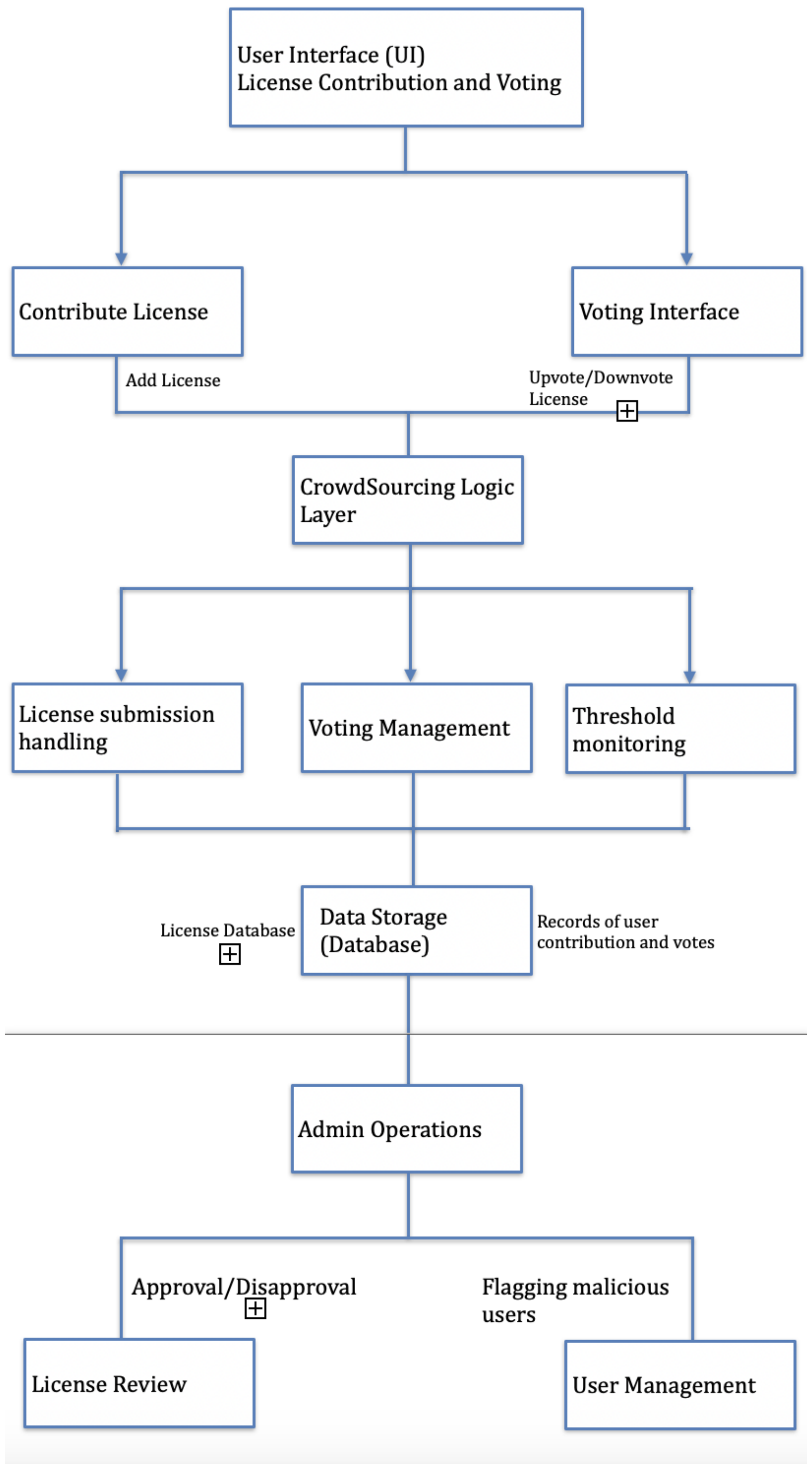


Figure 3: Architecture

To sum up, this chapter described the architecture for our crowdsourcing solution aimed at improving open-source license findings. The system was divided into four main components where each part plays a crucial role; users interact through a TypeScript.js frontend, their inputs are handled by a Java Spring backend, admins perform reviews using database tools, and everything is stored in a PostgreSQL database. By detailing how these components work together, we've set the foundation for implementing the crowdsourcing system.

5 Design principles

5.1 Overview

With the architectural framework in place, several software design patterns and ideas were used in the creation of the crowdsourced license findings system for **SCA Tool** in order to guarantee the solution's scalability, maintainability, and flexibility. The frontend, backend, service, and repository layers of the system are all developed using these patterns and concepts to create a clean design that will allow for future updates without requiring a lot of effort.

5.2 Separation of Concerns (SoC)

Principle: The separation of concerns (Noda and Kishi 2001) design principle recommends breaking up a software system into discrete pieces, each of which deals with a unique responsibility or concern.

Application in the crowdsourcing system:

- The frontend (user interface), backend (business logic and API), and repository (database interactions) are the three main layers that make up the system. Each layer is responsible for overseeing a certain area of the application.
- The **Frontend** focuses primarily on showing data to the user and sending/receiving data from the backend.
- The **Backend** focuses on processing the data, implementing business rules, and managing the information flow between the frontend and database.

- The **Repository** abstracts away the specifics of SQL queries and data persistence, focusing only on database operations.

Benefits:

- Because each layer may change on its own, it is simpler to update or refactor without impacting the system as a whole.
- Because developers may concentrate on specific issues without having to deal with the complexity of other portions of the system, maintainability is improved.

5.3 Single Responsibility Principle (SRP)

Principle: According to the single responsibility concept, a class or module should only have one responsibility and one reason for change (Rana and Khonica 2021)

Application in the crowdsourcing system:

- Every class in the system has a specific goal in mind when it is built.
  - The **GovernanceController** handles HTTP requests and passes processing to the service layer.
  - Business logic, such as addDiscoveredLicense and voteDiscoveredLicense, for managing licenses and votes is contained in the **Service Layer** (DependencyServiceImpl, LicenseServiceImpl).
  - The **Repository Layer** (ComponentRepository) is the only entity that can communicate with the database.

