LicenseLynx: Towards mapping licenses to canonical identifiers

MASTER THESIS

Leo Reinmann

Submitted on 1 April 2025

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

Supervisor:
Prof. Dr. Dirk Riehle, M.B.A.
Thomas Jensen, Dipl.-Inf., Siemens AG

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.

Erlangen, 1 April 2025

License

Erlangen, 1 April 2025

Abstract

In today's software development landscape, managing licenses across various projects is a complex task, especially with recent cybersecurity regulations necessitating accurate Software Bill of Materials (SBOM). This research addresses the problem of mapping arbitrary license strings to canonical identifiers, a crucial step for automated license compliance. Previous solutions, such as FOSSology and LDBcollector, have made strides in license identification but lack integration and ease of use in code. This thesis introduces LicenseLynx, an open source tool designed to fill this gap by providing deterministic mappings of license strings to canonical identifiers. Utilizing data from System Package Data Exchange (SPDX), ScanCode LicenseDB, and Open Source Initiative (OSI), the tool aims to enhance accuracy and reduce errors in automated license compliance processes. The methodology involves data collection, validation, and the development of programming libraries and a web API for license mapping. Key findings include the successful integration of major data sources, resulting in over 8500 mappings for 2300 licenses. The implications of this thesis are significant for improving license compliance and cybersecurity in software development, offering a robust solution for accurate license identification.

Contents

List of Figures

Listings

Acronyms

CRA Cyber Resilience Act

NIST The National Institute of Standards and Technology

CSF Cybersecurity Framework

SBOM Software Bill of Materials

IoT Internet of Things

CCPA California Consumer Privacy Act

SemVer Semantic Versioning

CalVer Calendar Versioning

SPDX System Package Data Exchange

CLI Command Line Interface

OSI Open Source Initiative

OSD Open Source Definition

JSON JavaScript Object Notation

MVP minimum viable product

API Application Programming Interface

1 Introduction

In today's software development landscape, managing licenses across various projects and even within a single project can be a daunting task. The recent developments in laws regarding cybersecurity and the need for security vulnerability monitoring have resulted in the necessity for accurate SBOM, including correct license information.

1.1 Fundamentals

Software clearing can be done in multiple ways. Figure 1.1 illustrates the automated license compliance process. To clear software and eventually deploy it to the customer, the Standard BOM plays a crucial role. The Standard BOM is a subset of the CycloneDX standard with the Siemens property taxonomy (Jensen and Greinacher, 2024). When code is pushed to the remote repository, a pipeline job is triggered. Appropriate dependency scanners go through the repository and create the Standard BOM file. The Standard BOM file is then used for further processing, such as license validation or generating a *README.OSS* file. The processing step enriches the Standard BOM before it is finally used for further use cases like Security Vulnerability Monitoring or License Compliance.

In the ideal case, the license in the Standard BOM is always an SPDX identifier. But in reality, the scanners can only find as much as how well-maintained the dependencies are. The licensing in dependencies is out of the control of the software developer and is not always found automatically by the scanners, or some license string is entered. The problem here is that for further processing, the tools need to understand the license string to find more details about it. This gap is important to fill with a solution for higher quality and less error-prone automated software clearing.

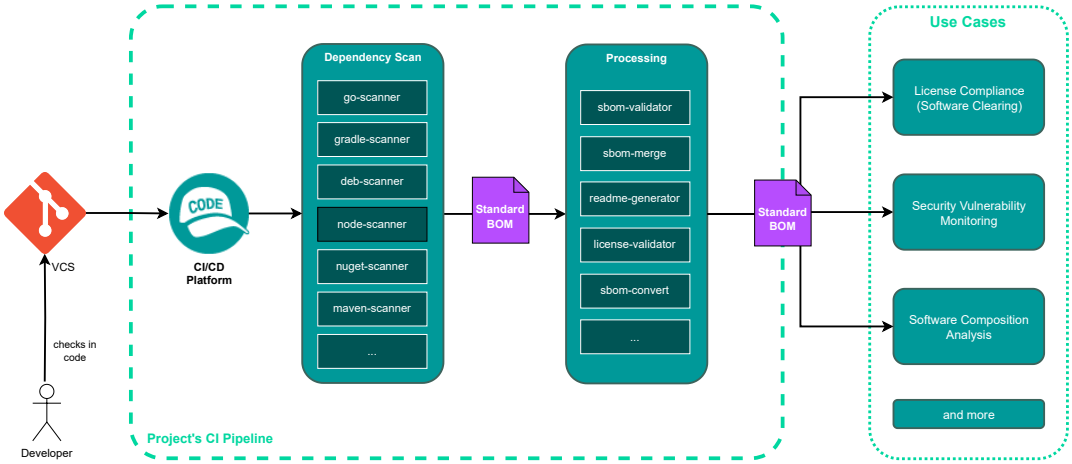


Figure 1.1: Activity diagram for automated software clearing

1.2 Motivation

To fill the gap, we need a tool that can take any license string and try to map it deterministically to a canonical identifier. We are not the first to attempt this, as we will discuss later in Chapter 2, but we need a solution that can be used in code with other tools. It should also be straightforward to contribute to this project with more mappings, so that the tool becomes more accurate over time.

One can use the tool in the scanners and in the processing tools within the pipeline. To make it more publicly accessible and visible, this tool should be an open-source project.

1.3 Objective

We aim to provide a deterministic solution to map licenses to canonical identifiers, which we call *LicenseLynx*. First, we analyze the existing solutions and what we can improve, as well as discover potential data sources. The project should have data engineering scripts to automatically collect data from data sources, update the data, and validate the collected data. Then we want to provide an Application Programming Interface (API) to simply map any license string and get its canonical identifier. The different components should be managed in a monorepo. The goal is for the project to be open-source and well-documented so that contributions are more likely to be made in the long run.

1.4 Outline

First, in Chapter 2, we review licensing, existing solutions, and the data sources we want to use. Then, in Chapter 3, we clarify our functional and non-functional requirements for the project. In Chapter 4, we describe the components of the project and their functionality. Next, in Chapter 5, we explain how we designed and implemented each component of our project. After that, in Chapter 6, we evaluate our project, gather statistics for our data, assess if we meet our requirements, and outline the found limitations. Lastly, in Chapter 7, we summarize our work and provide an outlook for future work.

2 Literature Review

In this chapter, we compare and explain the need for correct and accurate licensing by examining the cybersecurity regulations of the EU and the USA. Then, we look at existing solutions for identifying licenses in software, with a particular focus on the FOSSology project. Finally, we review different data sources, explaining their processes for adding new licenses to their lists and how these lists are structured.

2.1 Cybersecurity Regulations: A Comparison of EU and US Approaches to Licensing

We review the regulations regarding licensing and SBOM. First, we analyze the EU law and then the US law and compare both approaches to each other.

2.1.1 EU Law: The Cyber Resilience Act and Its Impact on Licensing

The Cyber Resilience Act (CRA), adopted by the European Commission in October 2024, is a comprehensive regulatory framework designed to improve the cybersecurity of digital products and software across the European Union (Official Journal of the European Union, 2024). The Act addresses the increasing prevalence of cyber threats by requiring manufacturers, importers, and distributors to ensure that their products meet stringent cybersecurity requirements throughout their lifecycle.

This legislation applies to a broad range of products with digital components, excluding only specific categories such as open-source software that is not commercialized (European Commission, 2024). By enforcing these requirements, the CRA aims to foster trust and resilience within the digital ecosystem while harmonizing cybersecurity standards across member

states.

A key element of the CRA's focus on cybersecurity is ensuring transparency in software development, including proper licensing and adherence to security obligations. Licensing defines the legal and operational parameters for software distribution, updates, and maintenance, making it fundamental for achieving compliance with CRA objectives.

Accurate identification and mapping of software licenses are essential for maintaining legal compliance, securing supply chains, and adhering to cybersecurity standards. Ambiguities or inconsistencies in license identification can hinder efforts to meet these requirements, particularly when clarity is critical for understanding software provenance and obligations.

Despite its importance, license identification often faces challenges due to ambiguities or the use of multiple naming conventions for the same license. These inconsistencies can complicate efforts to align with compliance frameworks such as the CRA, where understanding the obligations and provenance of software components is crucial. Resolving these ambiguities is essential to ensure transparency and meet regulatory goals.

The Cyber Resilience Act is not an isolated initiative but part of a broader global push toward enhanced software resilience and compliance. Similar efforts, such as the US Executive Order 14028 on software security, also underscore the importance of transparency and licensing clarity in mitigating cybersecurity risks. These parallel initiatives highlight the universal relevance of improving licensing practices as a foundation for building trust and accountability in the digital ecosystem.

2.1.2 US Law: Fragmented Approaches to Cybersecurity and Licensing

In contrast to the European Union's comprehensive CRA, the United States adopts a decentralized and sector-specific approach to cybersecurity regulation (Fahey, 2024). While there is no direct equivalent to the CRA at the federal level, several laws, frameworks, and executive orders focus on improving software security, supply chain resilience, and transparency. These initiatives reflect an increasing emphasis on mitigating cybersecurity risks across both public and private sectors.

The National Institute of Standards and Technology (NIST) developed the Cybersecurity Framework (CSF) as a voluntary set of standards, guidelines, and best practices for managing cybersecurity risks (National Institute of

Standards and Technology, 2022). It is widely adopted by organizations in the U.S. to improve their cybersecurity posture and ensure the security of digital products.

Issued in May 2021, Executive Order 14028 emphasizes software supply chain security and mandates federal agencies to adopt rigorous cybersecurity measures (The United States Government, 2021). Key provisions include:

- Requiring software developers to provide transparency into their software supply chains through the use of SBOMs.
- Establishing minimum security standards for software purchased by federal agencies.
- Encouraging security assessments and vulnerability disclosure mechanisms.

The Internet of Things (IoT) Cybersecurity Improvement Act targets the security of IoT devices used by federal agencies (House - Oversight and Reform; Science, Space, and Technology, 2020). It mandates that devices meet minimum cybersecurity requirements, focusing on secure development, identity management, and vulnerability management.

States like California have enacted laws that promote cybersecurity and software licensing transparency (State of California Department of Justice, 2024). The California Consumer Privacy Act (CCPA) and the California IoT Security Law are examples of how state-level regulations contribute to cybersecurity resilience by emphasizing secure and transparent practices in software and hardware development.

U.S. cybersecurity initiatives increasingly recognize the importance of transparency and compliance in software development. Licensing clarity, including the accurate mapping of licenses to canonical names, is essential for adhering to these frameworks. For example: The requirement for SBOMs under Executive Order 14028 demands clear and consistent identification of software components and their associated licenses. State-level and federal laws highlight the need for software users and developers to assess compliance risks tied to ambiguous or improperly documented licenses.

While the U.S. regulatory landscape is fragmented compared to the EU's CRA, its focus on software supply chain security and transparency underscores the global importance of addressing licensing challenges in the digital ecosystem.