Benefits: SRP makes the system's parts more unified and task-specific, which facilitates testing, debugging, and system extension. The only class or module that has to be changed when a specific system feature changes is the pertinent one.

5.4 Dependency Injection

Principle: Dependencies for an object can come from external sources instead of generating them internally thanks to the design pattern known as dependency injection (Yang et al. 2008)

Application in the crowdsourcing system:

- Dependency injection is a feature of the Spring framework, used in this system that enables the injection of components into controllers and service classes, including repositories and services.
- Services like **UserService**, **DependencyServiceImpl**, and **LicenseServiceImpl** are injected by the **GovernanceController**. In a similar vein, service classes manage database interactions by injecting repository objects .

Benefits:

- Dependency injection improves testability and maintainability by separating an object's behavior from its construction. During unit testing, for instance, services and repositories might be replaced with mock implementations.
- It promotes loose coupling between classes, which facilitates system refactoring and extension without requiring tight dependencies.

5.5 Repository Pattern

Pattern: By encapsulating the logic for accessing and altering data from the underlying data source, the repository pattern offers a means to abstract data access.

Application in the crowdsourcing system:

Votes, licenses, and governance component retrieval database operations are abstracted by the **ComponentRepository** class. Complex queries are handled via methods like **findAllByDistUnitIdAndDistUnitVersionAndUserId** to retrieve relevant data from the database without disclosing SQL query information to the rest of the application.

Benefits:

- Because modifications to the underlying database structure have no impact on the service or controller layers, abstracting the data layer makes system maintenance easier.
- Because many services can use the same repository methods without duplicating database access logic, it encourages reusability.

5.6 RESTful API Design

Pattern: Networked application design uses the architectural pattern known as representational state transfer (REST). It depends on client-server, stateless communication using common HTTP techniques (Li et al. 2016)

Application in the crowdsourcing system:

- **GovernanceController's** backend APIs use HTTP methods like GET and POST to fetch components and add licenses and submit votes, respectively, in accordance with RESTful principles.
- Every API endpoint functions statelessly and is assigned to a particular resource (such as components or licenses), enabling clients to communicate with the system using common web protocols.

Benefits:

- RESTful APIs are popular and simple to integrate with many types of clients. Since more services and customers can communicate with the API without requiring a tight connection, they guarantee that the system will continue to be scalable.
- REST APIs' stateless design makes it easier to distribute the system over several servers and improves its scalability.

5.7 DTO (Data Transfer Object) Pattern

Pattern: Reducing the amount of data transmitted between the client and server is possible by using the DTO pattern to move data across levels of an application while making sure that only pertinent data is passed (Pantaleev and Rountev 2007)

Application in the crowdsourcing system:

A DTO that contains license data (declared, discovered, and concluded licenses) and relevant flags (such as vote and concluded license flags) is the **GovernanceComponent** class. Only the essential data is included in the lightweight format that is returned with this object to the frontend.

Benefits:

The system improves data transfer by lowering the payload size and enhancing performance through the use of DTOs. Furthermore, it offers an extra degree of abstraction, guaranteeing that database structures and underlying models remain hidden from the client.

5.8 Strategy for Extensibility

Principle: The system is designed with extensibility in mind, allowing for future improvements without requiring large architectural modifications (Simons et al. 1999)

Application in the crowdsourcing system:

The modular design of the layers (frontend, service, repository) and the use of patterns like dependency injection and the repository pattern allow for easy modification. For instance, the controller and repository levels remain unaffected if new license types or voting procedures are incorporated into the service layer.

Benefits: This tactic guarantees that the system will continue to be flexible as new needs or features arise. Developers can incorporate new services or increase functionality with little effect on already-built components.

5.9 Why do these design principles and patterns work for crowdsourcing license correction system

- **Scalability**: The crowdsourcing system can accommodate more users and components as the project grows because of the separation of concerns and usage of RESTful APIs.
- **Maintainability**: Every component of the crowdsourcing system is easier to maintain when the single responsibility principle is followed because modifications to one do not affect modifications to others.
- **Security**: By abstracting delicate database activities, the repository pattern makes sure that direct access to the data is verified and regulated. Additionally, stateless client-server interactions are guaranteed by RESTful principles, which improve security.
- **Testability**: The system is very testable thanks to dependency injection and the DTO pattern. The use of lightweight DTOs guarantees that the logic stays testable without relying on complex database interactions, and components may be mocked simply during testing.
- **Flexibility**: The crowdsourcing system's modular design and layer decoupling enable for future improvements without requiring extensive overhaul.

6 Implementation

6.1 Overview

With the design principles in place, a layered architecture of frontend, backend (controller, service and repository layer) (O'Reilly-Team 2024) is used to implement the the crowdsourcing license correction functionality. This chapter explains the technical details of the implementation, describing the primary functions and how all these layers communicate with each other. The implementation makes use of modern frameworks such as React (TypeScript) for the frontend, SpringBoot (Java) for the backend and PostgresSQL for the database.

6.2 Add license functionality

Users are allowed to add new licenses for a given package. The client, backend and repository layers work together to validate, process and save the newly added license. The following steps outline how this feature is implemented in each layer.

Frontend implementation

The frontend provides a input to the user who wants to enter a new license and submit this data to the back end for validation and storage. The primary UI element here is a dialog box where the user provides license information and comments. Following form submission, the frontend sends the backend a POST request containing the license information.

Feedback is given to the user to let them know whether the operation succeeded or failed. Following submission, the user interface shows the licenses along with the recently added data.

Controller layer

The backend controller class **GovernanceController** handles the POST request coming from the frontend. Whenever a new license is submitted, the controller executes the **addLicense** method, which retrieves the package ID, license text and comments from the user interface along with the user details and then passes them as input values to the service layer.

To confirm whether the user has the right identification and is authorized, the controller additionally authenticates session data. The frontend receives an HTTP 200 response if the check is successful. The controller sends an error message accordingly in the event of validation or processing problems.

Service layer

The service layer class **DependencyServiceImpl** is responsible for the actual logic for adding a discovered license. Initially, it checks whether the discovered license in the package is already present in the system. If the package does not contain the license, then the relevant information is passed on to the repository layer (package Id, discovered license, user ID, a comment, and the initial status flag set to "temporary.")

If the operation was successful, the service will send the message to the controller which in turn will forward this information back to the frontend. Thus, the service layer makes sure that only valid or legitimate data is handled and saved.

Repository layer

The repository layer class **ComponentRepository** manages communication with the database. A method **addLicense** for adding a license is given, which involves keeping the newly found license in the relevant table **user_license_findings**. Using this method, the **user_license_findings** table is updated with the package ID, license, user ID, comment, timestamp, and "temporary" flag.

The repository layer spares the service and controller levels from having to communicate directly with the database by abstracting database operations. The division of responsibilities enhances the system's maintainability and scalability, since any changes to the database structure or queries are handled at the repository level without interfering with other components.

6.3 Voting mechanism implementation

One essential component of the crowdsourced license correction system is the **voting mechanism**. Users can vote on licenses they find, giving their approval (upvoting) or disapproval (downvoting) to a license that they themselves have added along with licenses that another user has submitted. The voting process not only collects feedback from the community but also determines whether a license should move from a discovered status to a concluded status for the community based on a threshold of upvotes. This section describes how this voting process will be put into practice, including how to handle all system data flow, update the finished license, and flag votes.

Frontend implementation

Voting user interface The user interface places the two primary icons (up and down) beside each license. The system displays a dialog window to confirm the vote when a user selects one of these buttons. Users can write a statement in this dialog box to explain why they are casting their vote, and that message is sent to the backend with the vote.

The vote data is sent by the frontend as a POST request to the backend following submission. The user interface is updated with the backend response, displaying the license's current status as well as any changes made by the vote (such as shifting it to a concluded status).

Controller layer

When users cast their votes, the GovernanceController's **voteLicense** method responds to the incoming POST requests from the frontend. After the voting data has been processed, it is sent to the service layer for additional business logic.

The following are overseen by the controller:

- Extracting the package ID, found license, vote type, and user details from the request and using them as input.
- Verifying the user's identification and confirming their eligibility to cast a vote.
- Sending the vote data to the service layer so that it may be processed and returned to the frontend with a suitable answer (success or failure).

Service layer

The vote processing and license status updating are handled by the service layer class **LicenseServiceImpl**. The method **voteDiscoveredLicense** handles the voting logic, comprising the following critical steps:

1. **Validating the vote**: The service first checks whether the user's vote and the most recent vote previously done by the same user for the exact same license, package and distribution unit and version is the same. If it is not equal, the system proceeds with storing the new vote. If they have, the system will update their previous vote.
2. **Storing the vote**: The license information is stored in the database along with the vote type i.e. up or down. The user's ID and the package details are also stored to track the vote.
3. **Counting votes**: The service counts the number of upvotes and downvotes to determine the net votes for the license after saving the vote. The net vote count is important so as to confirm the change in license's status from discovered to concluded.

4. **Updating the license status**: : The license is marked as "temporarily concluded" and moved to **suggested licenses** for the community if the net vote difference is more than a predetermined threshold (for example, +10 upvotes). This status indicates that the license is pending further assessment by the system or an administrator before being formally marked as finished.

Repository layer

When a vote is cast, the repository class **ComponentRepository** inserts the vote details into the **user_license_votes** table. This table tracks each vote, associating it with the specific package, license, and user.

In addition, the repository handles the complex query operations required to determine the overall upvotes and downvotes for a particular discovered license. By asking this question, the system makes sure that if a license has met the requirements to be marked as suggested license.

6.4 Component loading implementation

The **Component loading** feature is essential for providing the user with up-to-date governance data , such as declared, discovered, and concluded licenses for each package or component. This section will describe how the frontend receives the data from backend that has been retrieved and enhanced with additional governance information.

Controller layer

Retrieving governance components for a particular distribution unit and version is handled by the **GovernanceController's components** method. This method:

- Obtains the distUnitId and distUnitVersion from the incoming request.
- Calls the service layer to retrieve the distribution unit's component parts.

Following component retrieval, the controller adds governance data to each component, such as:

- **Declared license**: Taken from the metadata of the package.
- **Discovered license**: Based on the licenses submitted by the community, flagged and voted on as well.
- **Concluded license**: This information is included if the license has reached suggested findings(temporary concluded or concluded) status.

Service layer

The logic to retrieve and enhance the component data is implemented by the **DependencyServiceImpl** class. After interacting with the repository layer to obtain the raw data, each component receives governance information before the outcome is sent back to the controller.

Key responsibilities of this service layer include:

- **Retrieving component data**: The repository is used by the service layer to retrieve component data from the database.
- **Enriching data**: The service layer calls helper methods to enrich each component with governance groups for declared, discovered, and concluded licenses. These groups provide the licenses context, including information on which license family or categories they fall under.
- **Handling license flags**: In order for the user interface to show the user this information, the service additionally makes sure that any flags connected to concluded licenses are returned with the license.

Repository layer

The repository layer class **ComponentRepository** handles the complex database queries to retrieve the relevant component data. The **findAllByDistUnitIdAndDistUnitVersionAndUserI** method is responsible for fetching the components, including:

- Declared licenses.
- Discovered licenses, along with their associated votes and flags.
- Concluded licenses and their respective flags.

To offer a full view of each component's governance state, this approach builds a big query that aggregates data from several tables in the database, including mainly components, packages, **user_license_findings**, **user_license_votes** and others.

After the data is retrieved by the repository, it is transferred to the service layer for further enrichment before being sent to the controller.

Frontend handling

The extracted components are shown on the governance tab in the user interface. Each package contains:

- The **declared license**
- The **discovered license**
- The **concluded license** (blurry indicating as suggested license, upon double clicking and submitting, it performs both the add license and vote license mechanism.)

This provides a thorough picture of the licenses with their status for each packages, allowing the users to perform any of the described functions above.

6.5 User management and flagging by admin

Administrators can use the system's features for **user management** and **license flagging**. These tasks are done manually in the database by changing particular flags, as there isn't a dedicated admin interface. This section explains the database operations that administrators use to manage users and licenses.