2.2 Existing Solutions for License Identification

We look in the following into existing solutions regarding license identification. We go over the FOSSology Project and analyzing their tools to identify licenses. Then, we have a look at LDBcollector and review their solution. Lastly, we briefly explain tl;drLegal.

2.2.1 The FOSSology Project

The FOSSology project is an open-source license compliance tool that provides solutions for license management, code plagiarism, and vulnerability tracking (Gobeille, 2008, Cornec, 2012). It offers a web user interface or Command Line Interface (CLI) where users can upload projects to perform metadata detection, such as license detection.

The architecture of FOSSology consists of a software repository, database, and a cluster of agents (Gobeille, 2008). Each agent executes specific tasks. Their dashboard in the web browser shows each component and its metadata.

FOSSology uses two agents called Monk and Nomos to scan for licenses in source files (AG, 2016). The difference between both tools is flexibility and precision. Monk is more precise than Nomos but lacks flexibility because it only works on license texts by using full-text matching. Nomos, however, uses keywords and regular expressions to identify licenses, which can lead to false positives.

Nomos

Nomos is one of the original scanners developed by HP (Stewart, 2015). It uses regular expressions and heuristics to detect licenses based on signature definitions stored in the file *STRINGS.in* (Wang et al., 2018). This file contains entries that specify the regular expressions for each recognized license. To recognize additional licenses, it is possible to extend this file by using the following template in Listing 2.1.

```
1 %ENTRY% _TITLE_XYZ-LICENSE1
2  %KEY%      "licen[cs]"
3  %STR%      "XYZ-licen[cs]e (v|version ) 1\.?0"
```

Listing 2.1: Template to add new signatures to STRING.in

Then another file needs to be modified, the *parse.c* file. This file interprets the heuristics of the stored license signatures in *STRINGS.in*. The user has to add an else-if block to that file as shown in Listing 2.2.

```
1 else if (INFILE(_TITLE_XYZ-LICENSE1)) {
2     INTERESTING("XYZ_v1.0");
3 }
```

Listing 2.2: Template to add new signatures to parse.c

Monk

Monk was developed by Siemens and TNGtech and added to FOSSology in 2014 (Weber et al., 2018). In contrast to Nomos, it scans through the full license text and returns a similarity score based on the Jaccard index. Monk compares the entire content of a file against known license texts stored in the *license_ref table*, which is a database with all the metadata of licenses stored (Jaeger and Wang, 2015).

Monk scans through the whole license text and compares each word against the database, thus white space is ignored. However, it is case-sensitive, and commas, punctuation, and quotes are not discarded. The following examples show these rules in detail:

```
"license_name" != license_name
Apache License, version 2.0 != Apache License version 2.0
Case-sensitive != case-SENSITIVE
```

There are two different results Monk can return. The full match result is a one hundred percent match between the database and the scanned file. Monk points to the position where these complete matches were found with the format `matched: startPosition + length`, with the unit of measure being 1-byte characters.

If an exact match is not found, Monk produces a detailed diff output, highlighting differences between the detected license text and the reference, aiding users in identifying modifications or anomalies.
The format is `diff: {diff_1, ..., diff_N}` where each `diff` has three values.
The first value describes the range of which part in the input text is affected, similar to the format of the full match result, e.g., `t[0+1000]`, where the unit is again 1-byte characters.
The second value shows if the text in the range is equal to the reference with the character `M0`, if the text is in the reference but not in the input

file with the character `M-`, and if the text is added to the input file but not existing in the reference with the character `M+`.
The last value shows the position in the reference text, e.g., `s[30+1000]`. White space is counted to calculate the position in both the input text and the reference but, as mentioned before, not in the comparison itself.

The workflow of Monk is that it connects to the database, reads all reference texts from the *license_ref* table, and tokenizes them. Then the input file is scanned, and the content is also tokenized. After that, the match finding begins, where full matches are searched first. Every time a difference between the reference text and input text is found, Monk tries to search for the next full match. There are two parameters that control when the process needs to change to the next reference text. The first is *MAX_ALLOWED_DIFF_LENGTH*, which determines when the current reference is too different from the input text.
The second one is *MIN_TRAILING_MATCHES*, which determines how many consecutive equal tokens are needed to decide that the difference is finished and the matching can continue.
Then each diff match is ranked by the Jaccard Index with the following formula:

$$\frac{\#\{\text{matched}\}}{\#\{\text{added}\} + \#\{\text{removed}\} + \#\{\text{matched}\}}$$

The parameter *MIN_ALLOWED_RANK* automatically removes matches whose rank is not high enough.

The last step is match filtering, where the matches are divided into ordered groups such that the first element in each group holds all other matches within the same group. For each group, if the match with the highest rank corresponds to the first element, only the first match is retained. If the highest-ranked match is not the first element, the first match is discarded, and the remaining matches in the group are subjected to a recursive analysis.

2.2.2 LDBcollector

The LDBcollector collects metadata across many sources and merges them into a normalized format (Huber, 2024). It is possible to run the application with Docker, but unfortunately, the documentation for this tool is sparse and it requires some permissions to retrieve the Docker image.

However, what it lacks in documentation, it exceeds in data. Many data sources from various organizations are copied into the repository, which

has a size of 1.6 Gigabytes according to GitHub (GitHub, 2025). The reason may also be that they are currently working on version two but still offer a stable version one in a separate branch on GitHub.

Figure 2.1 shows all data sources used by LDBcollector (Huber, 2024). After the data collection, the data undergoes a parsing and normalization process and is then provided in different file formats. In version one, the different licenses were grouped into various file formats, e.g., JavaScript Object Notation (JSON) or Markdown. Version two groups the licenses into OSI and SPDX folders and does not group the licenses into different file formats.

The amount of data is unmatched by any other project we found, but without thorough documentation, it is difficult to use this tool. What we want to improve is to provide, along with the data, programming libraries and a programming language-agnostic API.

LDBcollector could be a good data source for more data in future work. Version one provides a CSV called *aliases.csv*, which contains over four thousand mappings and also provides an index file where 449 licenses are listed (Huber, 2021, Huber, 2022). Unfortunately, version two does not provide these kinds of files, which makes processing the data source a bit more difficult.

2.2.3 tl;drLegal

The platform tl;drLegal provides clear explanations of software licenses (tl;drLegal, n.d.-a). They summarize popular licenses and add a table to each license. The three columns describe what can and cannot be done and what the user must do. Their community has sent them over 1.5 million license summaries, and over one hundred thousand people have reviewed these summaries (tl;drLegal, n.d.-b). They clearly state that they do not provide legal advice, but they have teamed up with experts to provide the information as accurately as possible.

This is not comparable with our approach, nor do we share the same goals. They have a search function, but this does not serve the same purpose. We want to provide the canonical identifier for a license name to find out more about that license.

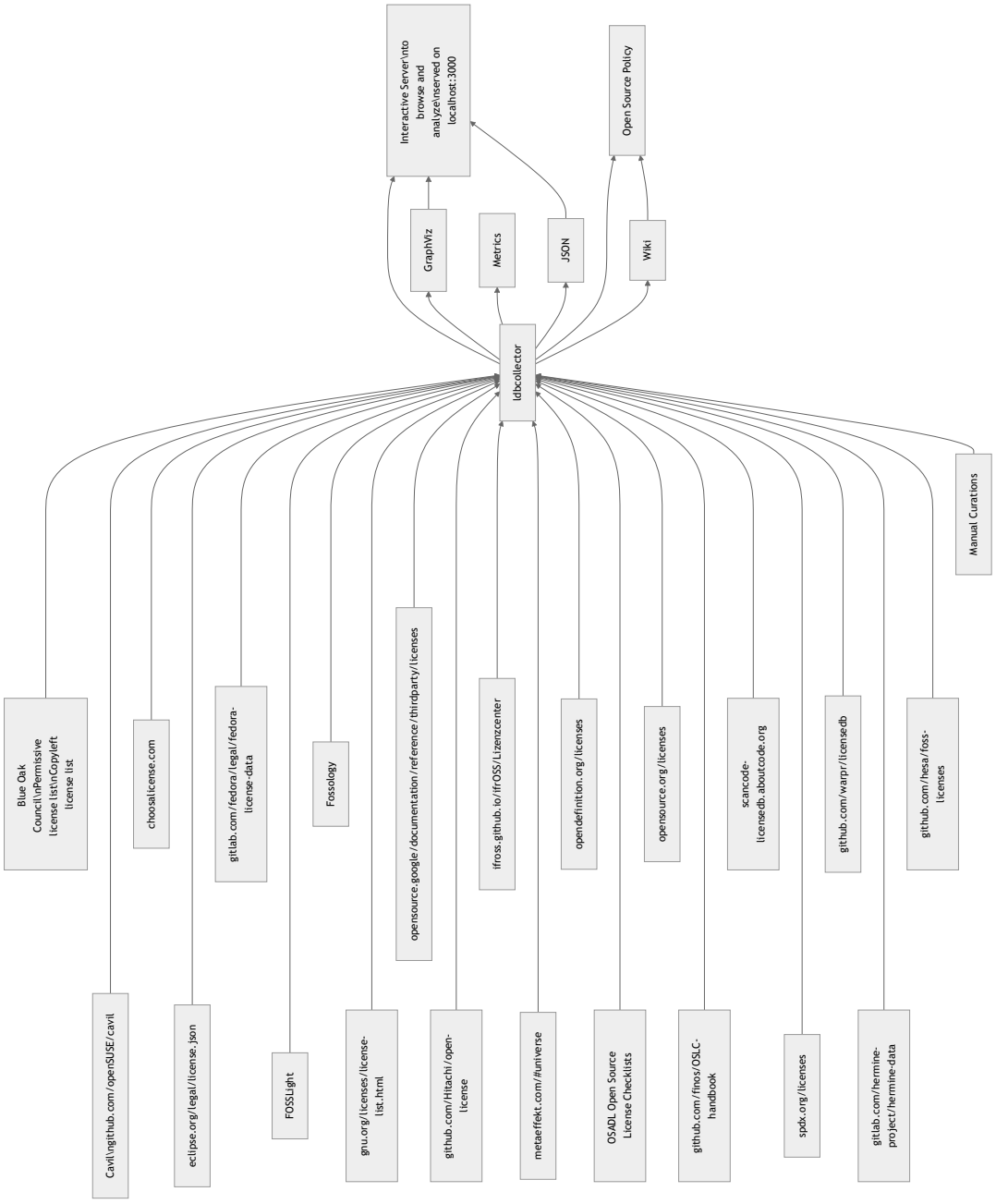


Figure 2.1: LDBcollector data chart

2.3 Data Sources for License Mapping

Now we look into data sources we want to use for LicenseLynx. First, we review the SPDX License List and explain their license review process. Then, we evaluate the ScanCode LicenseDB. At the end, asses the OSI Approved Licenses and explain their license review process. For all data sources we also show how they store their data.