Admin user management

When a user is suspected of being malicious or a troll (for example, by continuously providing false or irrelevant licenses), the administrator has the ability to take action by changing the person's status directly in the database. The steps below describe how this operates:

- **Flagging a User**: The administrator modifies the user's record in the database by altering its **record status to 'D'**.
- **Restricting User Actions**: After a user is reported, the system verifies their eligibility before allowing them to vote or contribute licenses. When someone is reported as a troll i.e. their record status 'D', the system stops their contributions from being seen by the community.

Managing license flags

There are several uses for the license-related flags:

- **Temporary flag**: Shows that a license is not yet finalized but has passed the voting threshold so it is visible on the user interface. Admins have the option to personally approve or reject these licenses in response to community feedback.
- **Disapproved flag**: Shows that a license has been reported as needing to be removed and shouldn't be seen by users any more.
- **Final Conclusion flag**: Raised when a license is concluded that is, once it has received final approval.

The accuracy and applicability of the discovered and concluded licenses in the system depend on these flags. To make sure the system runs properly, administrators manually manage these flags in the database.

6.6 Error handling

In order for the system to properly handle unforeseen problems like invalid inputs, server errors, and database failures; error handling is an essential component. This is how the system's various tiers implement error management.

Frontend

Errors resulting from network problems and user input are handled by the frontend. For instance:

- The frontend verifies the input and shows the relevant error messages if a user tries to submit an invalid license in terms of empty license information.
- The frontend detects the error and notifies the user that the operation could not be completed if the backend returns an error (such a 500 Internal Server Error).

This guarantees that consumers may adjust their behavior and are informed of any problems in real time.

Controller layer

Error handling is done at the controller layer to handle exceptions and return the proper HTTP status codes. Typical error types addressed are as follows:

- **Invalid inputs**: The controller returns a 400 Bad Request status and an error message detailing the reason for the failure if a user enters missing or erroneous data.
- **Resource not found**: The controller provides a 404 Not Found response in the event that a requested package or license cannot be located.

By doing this, a better user experience is made possible by ensuring that the frontend receives structured and relevant error notifications.

Service layer

Error management at the service layer is concentrated on business logic exceptions, like:

- **Validation errors**: The service throws a validation exception, which the controller detects and handles correctly, if a license does not match specific requirements (such as a empty license information).
- **Database errors**: The service detects the error and sends the relevant message to the controller in the event that a database operation fails (for example, because of a constraint violation).

This layer makes sure that improper actions are detected early and corrected before they affect the functionality or performance of the system.

Repository layer

To manage SQL exceptions, database activities in the repository layer are encapsulated in try-catch blocks. Repository errors propagate back through the service and controller layers when a query fails, for example, due to a missing record or constraint violation. For the purposes of auditing and debugging, the repository also logs errors.

6.7 Performance optimizations

The crowdsourcing license correction system is designed to handle and manage huge datasets efficiently, and numerous optimizations are implemented to guarantee seamless operation, even when a substantial number of users participate and assess licenses.

Database query optimization

The repository layer uses optimized queries to retrieve data efficiently:

- **Indexing**: To speed up lookups and minimize query execution time, important database fields including package_purl, discovered_license, and vote_type, are indexed.
- **Aggregation queries**: To reduce the amount of database transactions, optimal aggregation queries are used to retrieve vote counts and other summary statistics.

These optimizations make sure the system stays responsive and effective even with an increase in the number of packages and licenses.

Pagination and lazy loading

To prevent the frontend from becoming overloaded with big datasets, the system provides paging and slow loading:

- **Pagination**: The system reduces the amount of data exchanged between the server and frontend by returning only a fraction of the results when obtaining components or license data, based on the current page number and limit.
- **Lazy loading**: This technique speeds up the initial load time and cuts down on pointless network traffic by only loading extra data (such license details) when needed.

In summary, the architecture of the crowdsourcing license findings was implemented using a layered architecture with a React and TypeScript frontend, a Spring Boot and Java backend, and a PostgreSQL database. The system allows users to add new licenses, vote on existing ones, and view license information, all while ensuring data validation and efficient communication between layers. Admin functionalities were implemented for user management and license flagging directly through the database. Throughout the implementation, close attention was given to error handling and performance optimizations like database indexing and pagination to make sure the system is robust and responsive.

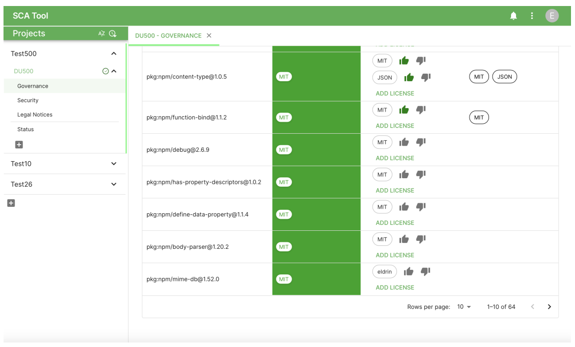
7 Evaluation

In this chapter, the system is put to the test by evaluating how well it meets both the functional and non-functional requirements described earlier. All the important features like adding new licenses, the voting mechanism, and the process of concluding licenses from both individual and community perspectives to see if they function as intended. Administrative capabilities were also tested such as license flagging by admins to ensure they effectively help maintain the system's integrity. On the non-functional side, aspects like performance, security, and scalability to determine if the system can handle real-world demands were tested. Finally, some of the limitations were also outlined, offering insights into areas where the system could be improved.

7.1 Functional requirements evaluation

The functional evaluation was conducted based on the core features of the system. Each key functionality was tested to ensure it works as intended and meets user expectations.

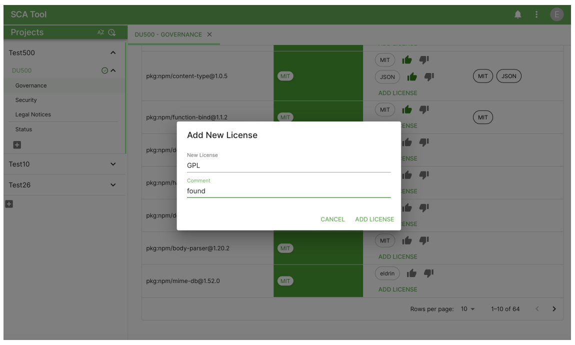
License contribution



pkg:npm/content-type@1.0.5	MIT	MIT, JSON, ADD LICENSE	MIT, JSON
pkg:npm/function-bind@1.1.2	MIT	MIT, ADD LICENSE	MIT
pkg:npm/debug@2.6.9	MIT	MIT, ADD LICENSE	
pkg:npm/has-property-descriptors@1.0.2	MIT	MIT, ADD LICENSE	
pkg:npm/define-data-property@1.1.4	MIT	MIT, ADD LICENSE	
pkg:npm/body-parser@1.20.2	MIT	MIT, ADD LICENSE	
pkg:npm/mime-db@1.52.0	MIT	eldrin, ADD LICENSE	

Rows per page: 10 1–10 of 64

Figure 4: Navigating the user interface



Add New License

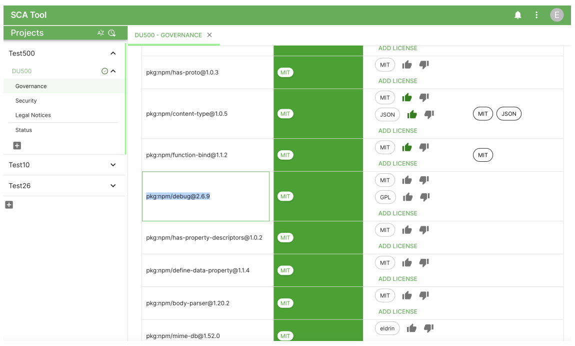
New License: GPL

Comment: found

pkg:npm/content-type@1.0.5	MIT	MIT, JSON, ADD LICENSE	MIT, JSON
pkg:npm/function-bind@1.1.2	MIT	MIT	MIT
pkg:npm/body-parser@1.20.2	MIT	MIT, ADD LICENSE	
pkg:npm/mime-db@1.52.0	MIT	eldrin, ADD LICENSE	

Rows per page: 10 1–10 of 64

Figure 5: Adding a license "GPL" along with comments

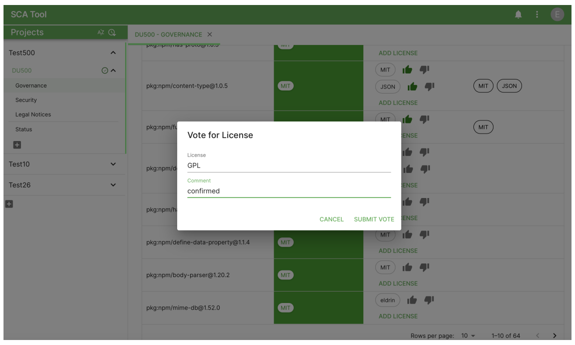


pkg:npm/has-proto@1.0.3	MIT	MIT, ADD LICENSE	
pkg:npm/content-type@1.0.5	MIT	MIT, JSON, ADD LICENSE	MIT, JSON
pkg:npm/function-bind@1.1.2	MIT	MIT, ADD LICENSE	MIT
pkg:npm/debug@2.6.9	MIT	MIT, GPL, ADD LICENSE	
pkg:npm/has-property-descriptors@1.0.2	MIT	MIT, ADD LICENSE	
pkg:npm/define-data-property@1.1.4	MIT	MIT, ADD LICENSE	
pkg:npm/body-parser@1.20.2	MIT	MIT, ADD LICENSE	
pkg:npm/mime-db@1.52.0	MIT	eldrin	

Figure 6: Successfully added GPL license

Voting mechanism

Users were able to upvote or downvote licenses and provide comments for their decisions.



Vote for License

License: GPL

Comment: confirmed

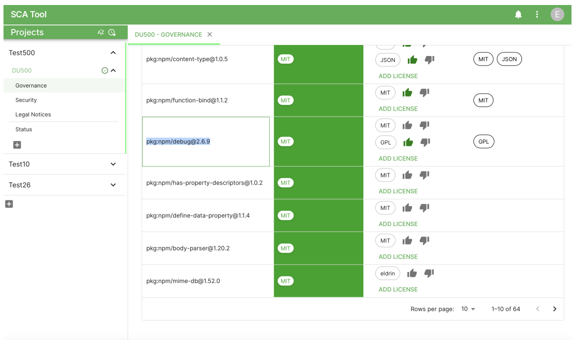
pkg:npm/content-type@1.0.5

pkg:npm/define-data-property@1.1.4

pkg:npm/body-parser@1.20.2

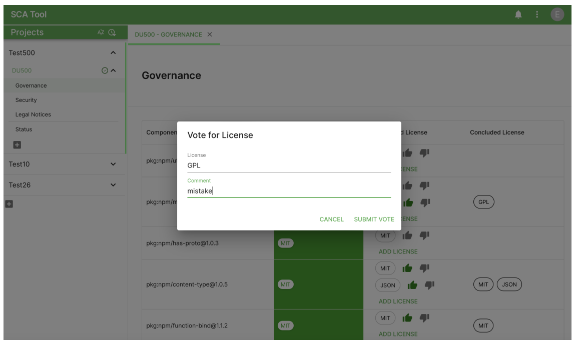
pkg:npm/mime-db@1.52.0

Figure 7: Approving/Upvoting a discovered license



Component		Licenses	Concluded License
pkg:npm/content-type@1.0.5	MIT	JSON	MIT, JSON
pkg:npm/function-bind@1.1.2	MIT	MIT	MIT
pkg:npm/debug@2.6.9	MIT	MIT, GPL	GPL
pkg:npm/has-property-descriptors@1.0.2	MIT	MIT	
pkg:npm/define-data-property@1.1.4	MIT	MIT	
pkg:npm/body-parser@1.20.2	MIT	MIT	
pkg:npm/mime-db@1.52.0	MIT	eldrin	

Figure 8: Approval/Upvoted successfully



Vote for License

License: GPL

Comment: mistake

pkg:npm/has-proto@1.0.3

pkg:npm/content-type@1.0.5

pkg:npm/function-bind@1.1.2

Figure 9: Disapproving/Downvoting discovered license

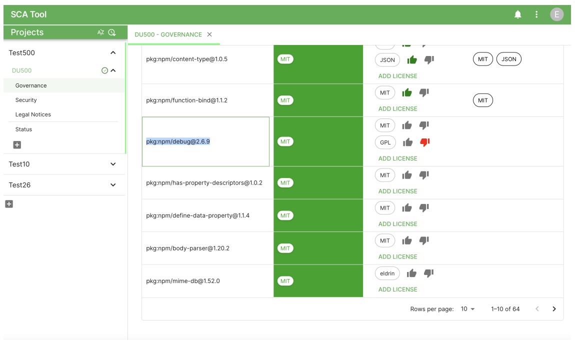


Figure 10: Disapproval/Downvote successful

Concluded license: Same user

User was successfully suggested a concluded license if the same user has added the concluded license for the same package but for a different distribution unit.