2.3.1 SPDX License List

The SPDX License List is a list of commonly used licenses found in free and open source or collaborative software, data, hardware, or documentation (Lovejoy, 2023). The goal of the SPDX organization is to reliably identify licenses in software projects. The most notable attribute of the license list is the SPDX short identifier, which enables a machine-readable way to retrieve license information.

To include licenses, the SPDX License List follows some inclusion principles (Winslow, 2023). Their candidate license analysis is based on five definitive factors and five further factors that are less significant. The first definitive factor is that a submitted license must not match another existing license in the license list. For this, SPDX provides a matching guideline (Suriyawongkul, 2024). Secondly, all OSI-approved licenses are added to the SPDX License List. Thirdly, if the license applies only to the executable and the source code is not available, then it cannot be included in the license list. Fourthly, the license text must be stable, so it will not change once it is included in the license list. Lastly, license stewards should version new versions of the license, and no modifications are allowed to added licenses as the last factor already implied. Other factors include, for example, that the license has substantial use in many projects.

Licenses can be contributed via SPDX's own online tool or via an issue template for new license submissions (Lovejoy, 2024c). For the review process, a legal team must review any submission, which they communicate via comments in the GitHub issue or, if needed, bi-weekly calls. The legal team differentiates between two cases. The first case is that a new license is submitted and three legal team members agree that the submission meets the inclusion principles mentioned before and the SPDX-legal community does not raise any objections. If there are objections, then the license must be discussed in the bi-weekly meeting. The second case is that the license is already allowed in Debian or Fedora and that code is already licensed with the submitted license in one of the two distributions. Then only two legal

team members must approve because the license already meets important criteria, such as being substantially used in a major project.

Each license in the SPDX License List consists of eight fields (Lovejoy, 2024a). Each of these fields has a certain set of rules that must be applied.

The first field is the full name of the license. The full name should be consistent with well-known sources like OSI, and if the license has no obvious full name, then the short identifier with the suffix *license* or *exception* should be used. Also, words like *the* should be omitted, commas are not allowed, the word *version* should be shortened to the character *v* with no period or space between the version number or completely omitted, and abbreviations should not be included.

The next field is the short identifier, which is the core attribute of the license list to accurately identify a license. It consists of ASCII letters, digits, full stops, and the hyphen character and is based on the commonly used short names or acronyms. If this does not exist, then perhaps the project name or the copyright holder can be used. If a version number is applicable, it is appended after the abbreviation, separated with the hyphen character. Also, as the field name suggests, the short identifier should be kept as short as possible without breaking any other criteria.

The full name and the short identifier are the most important fields for our use case. The other fields can be helpful in other use cases, like the text field for the full license text.

We will use the SPDX License List as the basis for our project because it is well-defined and well-known to other projects and organizations. Package managers like Poetry recommend using the SPDX identifiers for licensing (Döring and Morignot, 2025). Also, the OSI adopted the SPDX identifiers in Spring 2011 (Lovejoy, 2024b).

Listing 2.3 shows an example of a license in the SPDX License List.

```
{
    "reference": "https://spdx.org/licenses/0BSD.html",
    "isDeprecatedLicenseId": false,
    "detailsUrl": "https://spdx.org/licenses/0BSD.json",
    "referenceNumber": 2,
    "name": "BSD Zero Clause License",
    "licenseId": "0BSD",
    "seeAlso": [
        "http://landley.net/toybox/license.html",
        "https://opensource.org/licenses/0BSD"
    ],
    "isOsiApproved": true
}
```

Listing 2.3: 0BSD license representation in the SPDX License List

2.3.2 ScanCode LicenseDB

The ScanCode LicenseDB is one of the largest collections of license data in the open-source space (Ombredanne et al., 2024). It contains over two thousand licenses, which are curated with a license text and other metadata. They include all OSI and SPDX licenses.

The database is organized with an index, and each license entry has a key that leads to its own file with more details of the license. They provide both the index and the license details as JSON and YAML files. A license is represented as shown in Listing 2.4.

```
{
    "license_key": "bsd-zero",
    "category": "Permissive",
    "spdx_license_key": "0BSD",
    "other_spdx_license_keys": [],
    "is_exception": false,
    "is_deprecated": false,
    "json": "bsd-zero.json",
    "yaml": "bsd-zero.yml",
    "html": "bsd-zero.html",
    "license": "bsd-zero.LICENSE"
}
```

Listing 2.4: 0BSD license representation in the index JSON file of ScanCode LicenseDB

The significant fields for our implementation are *license_key*, *spdx_license_key*, and *other_spdx_license_keys*. How we use them exactly is explained in Chapter 5.

Now, following the *json* field, we get even more license information for each license in the index file. The following Listing 2.5 shows an example of how a license is saved in detail.

```
{
    "key": "bsd-zero",
    "short_name": "BSD Zero Clause License",
    "name": "BSD Zero Clause License",
    "category": "Permissive",
    "owner": "Rob Landley",
    "homepage_url": "http://landley.net/toybox/license.html",
    "spdx_license_key": "0BSD",
    "other_urls": [
        "https://opensource.org/licenses/0BSD"
    ],
    "minimum_coverage": 70,
    "text": "..."
}
```

Listing 2.5: Detailed 0BSD license representation as JSON file in the ScanCode LicenseDB

These detailed license files will enrich our own data with different namings for one license name. The notable fields are *short_name* and *name*. Both are assigned by the scancode-toolkit (LicenseDB, n.d.).

2.3.3 Open Source Initiative Approved Licenses

The Open Source Initiative (OSI) is a public benefit corporation based in California with the goal to promote the significance of open-source communities and software for the industry (OSI, 2024c). One of their main activities is maintaining the Open Source Definition.

The Open Source Definition (OSD) consists of ten criteria that the distribution terms of open-source software must meet (OSI, 2024a). This definition is derived from the Debian Free Software Guidelines. Four of the ten criteria are regarding licensing:

- Distribution of License
- License Must Not Be Specific to a Product
- License Must Not Restrict Other Software
- License Must Be Technology-Neutral

The license review process ensures that licenses conform to the OSD, avoid redundancy, and maintain transparency (OSI, 2024b).
The process includes:

1. Submission: Licenses are submitted to the public license-review mailing list. Submitters are encouraged to first seek feedback via the license-discuss list.
2. Participation: The review is consensus-driven, with OSI board members participating in a personal capacity. Submitters must engage actively by responding to feedback.
3. Revisions: Licenses cannot be altered mid-review. To make changes, the current submission must be withdrawn and resubmitted.
4. Decision: Reviews typically conclude in 60 days or 30 days for revised licenses. The License Committee assesses consensus and recommends approval or rejection to the OSI board, which makes the final decision.
5. Legacy vs. New Licenses:
   Legacy licenses, which are used for five or more years by twenty or more projects, require proof of compliance with the OSD and details about existing use. New licenses must identify gaps they fill, compare to existing licenses, and detail any legal review.

6. Approval Criteria:
   All licenses must meet the OSD. New licenses must be reusable, neutral, unambiguous, and address unmet needs. Licenses cannot favor licensors or contain unapprovable restrictions.

We use the license list of OSI because they use the SPDX identifier as a key and provide more different names for a license. The Listing 2.6 shows a representation of GPL-2.0.

```
{
    "id": "GPL-2.0",
    "identifiers": [
      ...
    ],
    "links": [
      ...
      {
        "note": "OSI Page",
        "url": "https://opensource.org/licenses/GPL-2.0"
      }
    ],
    "name": "GNU General Public License, Version 2.0",
    "other_names": [],
    "superseded_by": "GPL-3.0",
    "keywords": [
      ...
    ],
    "text": [
      ...
    ]
}
```

Listing 2.6: JSON representation of GPL-2.0 in the OSI license list

The fields *name* and *other_names* can provide more different namings for a license, although some licenses are already covered by the SPDX license list.

3 Requirements

After reviewing the existing solutions and data sources in the previous chapter, we now define the requirements we expect the project to meet in the end. In general, the project should be maintained in one monorepo with different components, separated by folders. We identified four main components the project will have: a data source handler, programming libraries, the data, and a web API. We utilize UML use case diagrams to identify the functional requirements. To analyze the non-functional requirements, we use parts of the product quality model structure from the ISO standard 25010:2023 (International Organization for Standardization, 2023). To identify measurement metrics for the quality attributes, we use quality attribute scenarios (Bass et al., 2003).

3.1 Functional Requirements

To identify the functional requirements, we outline a use case model as described in Figure 3.1.

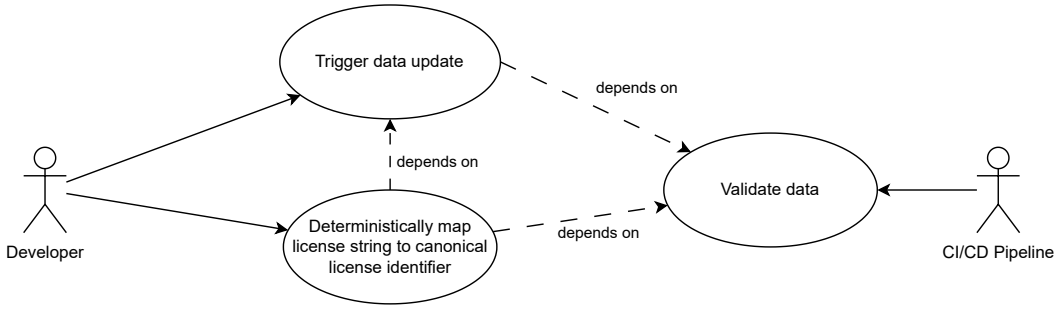


Figure 3.1: Use case diagram to identify the functional requirements

Once the data update is triggered, the CI/CD pipeline takes over to validate the data. This step ensures that any new data does not introduce inconsistencies or errors. Since validation depends on a fresh data update, the

process follows a logical sequence where incorrect information is prevented from entering the system.

Another key part of the model is deterministically mapping license strings to canonical license identifiers. Software licenses often come in different formats, e.g., the JSON formats of the different data sources in Chapter 2, and without a standardized approach, this can lead to inconsistencies. By mapping these licenses to a canonical form, the system reduces ambiguity. The mapping should be deterministic, so the that there is no probability of a false positive in the result. That means we exclude the use of regular expressions or artificial intelligence. However, this step relies on retrieving the most recent and validated data.

3.2 Non-functional Requirements

The product quality tree is separated into three sections. The first section contains the attributes of a software project. These attributes are connected to the concerns. Lastly, each concern is described by scenarios where the connection between them is labeled with the importance and the complexity. Our product quality tree is described in Figure 3.2.