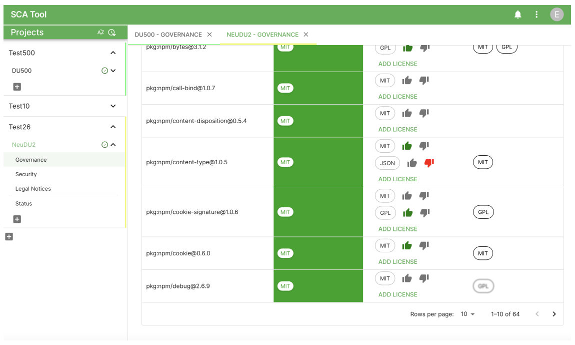
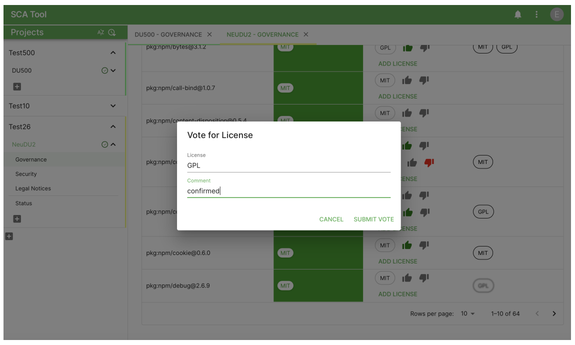


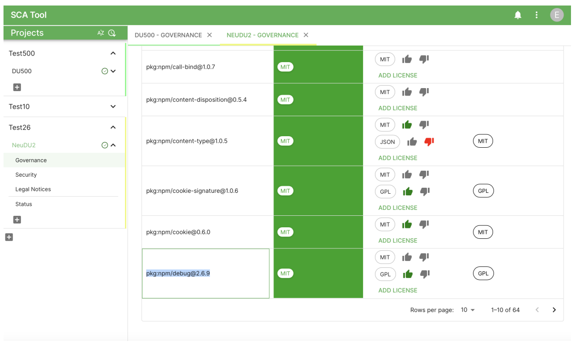
Figure 11: Same user(eldrin), same package (pkg:npm/debug@2.6.9), different distribution unit getting suggested a license since he concluded a license in the other distribution unit


Vote for License

License
GPL

Comment
confirmed

Figure 12: User marking/voting GPL license as concluded in this distribution unit as well

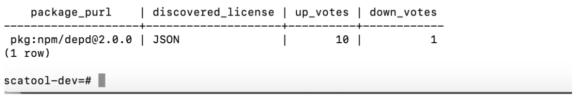


pkg:npm/call-bind@1.0.7 MIT
pkg:npm/content-disposition@0.5.4 MIT
pkg:npm/content-type@1.0.5 MIT
pkg:npm/cookie-signature@1.0.6 MIT
pkg:npm/cookie@0.6.0 MIT
pkg:npm/debug@2.6.9 MIT

Figure 13: Task successful

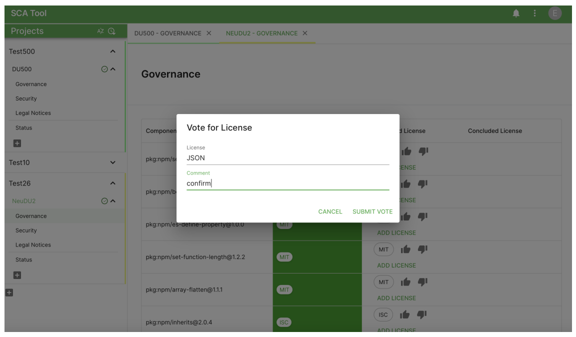
Concluded license: Community

Another user was successfully suggested a concluded license if the net difference reached the threshold of 10 upvotes across multiple users in the system.



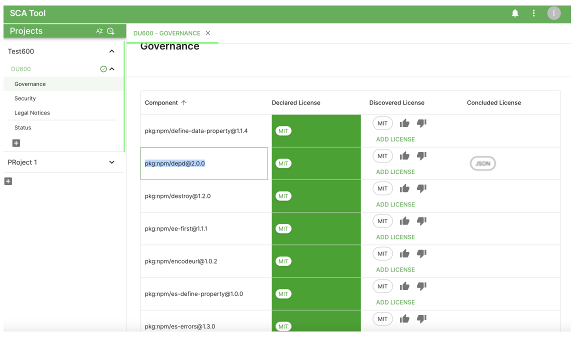
```
    package_purl     | discovered_license | up_votes | down_votes
---------------------+--------------------+----------+------------
 pkg:npm/depd@2.0.0  | JSON               |       10 |          1
(1 row)

scatool-dev=#
```

Figure 14: JSON License has a net difference of 9 upvotes which is 1 vote less than the threshold in package "pkg:npm/depd@2.0.0".



pkg:npm/es-define-property@1.0.0
pkg:npm/set-function-length@1.2.2
pkg:npm/array-flatten@1.1.1
pkg:npm/inherits@2.0.4

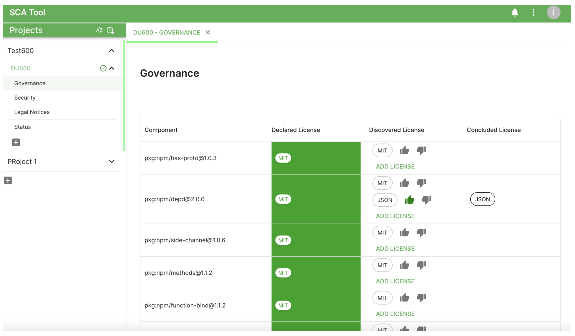
Figure 15: User "Eldrin" upvoting the JSON license making the net difference to 10 upvotes.