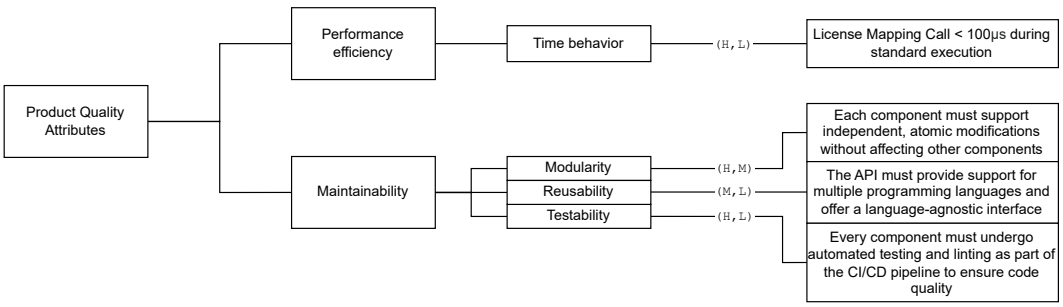


Figure 3.2: Product Quality Tree;
(X,Y)=(Importance,Complexity); H=High, M=Medium, L=Low

We want to mainly focus on performance efficiency and maintainability. To figure out how to measure the fulfillment of the requirements, we draw a quality attribute scenario for each scenario. Each scenario consists of a source, which is the description of the scenario, a stimulus, which triggers the scenario, an artifact, which is either a part of the system or the whole system, an environment under which the stimulus occurs, a response to take on the stimulus, and a response measure to test if the requirement is fulfilled (Bass et al., 2003).

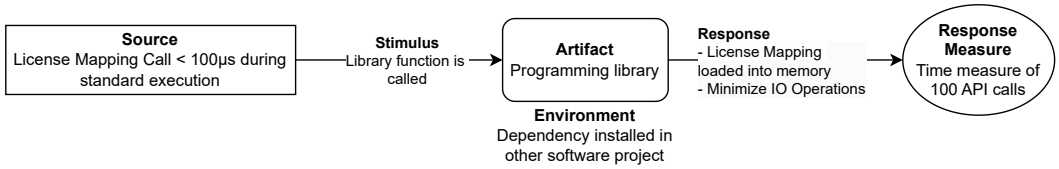


Figure 3.3: Quality attribute scenario for time behavior

The first scenario in Figure 3.3 is that a license mapping call must be under $100\mu s$ during normal execution. The execution environment is a computer with an Intel i5-1345U 13th Generation and 32 Gigabytes of RAM. We measure the response by calling the API one hundred times and taking the average because the API should not be a slow point when other projects need a canonical identifier.

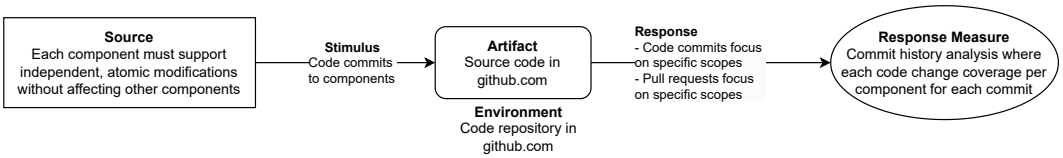


Figure 3.4: Quality attribute scenario for modularity

The scenario for modularity in Figure 3.4 is necessary because we want to manage the components in a monorepo. When code changes are made, the modifications should be atomic and not affect other components. We want to measure this by analyzing the commit history of the project and how many components are affected on average per commit.

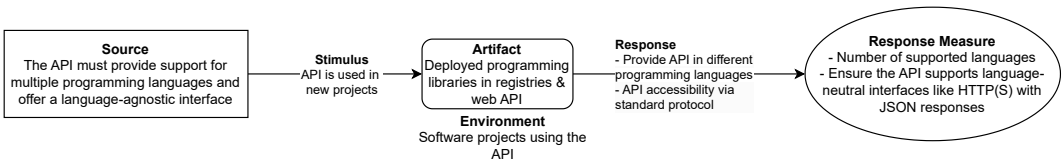


Figure 3.5: Quality attribute scenario for reusability

The scenario for reusability in Figure 3.5 should be tackled by providing the API in different programming languages and providing a language-agnostic approach. We measure this by simply counting the number of supported languages and evaluating the language-neutral interface.

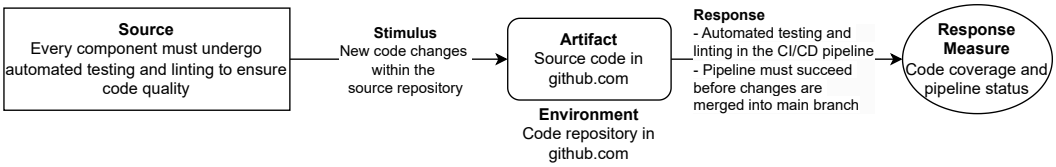


Figure 3.6: Quality attribute scenario for testability

The last scenario in Figure 3.6 is important for new code changes and features for our project. This is especially important from the open-source perspective if other developers outside the organization want to contribute to the project. This scenario is measurable by ensuring a certain threshold of code coverage by the tests and that the pipeline status is positive.

4 Architecture

In this chapter, we present our architecture for the project to meet the requirements from the previous Chapter 3. First, we outline how we handle the data. Then we explain our concept for the programming libraries and lastly our website and the web API.

4.1 Data Handling

In the following we describe our data handling architecture by detailing the data flow process, how we store the data and how we validate the data.

4.1.1 Data Flow

As we explored the data sources in Chapter 2, we use the SPDX License List, ScanCode LicenseDB, and OSI Approved License List as our data sources to enrich the data. After processing the data, we will save each license as a JSON file. This is the basis of our data. In the *MergeData* function, we then merge all the data to create a data mapping file which is used for the programming libraries and the web API. Figure 4.1 illustrate the whole data flow.

One extra data source is the users. Users can make contributions to license files by adding custom entries to the license file. This is especially valuable because there are numerous naming variations for licenses, which are not stored in the license lists.

To process the data sources, we have one abstract class *BaseDataUpdate* from which the classes *SpdxDataUpdate*, *ScancodeLicensedbDataUpdate*, and *OsiDataUpdate* inherit. Each class handles the data source of its class name. By encapsulating each data source, the process is extensible for new data sources. In Figure 4.2, the conceptual class diagram is shown.

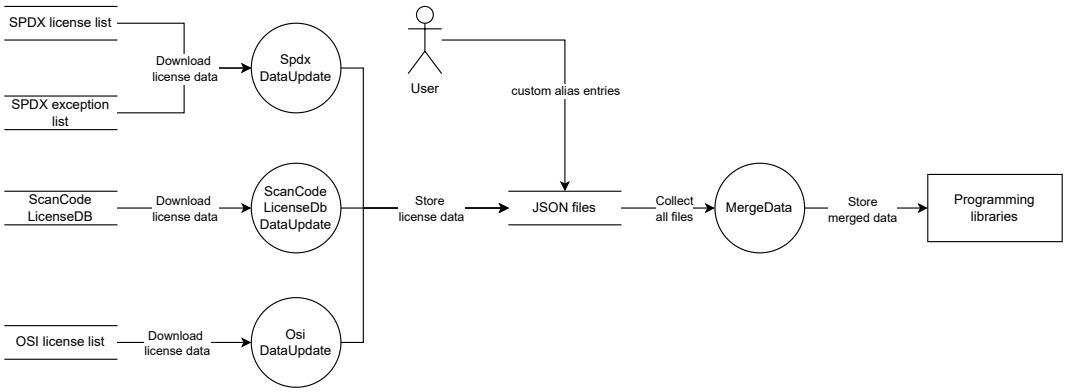


Figure 4.1: Data-flow diagram for the whole process

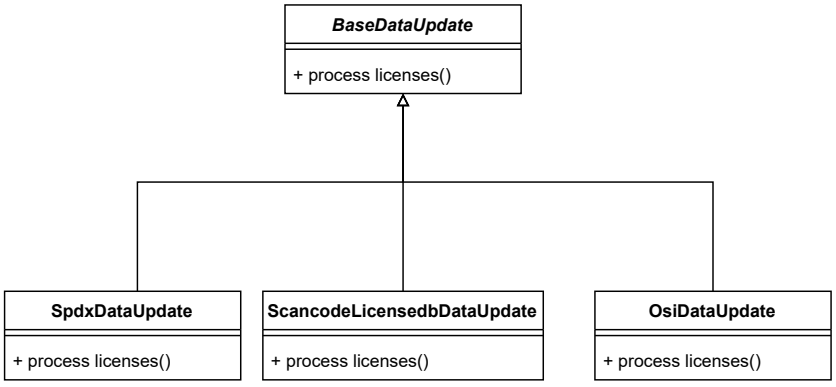


Figure 4.2: UML class diagram for processing the data from different data sources

4.1.2 Data Storage

The simplest way of storing the data is using a file system approach. We do not have complicated dependencies or relationships between entities; we only need to store the license data in files. Every license is stored in one file and versioned by Git. This makes contributions and modifications to the license data easier while also logging past changes.

We use GitHub as the remote repository and their issue system to track user modifications and suggestions for licenses.

4.1.3 Data Validation

A crucial step for our data is validation. To maintain good quality of data, some constraints or rules must be kept. This validation step is always executed in the pipeline if modifications or additions to the data are made.

In the following, we list all the rules for the data.

File name must be equal to canonical identifier

The file name of the JSON file must be equal to the canonical identifier. The reason is to keep consistency within the file system. If the canonical identifier changes, the file name must also change.

Globally unique alias

Each alias must be unique globally because we want a deterministic and exact return value. Therefore, duplicate aliases would not achieve this goal. If duplicates exist, it must be decided case by case how to handle this incident.

Canonical identifier and source must be equal

The canonical identifier must be the same as the one from the source. E.g., if *spdx* is the source of a license but the canonical identifier and the license identifier in the SPDX license list do not match, then perhaps the canonical identifier saved in the repository is not the right one. Having a clear source is important for further process steps after retrieving the canonical identifier, e.g., getting the license text.

Length maximum and forbidden characters

This should prevent cluttering the JSON files with useless or even malicious strings. Although the probability of having a negative effect from a string value within code is near zero, it is better to catch non-sense values automatically. The canonical identifier is checked for forbidden characters and maximum length, and the aliases and source are also checked for maximum length.

No empty fields

The JSON files in the repository should never have null values or empty strings. The reason is that we want to prevent the canonical identifier, the aliases, or the source from being null.

4.2 Programming Libraries

We provide programming libraries in Python, Java, and TypeScript so that the license mapping is also possible in code. Other projects like scanners that create SBOMs or tools that use the SBOMs can use these libraries to identify more licenses to achieve a more accurate SBOM.

4.2.1 Usage of Programming Libraries

The programming libraries are available in well-known open source package registries. The user is only provided with one API call, a method which takes in some license string and returns the canonical identifier.

The data for the programming libraries is a JSON file with all the mappings included. The libraries then have to read it into their internal representation of a map. This way, the libraries do not have to rely on an internet connection for a query.

The code looks similar to the following code snippet:

```
result = LicenseLynx.map("The Apache License, 2.0")
print(result)

# Output
result: canonical: "Apache-2.0",
        src: "spdx"
```

The implementation detail for each language varies, but the functionality is the same.

4.2.2 Singleton Pattern

Because we import a file, we want to prevent the data from being read from the file system every time it is used. Therefore, we will use the Singleton pattern to enable static loading because the data is only loaded once.

Although the Singleton pattern is sometimes criticized for reducing flexibility, complicating testability, and posing thread safety concerns (Safyan, n.d.), these issues do not apply in our While all these reasons are valid, the Singleton pattern makes sense in our case because we are only loading one file which does not change during runtime. However, we must guarantee that the libraries are thread-safe. The implementation of this pattern depends on the programming language, but generally, the Singleton

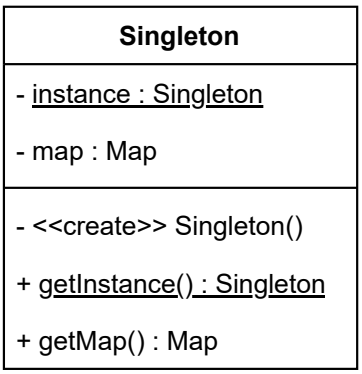


Figure 4.3: Singleton pattern

saves the license mappings as a Map and provides both a *getInstance()* and a *getMap()* method, as described in Figure 4.3. For testability improvements, a package-private constructor may be added if necessary to inject the Singleton with mock data.

4.3 Website and Web API

A documentation website is crucial for maintenance and contributions in open source projects. For that reason, we provide a website deployed with GitHub Pages and a static site generator to easily write the documentation in Markdown and deploy modifications automatically via GitHub Actions.

We also provide a programming language-agnostic API to access the data. It is similar to the API of ScanCode LicenseDB. The query is a license string and the return value is the canonical identifier with the source. All mappings are also available as one JSON, which is the same data that the programming libraries get.

5 Design and Implementation

This chapter describes our design and implementation of the architecture. We start with the implementation of the data handling, where we show the implementation of the data flow, data storage, data validation, and explain how we use this data for the API. Then we explain our implementation of the API in the form of the programming libraries and go into detail for the languages Python, Java, and TypeScript. After that, we go over the implementation of the website and with that also the implementation of the web API. Next, we figuratively show our CI/CD Pipeline in GitHub Actions. Lastly, we explain our semantic versioning.

5.1 Implementation of Data Handling

In the following we show our implementation details for our data handling process.

5.1.1 Data Flow

As shown in Chapter 4, we separate each data source with its own class, which inherits from *BaseDataUpdate*. The general process of retrieving the data and normalizing it into a standard format is very similar for each data source. We implemented the data update process in Python. In Figure 5.1, the conceptual class diagram is shown without implementation details. These details are shown as single class diagrams for each class in the following. The full class diagram is shown in Appendix B.

BaseDataUpdate Implementation

In Figure 5.2, we show the implementation details represented as a UML class diagram. We can generalize a few methods and variables in the abstract class *BaseDataUpdate*. The variable *_LOGGER* uses the *logging* framework of Python, which we primarily use for debug log messages,

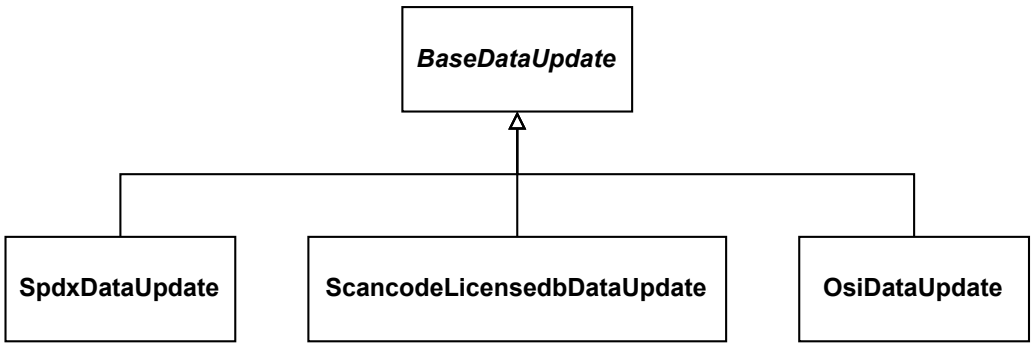


Figure 5.1: Conceptual class diagram of update data

_DATA_DIR is the relative path where all the license files are located, and *_src* provides the information on which data source we are processing. All three variables are private attributes.

The constructor has parameters *src* and *log_level*. The former is hard-coded in every subclass to indicate which data source is processed. The latter defines the granularity of log messages.

The methods *download_json_file*, *load_json*, and *delete_file* download a JSON file, load it as a Python dictionary, and delete the JSON file after the process is done.

The method *update_license_file* has as parameters the canonical identifier *canonical_id* to load the affected license file and a list of aliases for the license, named *aliases*. It takes all existing aliases of the license file, called *existing_aliases*, flattens them into a one-dimensional list, and also appends the canonical identifier. Then it loops through the *aliases* list and adds an alias to the key-value pair with the data source as the key if it does not exist in the one-dimensional list *existing_aliases*.

If a license file for a canonical identifier does not exist, the method *create_license_file* creates a new JSON file. It has as parameters the canonical identifier *canonical_id* and the list of aliases *aliases*, similar to the method *update_license_file*. Subsection 5.1.2 explains in detail how the format for storing the license information looks.

The last method *get_file_for_unrecognized_id* tries to find the license file iteratively if an identifier is not found as a canonical identifier in the data. The parameter is a list of different naming variations provided by the data source, and for each variation, all license files are searched for a full match. This is a time-consuming method and it rarely happens but is necessary,

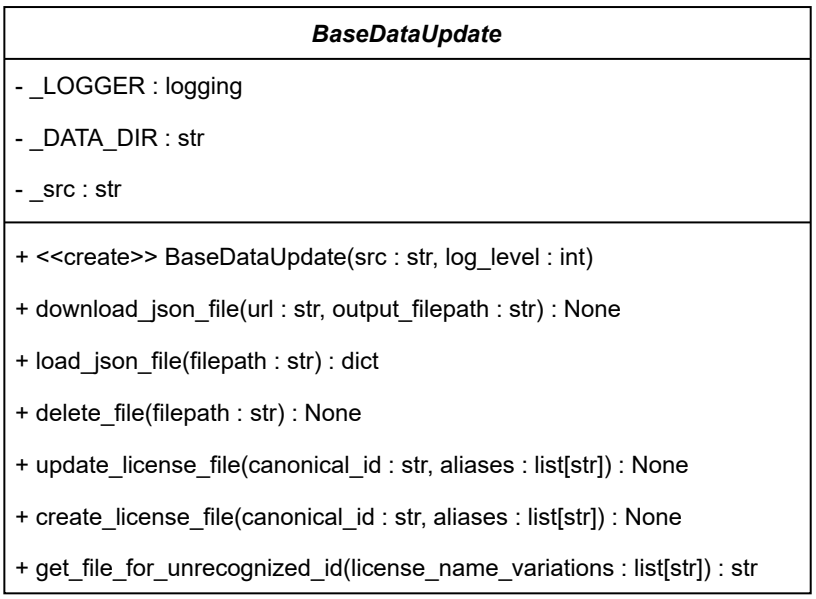


Figure 5.2: Class diagram for BaseDataUpdate

e.g., if the canonical identifier changes.

SPDX Implementation

The subclass *SpdxDataUpdate*, as seen in Figure 5.3, processes the SPDX License List and the SPDX Exception List. We presented the JSON objects of the license list in Chapter 2. Now we briefly explain which fields we use for our process. The SPDX License List and the SPDX Exception List have two fields of interest, respectively. One is the field *name*, which is the commonly used name for the license, where the key name is the same for both lists. This field is used as an alias for the canonical identifier. The second and more important field in the SPDX License List is *licenseId*, and for the SPDX Exception List, it is *licenseExceptionId*. This is the canonical identifier for our data.

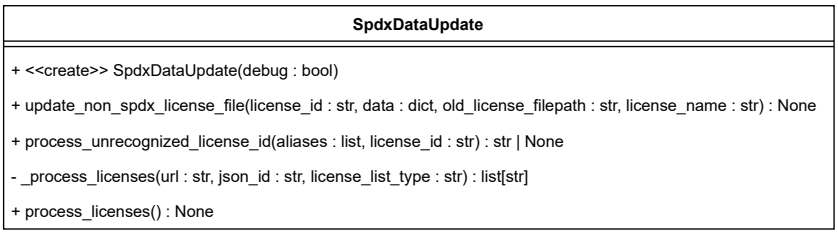


Figure 5.3: Class diagram for SpdxDataUpdate

The class itself has no attributes, but when calling the constructor, the user can enable the debug log. Inside the constructor, the attribute *_src* from the parent class is passed with the value *spdx*, and the log level is also passed to the parent class.

The method *process_licenses* is the entry point for processing the SPDX files. Here, the private method *_process_licenses* is called for both license lists. In Figure 5.4, we describe the whole process in an activity diagram.

In the method *_process_licenses*, we download the license list and create a list of all files in the data folder. Then, for each entry in the license list, a lookup in the files list happens to see if the license file already exists or must be created. That is possible because the canonical identifier and the JSON file name are the same. If the file exists, then a lookup in the source occurs to see if *update_license_file* of the parent class can be called or *update_non_spdx_license_file* if the source is not SPDX. If the license could not be found, the method *process_unrecognized_license_id* is called. If the license could not be found, it is created by calling *create_license_file*.

When *process_unrecognized_license_id* is called, all different variations collected for the specific license from the license list are aggregated, and then *get_file_for_unrecognized_id* is called. If the file name could not be found, the license identifier provided by the license list is returned. But if it is found, then the same process as in *_process_licenses* happens, and either *update_license_file* or *update_non_spdx_license_file* is called based on the source of the license file.

The method *update_non_spdx_license_file* is used when a license file already exists but the source is not SPDX. SPDX has the highest priority because it is widely adopted, so we always update a license file if the source is not SPDX. That means we have to update the canonical identifier, the source, and the aliases. Also, the file name must change to the new canonical identifier. The alias entries, which have already stored the metadata SPDX is providing, are reassigned to SPDX.