Governance

Component	Declared License	Discovered License	Concluded License
pkg:npm/define-data-property@1.1.4	MIT	MIT ADD LICENSE	
pkg:npm/depd@2.0.0	MIT	MIT ADD LICENSE	JSON
pkg:npm/destroy@1.2.0	MIT	MIT ADD LICENSE	
pkg:npm/ee-first@1.1.1	MIT	MIT ADD LICENSE	
pkg:npm/encodeurl@1.0.2	MIT	MIT ADD LICENSE	
pkg:npm/es-define-property@1.0.0	MIT	MIT ADD LICENSE	
pkg:npm/es-errors@1.3.0	MIT	MIT	

Figure 16: Another user "Issac" is being suggested the JSON license as it has reached the threshold.



Governance

Component	Declared License	Discovered License	Concluded License
pkg:npm/has-proto@1.0.3	MIT	MIT ADD LICENSE	
pkg:npm/depd@2.0.0	MIT	MIT JSON ADD LICENSE	JSON
pkg:npm/side-channel@1.0.6	MIT	MIT ADD LICENSE	
pkg:npm/methods@1.1.2	MIT	MIT ADD LICENSE	
pkg:npm/function-bind@1.1.2	MIT	MIT ADD LICENSE	

Figure 17: User Issac has now added the JSON license.

License flagging by admins

The system includes the ability for admins to manually flag or disapprove licenses in the database. Admins were able to:

- Mark licenses as disapproved by setting a flag in the database.
- Flag users as trolls if they consistently contributed malicious or incorrect licenses, preventing those users contribution to be counted among the community.

7.2 Non-functional requirements evaluation

In addition to functional testing, the system was evaluated against several non-functional criteria, including performance, security, scalability, and usability.

- **Performance**

  The performance of the system was measured based on response times and system throughput. Key results include:

  - **Response times**: The response time of the system remains unchanged after crowdsourcing functionality was introduced indicating that it does not affect the performance.
  - **Concurrency**: The system was tested with multiple concurrent users contributing and voting on licenses. It scaled effectively, with no significant degradation in performance.

  Caching mechanisms implemented in the repository layer, such as vote counts and component data, contributed to the system's performance, reducing the number of database queries and improving overall speed.

- **Security**

  - **User management**: Admins could successfully flag users and prevent them from contributing further licenses to the community, ensuring that malicious users could not disrupt the community.

- **Scalability**

  An increasing number of users and licenses were intented to be handled by the system as it scaled. Key results from scalability testing include:

  - **Database Efficiency**: The repository layer's queries were optimized to handle large datasets efficiently. Indexing on key columns like package_purl and discovered_license helped improve query performance.
  - **Pagination**: The system's use of pagination in component loading helped maintain performance, even when handling large numbers of packages.

  The system's architecture is well-positioned to scale as the number of packages, licenses, and users grows.

7.3 Limitations

Despite its strengths, the system has several limitations:

- **Admin Overhead**: The system relies on admins manually approving or disapproving licenses, which can become time-consuming if the number of submissions grows. Automating some aspects of admin tasks could improve scalability.
- **Dependence on Community Participation**: The system's effectiveness depends on active community participation. If the user base is small or inactive, the system may struggle to identify and conclude licenses efficiently.
- **Potential for Malicious Contributions**: Although mechanisms are in place to flag and block malicious users, there is still a risk of spam or incorrect contributions overwhelming the system, especially in the absence of a dedicated admin interface.

8 Conclusion

8.1 Conclusion

In this thesis, a crowdsourcing-based licensing correction system that is coupled with **Software Composition Analysis (SCA)** tool was designed and developed. The objective was to increase the accuracy of open-source software (OSS) license findings because this is a crucial job that ensures compliance with legal and regulatory requirements. Because of the continually changing open-source ecosystem, traditional license scanners such as ScanCode and FOSSology, while good at recognizing established licenses, frequently have trouble with new or updated licenses. The purpose of this thesis was to close this gap by adding human judgment to the license correction process.

The system allows users to contribute discovered licenses, vote on the accuracy of those licenses, and provide comments explaining their decisions. When enough upvotes are cast for a license, it is designated as "concluded license," awaiting administrative approval. Admins can also regulate and report malicious users and improper votes using the tools built into the system. The overall accuracy of license identification is improved by combining administrative control with a community-driven approach.

The system's assessment showed that both functional and non-functional requirements were satisfactorily addressed. The crowdsourcing method made it easier to identify licenses that automated programs had overlooked, resulting in a more accurate and trustworthy license database. Strong user interactions and efficient admin controls to regulate user behavior characterized the voting system. Additionally, the system managed security issues including some malicious input and user authentication, and it operated well under pressure.

The significance of crowdsourcing in enhancing the identification of open-source licenses was also emphasized in the thesis. The system remains up to date with new and evolving licenses, something that automated systems by themselves are unable to handle, thanks to community engagement. Even if the system has shown to be successful, there is still room for development, especially when it comes to scaling and simplifying administrative duties.

8.2 Future work

Even though the system's current implementation serves its intended function, there are a number of ways to improve it going forward, including scalability, usability, and overall effectiveness.

Scaling the crowdsourcing system

The system will have to develop to accommodate more license contributions, votes, and user interactions as the open-source ecosystem expands and new OSS components appear.

Dedicated admin interface

Currently, admin tasks are carried out directly in the database. These tasks include controlling harmful contributions, flagging users, and reviewing and approving licenses. Although practical, this manual method could become ineffective as the number of users increases.

Incorporating machine learning and large language models (LLMs)

The current system allows users to contribute, vote, and comment on licenses only through human input. Large Language Models (LLMs) and machine learning may be added to the system in the future to improve license detection accuracy and expedite user engagement.

Bibliography

Black-Duck-Team (2023). *Black Duck Software Composition Analysis (SCA) tool | Black Duck*. Available at `https://www.blackduck.com/software-composition-analysis-tools/black-duck-sca.html`.