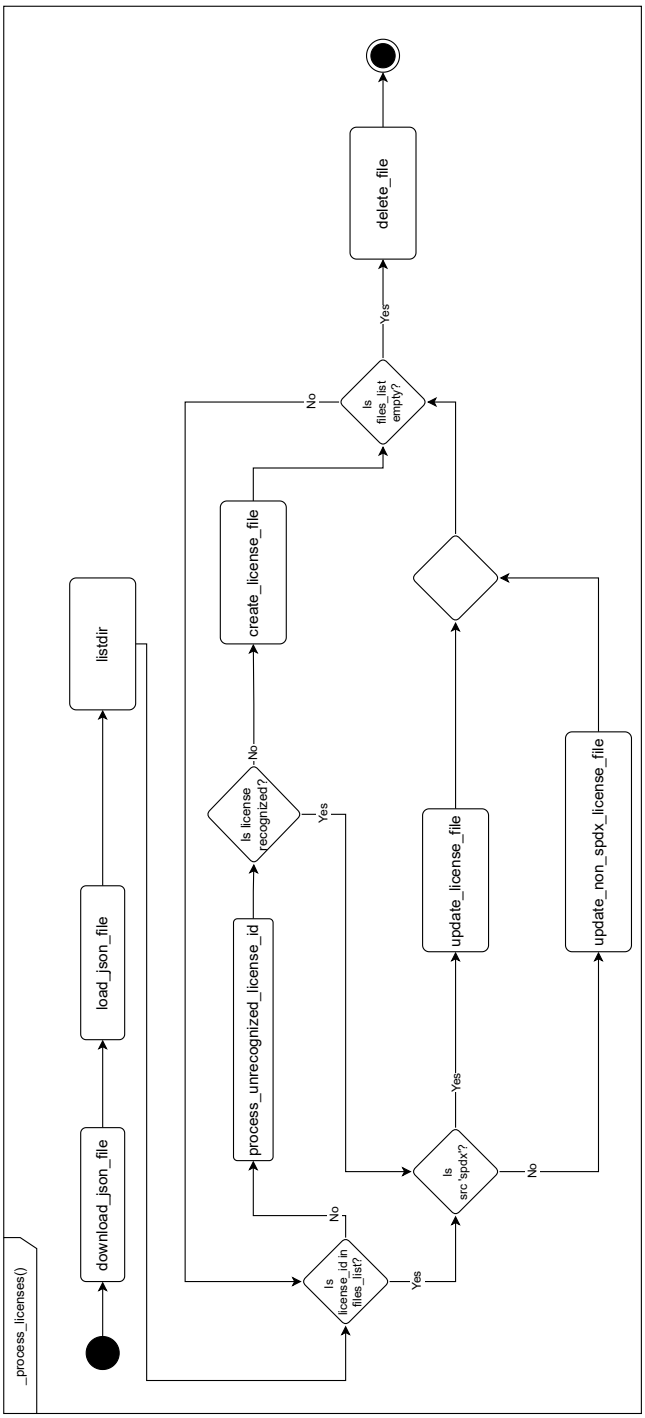


Figure 5.4: UML activity diagram for processing SPDX

ScanCode LicenseDB Implementation

The ScanCode LicenseDB provides both a JSON index file with all their licenses and a more metadata-enriched JSON file for each license, as explained in Chapter 2. The class diagram is shown in Figure 5.5.

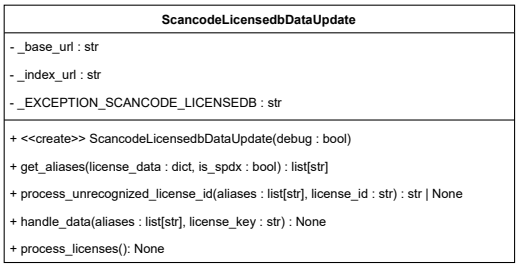


Figure 5.5: Class diagram for ScancodeLicensedbDataUpdate

The field of interest in the index file is *license_key*. This key leads to the more specific JSON files. In the file, we always have the fields *key*, which is the same as *license_key*, *short_name*, and *name*. In most cases, it also has the field *spdx_license_key* and often *other_spdx_license_keys*. The value of *spdx_license_key* is either the same as the SPDX identifier or the key given by ScanCode LicenseDB with the prefix *LicenseRef*. If the latter is the case, then the canonical identifier for the license is the key given by ScanCode LicenseDB.

The subclass *ScancodeLicensedbDataUpdate* has three private attributes: the base URL of ScanCode LicenseDB *_base_url*, the URL to the index file *_index_url*, and a list of licenses we skip due to their ambiguity to other licenses, *_EXCEPTION_SCANCODE_LICENSEDB*.

In the method *process_licenses*, we download the index file and iterate through each license. If the license is in the exception, we continue with the next license. After downloading the more detailed license file, we check if the field *spdx_license_key* exists. If that is the case, then a further check ensures that the SPDX identifier is the value and not the *LicenseRef* prefix identifier created by ScanCode LicenseDB. If there is a valid SPDX identifier, then we already have the license file and only update it with aliases provided by ScanCode LicenseDB. For the case with the prefix as well as for the case without the field *spdx_license_key*, we also collect the aliases and then call the method *handle_data*. The activity diagram in Figure 5.6 describes the process.

To collect the aliases, the method *get_aliases* is called. It has as parameters the license data from the JSON file as a Python dictionary and a boolean

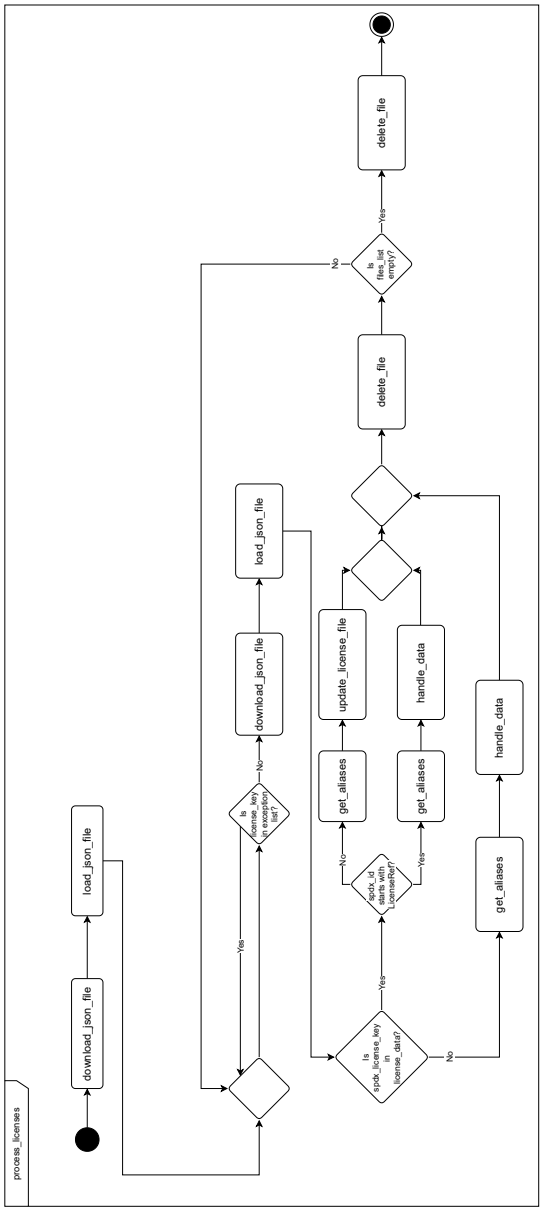


Figure 5.6: UML activity diagram for processing ScanCode LicenseDB

value indicating if the file has a valid SPDX identifier. The values of the keys *short_name* and *name* are stored in variables with the same names as the keys and then saved to the set *aliases*. If the boolean is true, we also add the key from ScanCode LicenseDB to *aliases*. Then we check if *spdx_license_key* exists and if it starts with *LicenseRef*, and if that is true, then we add it to *aliases*. Lastly, if the key *other_spdx_license_keys* exists, we add the list to *aliases*. At the end, we convert the set to a list and return it. The reason for using a set is to eliminate duplicates.

After we collect the aliases, we call the method *handle_data*, which takes as parameters the aliases and the key provided by ScanCode LicenseDB. In this, we search for the file with the key because it is the canonical identifier for all license files where SPDX is not the canonical identifier. If the file is found, we update the license file with aliases; if not, then we try to search for the file with all naming variations we have by calling *process_unrecognized_license_id*, similar to *SpdxDataUpdate*. This method returns the key if no file was found, so we create a new license file if that happens.

OSI Implementation

The important fields in the OSI license list are simpler than in ScanCode LicenseDB. We want to use the license identifier *id*, the field *name*, and their list *other_names*. They also provide a list of links for every license, from which we extract the identifier from the URL. The class diagram is shown in Figure 5.7.

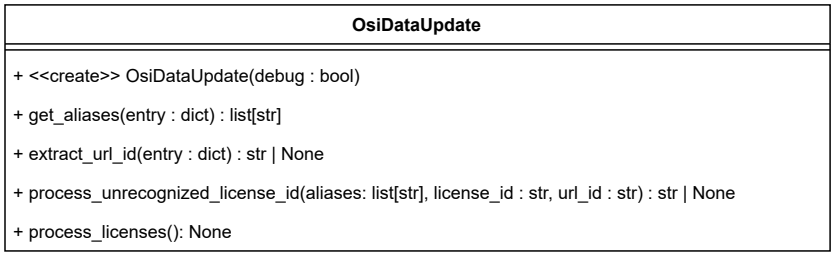


Figure 5.7: Class diagram for OsiDataUpdate

In the method *process_licenses*, we iterate through all licenses. In Figure 5.8, we describe it as an activity diagram. We store the identifier as *license_id* and extract the identifier OSI uses in the URL to the license text hosted on their website as *url_id* with the method *extract_url_id*. Then we collect the aliases with the method *get_aliases*, where we aggregate the

data from the fields *name* and *other_names* into one list. If neither the *license_id* nor *url_id* are found as canonical names in the existing data, we search through all license files with all naming variations we collected by calling the method *process_unrecognized_licenses*. If a license file is not found, we do not create a new license file because it needs manual verification. The reason is that all OSI licenses should be in the SPDX License List, as explained in Chapter 2. If either *license_id* or *url_id* were found as canonical identifiers, we update the license file with the new data.