Domar Bolmstam, S., S. Hanifi, K. Skolan, F. Elektroteknik, and O. Datavetenskap (2020). *Security Guidelines for the Usage of Open Source Software*. Avaialable at `https://www.diva-portal.org/smash/get/diva2:1463731/FULLTEXT01.pdf`.

Duan, R., A. Bijlani, M. Xu, T. Kim, and W. Lee (2017). "Identifying Open-Source License Violation and 1-day Security Risk at Large Scale". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. Avaialable at `https://dl.acm.org/doi/pdf/10.1145/3133956.3134048`.

Fendt, O. and M. C. Jaeger (2019). "Open Source for Open Source License Compliance". In: *Open Source Systems*. Ed. by F. Bordeleau, A. Sillitti, P. Meirelles, and V. Lenarduzzi. Cham: Springer International Publishing, pp. 133–138.

Fontana, R., B. Kuhn, E. Moglen, M. Norwood, D. Ravicher, K. Sandler, J. Vasile, and A. Williamson (2008). *A Legal Issues Primer for Open Source and Free Software Projects*. Avaialable at `https://softwarefreedom.org/resources/2008/foss-primer.pdf`.

FOSSology-FAQ (2024). *FOSSology FAQ*. Avaialable at `https://wiki.fossology.org/faq`.

FOSSology-Setup (2024). *FOSSology Setup*. Available at `https://github.com/fossology/fossology/wiki/Install-from-Source`.

FOSSology-Wiki (2024). *FOSSology Documentation*. Avaialable at `https://github.com/fossology/fossology/wiki`.

Harutyunyan, N., A. Bauer, and D. Riehle (2019). "Industry requirements for FLOSS governance tools to facilitate the use of open source software in commercial products". In: *Journal of Systems and Software* 158. Avaialable at `https://osr.cs.fau.de/wp-content/uploads/2019/08/jss-2019-harutyunya-bauer-riehle.pdf`, p. 110390.

HU, D., L. ZHAO, and J. CHENG (2024). "Reputation management in an open source developer social network: An empirical study on determinants of positive evaluations". In: *Decision support systems* 53. Avaialable at `http://pascal-francis.inist.fr/vibad/index.php?action=getRecordDetail&idt=26074049`, pp. 526–533.

Kemppainen, P. (2023). *Managing 3rd Party Software Components with Software Bill of Materials*. Tech. rep. Available at `https://trepo.tuni.fi/bitstream/handle/10024/148790/KemppainenPaavo.pdf?sequence=2`. Tampere University.

Li, L., W. Chou, W. Zhou, and M. Luo (2016). "Design Patterns and Extensibility of REST API for Networking Applications". In: *IEEE Transactions on Network and Service Management* 13.1. Available at `https://ieeexplore.ieee.org/document/7378522`, pp. 154–167.

Morris, P. and M. Martin (2004). *Open Source Software Licenses: Perspectives of the End User and the Software Developer*. Avaialable at `https://www.mmmlaw.com/files/documents/publications/article_238.pdf`.

Noda, N. and T. Kishi (2001). "Design pattern concerns for software evolution". In: Available at `https://dl.acm.org/doi/pdf/10.1145/602461.602498`, pp. 158–161.

O'Reilly-Team (2024). *Software Architecture in Practice, 4th Edition [Book]*. Available at `https://www.oreilly.com/library/view/software-architecture-in/9780136885979/`. www.oreilly.com.

Pantaleev, A. and A. Rountev (2007). "Identifying Data Transfer Objects in EJB Applications". In: *Fifth International Workshop on Dynamic Analysis (WODA '07)*, pp. 5–5.

Phipps, S. and S. Zacchiroli (2020). “Continuous Open Source License Compliance”. In: *arXiv preprint arXiv:2011.08489*. Available at `https://arxiv.org/pdf/2011.08489`.

Rana, M. E. and E. Khonica (2021). “Impact of Design Principles and Patterns on Software Flexibility: An Experimental Evaluation Using Flexible Point (FXP)”. In: *Journal of Computer Science*.

Al-Samman, A. (2020). “Modeling FLOSS Dependencies in Products”. In: *Open Source Software*. Available at `https://oss.cs.fau.de/wp-content/uploads/2020/10/al-samman_2020.pdf`.

Samman, A., A. Bauer, and D. Riehle (2024). *MODELING FLOSS DEPENDENCIES IN PRODUCTS*. Available at `https://oss.cs.fau.de/wp-content/uploads/2020/10/al-samman_2020.pdf`.

ScanCode-Wiki (2024). *ScanCode Toolkit Documentation*. Avaialable at `https://scancode-toolkit.readthedocs.io`.

Simons, A., K. Hung, and M. Snoeck (1999). “Using Design Patterns to Reveal the Competence of Object-Oriented Methods in System-Level Design”. In: *Computer Systems: Science Engineering - CSSE* 14.

SPDX-Concluded-License-Info (2023). *SPDX Specification v2.3 - Concluded License Field*. Accessed: 2024-10-06.

SPDX-Document-Info (2022). *Clause 6: Document Creation Information - specification v2.3.0*. Available at `https://spdx.github.io/spdx-spec/v2.3/document-creation-information`. Github.io.

SPDX-License-List-Info (2022). *Annex A: SPDX License List - specification v2.3.0*. Available at `https://spdx.github.io/spdx-spec/v2.3/SPDX-license-list`. Github.io.

Wagner, M. (2023). “JavaScript User Interface License Compliance Best Practices”. In: *Open Source Software*. Available at `https://oss.cs.fau.de/wp-content/uploads/2023/06/Wagner_2023.pdf`.

Wolter, T., A. Barcomb, D. Riehle, and N. Harutyunyan (2022). “Open Source License Inconsistencies on GitHub”. In: *ACM Transactions on Software Engineering* *and Methodology*. Avaialable at `https://dirkriehle.com/wp-content/uploads/2022/10/License_Inconsistencies_on_Github.pdf`.

Yang, H. Y., E. Tempero, and H. Melton (2008). "An Empirical Study into Use of Dependency Injection in Java". In: *19th Australian Conference on Software Engineering (aswec 2008)*. Available at `https://dl.acm.org/doi/pdf/10.1145/602461.602498`, pp. 239–247.

Declaration of authorship

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

Erlangen, 18.10.2024

Eldrin Sanctis