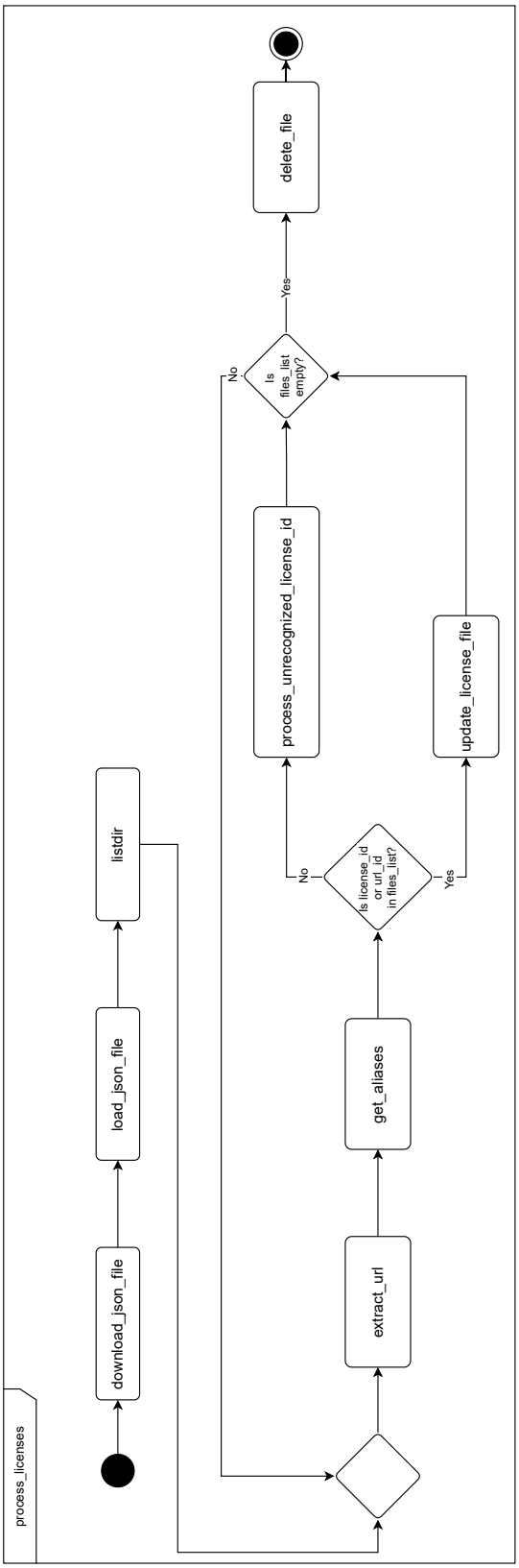


Figure 5.8: UML activity diagram for processing OSI

5.1.2 Data Storage

After retrieving the data from the different data sources, the information is saved as JSON files. Each license is a JSON file, where the file name is the same as the canonical identifier. In addition, the source of the canonical identifier and all its aliases are saved. The *aliases* value is a JSON object, where each key is the source and the value is a list of aliases retrieved from the source. Each file also has a *custom* key, where contributors can put in an alias for the license.

A JSON file for a license looks structure-wise as follows:

```
{
"canonical": "0BSD",
"aliases": {
    "spdx": [
        "BSD Zero Clause License"
    ],
    "custom": [
        "0BSD License",
        "BSD Zero Clause",
    ],
    "scancode-licensedb": [
        "bsd-zero"
    ]
},
"src": "spdx"
}
```

Listing 5.1: JSON file of 0BSD license, shortened

All changes made to a JSON file are tracked by git and GitHub. When a user suggests changes to a specific license, they must do it with an issue or a merge request. Doing this, changes and suggestions can be understood by future users. Before changes are included in the master branch, the changes have to be validated.

5.1.3 Data Validation

We program the implementation of the data validation in Python. Each constraint is handled by one method. For the constraint *No empty fields*, we exclude the field *custom* in *aliases*. Every license file has this field to provide a starting point for manual contributions, but for most licenses, it is

empty. In Appendix A, we show the detailed code for each constraint. The script always runs in the pipeline when changes are made to the data.

5.1.4 Merging Data Before Building and Publishing

During the pipeline runtime, we create resource files for the programming libraries before they are published. The file is not committed before for each library because we want to prevent individual changes to certain resources. If, e.g., a contributor changes the resources for the Python library, the changes would not likely be transferred to the Java or TypeScript library because the contributor perhaps does not need them. The workflow for data contributions is to edit the license file in the data folder.

To generate the resources during the pipeline, a script is executed. This script reads all JSON files from the data folder and merges them into a large JSON file. Each release-build-job then copies this file to its resource folder, and then the libraries are built and released. The structure of the merged JSON file is as shown in Listing 5.2.

```
{
    "0BSD": {
        "canonical": "0BSD",
        "src": "spdx"
    },
    "BSD Zero Clause License": {
        "canonical": "0BSD",
        "src": "spdx"
    },
    ...
}
```

Listing 5.2: JSON file of merged data, shortened

The file consists of a dictionary where each key has a JSON object which consists of the canonical identifier and the source of the canonical identifier. This is useful, e.g., for getting the license text. When knowing what the canonical identifier and the source are, the request to retrieve the license text is very clear.

5.2 Programming Libraries Implementation

5.2.1 Java

In Java, we implement four classes: *LicenseLynx*, *LicenseMapSingleton*, *LicenseObject*, and *LicenseDataLoader*. Among these, only the *LicenseLynx* class provides a public API through the *map()* method. The remaining classes are either private or package-private, as illustrated in Figure 5.9.

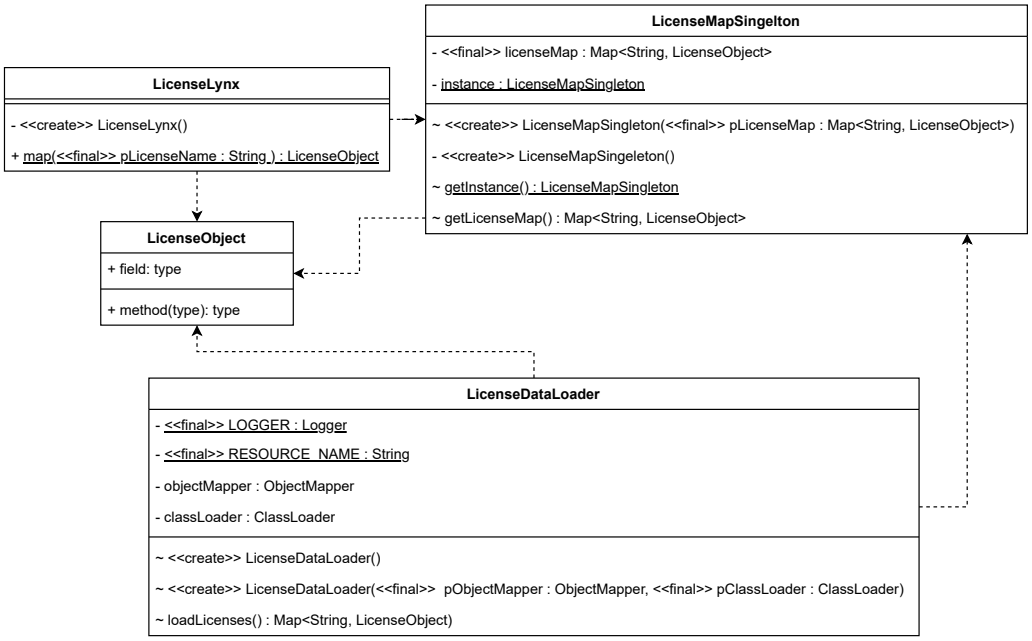


Figure 5.9: UML class diagram of Java implementation

The *LicenseDataLoader* processes the JSON file containing the merged data and returns a *Map<String, LicenseObject>*. We provide two different constructors for testability, allowing the injection of the class loader and the object mapper with mocked objects. To import the JSON file, we use the native class loader to avoid hard-coded file paths.

The *LicenseMapSingleton* class utilizes the *loadLicenses()* method from the *LicenseDataLoader* when it is instantiated for the first time. We opted for lazy initialization for the singleton class. The JSON file is loaded as an object only when the *getInstance()* method is called. The singleton is also thread-safe due to the use of the *synchronized* keyword. The Listing 5.3 demonstrates the implementation of the *getInstance()* method.

```
static synchronized LicenseMapSingleton getInstance()
{
    if (instance == null)
    {
        instance = new LicenseMapSingleton();
    }
    return instance;
}
```

Listing 5.3: Singleton implementation in Java

The *LicenseObject* class represents the Java equivalent of the JSON object, with two properties: *canonical* and *src*. To clarify that this is an immutable JSON object, we use annotations from the *fasterxml* dependency. The *@Immutable* annotation indicates that the object is read-only, while the *@JsonProperty* annotation designates the global variables as JSON properties, facilitating correct parsing to JSON.

LicenseLynx provides the only static public method, *map()*. Users simply need to call `LicenseLynx.map()` to utilize this method. This method calls *getInstance()* to obtain the singleton instance and then *getLicenseMap()* to retrieve the object with merged data. If the input string cannot be found in the data, null is returned; otherwise, the corresponding *LicenseObject* is returned. The Listing 5.4 shows the implementation.

```
@CheckForNull
public static LicenseObject map(@Nonnull final String pLicenseName)
{
    LicenseMapSingleton licenseMapSingleton = LicenseMapSingleton.
       getInstance();
    Map<String, LicenseObject> licenseMap = licenseMapSingleton.
       getLicenseMap();

    return licenseMap.get(pLicenseName);
}
```

Listing 5.4: Java implementation of map method

For local development, we added an extra step in the *build.gradle* file when building the project. While we do not commit any merged JSON file, the build configuration can be specified. In Gradle, it is possible to read system environment variables. In GitHub Actions, the environment variable *CI* is always set to true. In the *build.gradle*, this variable can be referenced as

isCI. We use this mechanism to download the most recent merged JSON file from the internet. The Listing 5.5 roughly shows how we implement it.

```
tasks.register("checkAndDownloadJson") {
    def resourcesDir = file("src/main/resources")
    def jsonFile = file("$resourcesDir/merged_data.json")
    def downloadUrl = "https://licenselynx.org/json/latest/mapping.
       json"

    doLast {
        // Check if file already exists; if not download it
    }
}

tasks.named("processResources") {
    def isCI = System.getenv("CI")

    if (!isCI.present) {
        dependsOn("checkAndDownloadJson")
    }
}
```

Listing 5.5: Gradle config for local development

5.2.2 Python

The Singleton implementation in Python uses lazy loading, meaning that the JSON file is loaded when the *map* method is called. We also implemented it to be thread-safe by using metaclasses. The class diagram in Figure 5.10 illustrates our implementation.

In Python, everything is an object, including classes. These objects have a default metaclass, *type*. It is possible to define a custom metaclass and specify classes to use it. This allows us to manipulate the behavior of instantiating an object of the class using the custom metaclass.

Our custom metaclass, *Singleton*, has a private dictionary, *_instances*, to store all instances when *__call__* is invoked, and a private *_lock* variable to ensure thread safety. Every time a new instance of a class using the *Singleton* metaclass is created, the *__call__* method of the metaclass checks if the instance already exists by looking it up in the dictionary. If the instance does not have an entry in the dictionary, a new entry is created. After the check, the value of the dictionary key is returned, where the value

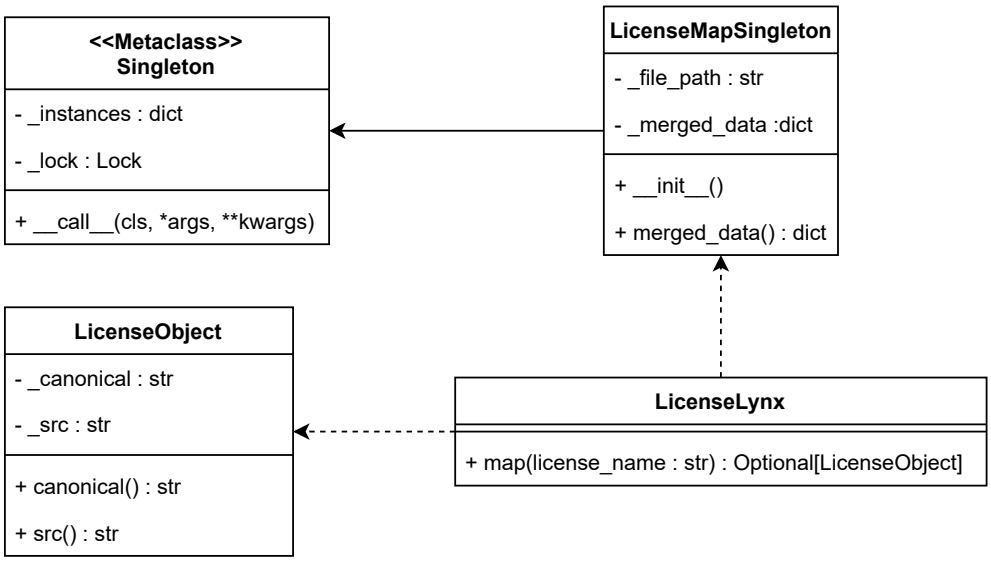


Figure 5.10: UML class diagram of Python implementation

is the Singleton instance and the key is the class being instantiated. Listing 5.6 shows the implementation.

```
1 def __call__(cls, *args, **kwargs):
2     with cls._lock:
3         if cls not in cls._instances:
4             instance = super().__call__(*args, **kwargs)
5             cls._instances[cls] = instance
6     return cls._instances[cls]
```

Listing 5.6: Python implementation of __call__ method

To ensure the immutability of the JSON file, the private attribute *_merged_data* in the *LicenseMapSingleton* class can only be retrieved by calling the *merged_data* method. With the *@property* decorator, the method can be called like a variable. In contrast to Java, loading a JSON file in Python is straightforward and is automatically cast to a Python dictionary.

The *LicenseObject* class uses the same *@property* functionality as *LicenseMapSingleton* for the private attributes *_canonical* and *_src*. The *LicenseObject* is also the return value of the *map* method in the *LicenseLynx* class. *Optional* is a type hint equivalent to the notation `LicenseObject | None`.

5.2.3 TypeScript

The implementation of the TypeScript library is shown in Figure 5.11. This implementation does not require a Singleton because JSON files can be imported and used like regular JavaScript objects. This convenient feature of TypeScript and JavaScript greatly simplifies the implementation.

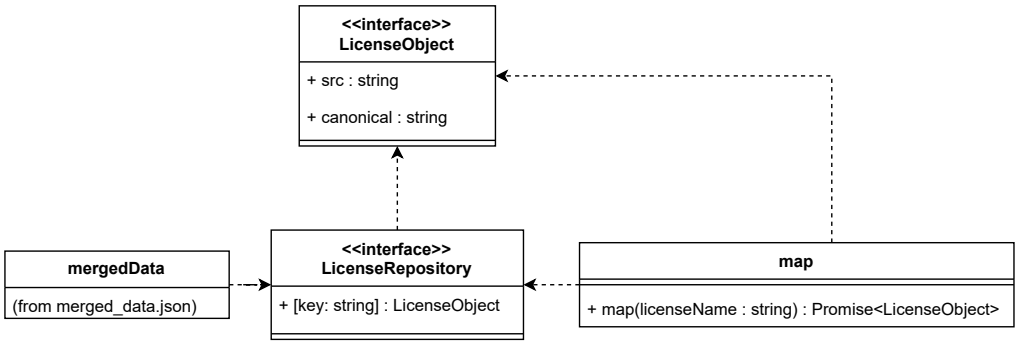


Figure 5.11: Class diagram of TypeScript implementation

We introduce two interfaces to ensure type safety. The first interface is *LicenseRepository*, which represents the merged data file, where the key is a string and the value is the *LicenseObject* interface. This interface has two *readonly* string variables, *canonical* and *src*.

In contrast to the Python and Java implementations, in the TypeScript implementation, we export functions and interfaces rather than classes. Thus, we only export the *LicenseObject* interface and the *map* function to be accessible to other modules.

The return value of *map* is a *Promise* with the type *LicenseObject*. Promises enable asynchronous operations, providing better control over the flow of code.

5.3 Website and Web API Implementation

5.3.1 Documentation

An important aspect of an open-source project is the documentation. We use *mkdocs* as the static site generator, which is then published with GitHub Pages. The documentation follows the typical structure of many open-source projects:

- Introduction Brief introduction, outlining the problem we aim to solve and an overview of the other pages.
- FAQ Explanation of design decisions and differences from other projects.
- How LicenseLynx works Description of the process of obtaining licenses from different data sources.
- Installation Installation guide for the different libraries.
- Usage Instructions on how to use the different libraries and the web API.
- Contribution Contribution guidelines and an explanation of our versioning.

5.3.2 Web API

We aim to provide a programming language-agnostic method for linking arbitrary license names with canonical identifiers. To reduce maintenance and costs, we do not host a web server ourselves; instead, the web API consists of license objects as JSON files in a directory, published via GitHub Pages.

To retrieve a license object as a JSON file, the HTTP GET call is `/api/license/{license_name}.json`. The placeholder *license_name* is the parameter, similar to the parameter for the programming languages. Because the request must be URL encoded, all slash characters `/` must be replaced with the underscore character `_`.

It is also possible to get the entire merged data as a JSON file, which we already do for the local Gradle build process. To get the latest version, `/json/latest/mapping.json` will return it.

5.4 GitHub Actions

For continuous integration and continuous deployment, we use GitHub Actions to test, lint, and deploy our code. Each programming library and script has its own workflow, and deployment has its own workflow as well. Each workflow is described in a separate YAML file.

Figure 5.12 shows a simplified activity diagram of the pipeline for deploying the programming libraries. It is triggered only when a git tag is created with the regular expression `'v[0-9]+.[0-9]+.[0-9]+'`. The deployment for each library occurs in parallel.

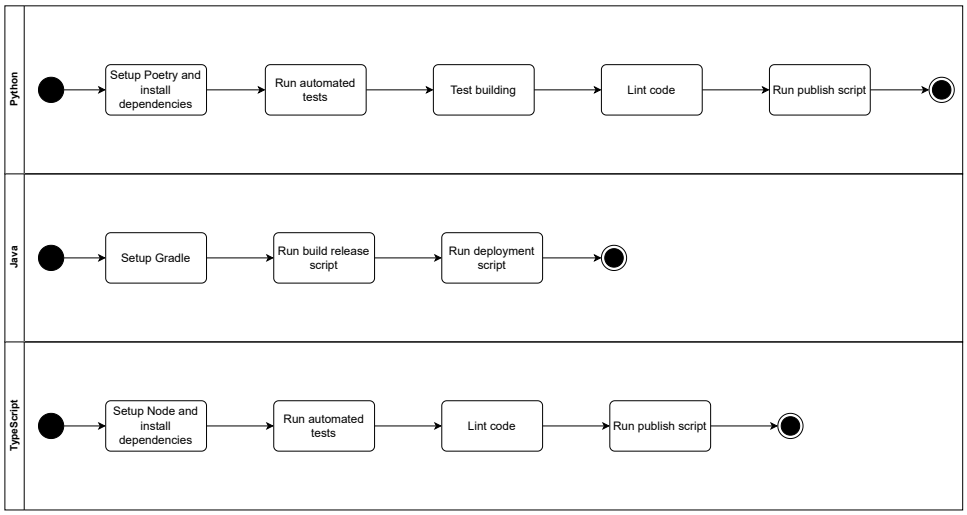


Figure 5.12: Simplified UML activity diagram for deployment

For the web API, we generate the files in the pipeline. We host it with GitHub Pages to save on hosting and maintenance costs. Figure 5.12 shows the pipeline for creating the web API. First, we build the website with mkdocs, and then we run a Python script that uses the merged data to generate a JSON file for each license object. To deploy on GitHub Pages, the content must be placed into a folder named *public*.

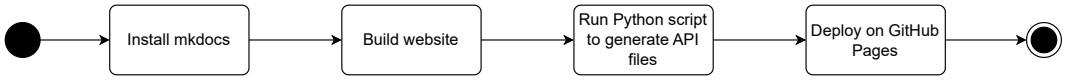


Figure 5.13: Simplified UML activity diagram for web API generation

Lastly, the data is validated in the pipeline. This workflow is triggered only if changes are made in the data folder. Figure 5.14 shows the process. Not

only is the data validated, but our statistics are also updated. The statistics include the top five licenses with the most aliases, the total number of licenses, and the number of mappings.

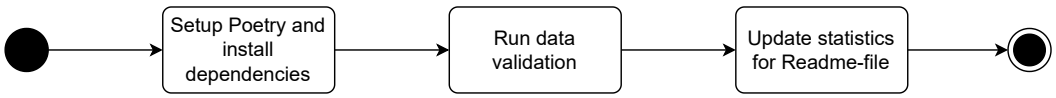


Figure 5.14: Simplified UML activity diagram for data validation

5.5 Semantic Versioning

Choosing the right system for versioning a software product is crucial for clients and other developers. Inconsistencies in versioning or arbitrary version bumps confuse the meaning behind a version update. For this reason, different versioning conventions were invented. The most prominent are Semantic Versioning (SemVer) and Calendar Versioning (CalVer).

The SemVer convention is a versioning system consisting of three numbers (Preston-Werner, 2023). The scheme is *(MAJOR.MINOR.PATCH)*. *PATCH* describes a backward-compatible bug fix, *MINOR* is a backward-compatible feature, and *MAJOR* indicates a breaking change. If the software uses SemVer, the API must be public, and once a version is released, it cannot be changed. The version number can only be incremented.

The reason for using this strict convention is that clients know exactly what a version increase means for the component. Ideally, a *PATCH* or *MINOR* version increase will not break the software using the updated component, and reading release notes should not be necessary. This also makes it easy to have bots running in the background that update the components used in a project without any issues.

CalVer is a versioning convention based on dates rather than numbers (Hashemi, 2019). The scheme is not as strict as SemVer, but it provides a terminology for the dates, e.g., *YYYY* is a full year and *YY* is a short year relative to the year 2000. The flexible convention may not be as machine-readable as SemVer, but the use case is different. Using CalVer makes sense for projects where older versions are not intended to be supported, such as video games or operating systems.

At first glance, CalVer seems suitable for our use case because the programming libraries may always be used with the most recent data. However, we decided that the data should not be part of the API, and data updates to

the libraries should only be noted as a *PATCH* version increase. We want to have a strict, unambiguous versioning concept, which many other programming libraries also use, and package managers handle SemVer better than other concepts.

To be more precise about how we manage the versions of our programming libraries: all libraries share the same version. The disadvantage is that changes for one programming library will bump up the version for all the others, even if they are not affected. However, the advantage is that we only need to use one version tag. We expect that most updates will be data updates of the merged JSON file. These data updates count as *PATCH* updates to the libraries, as mentioned before. This implies that users should never rely on the data output itself. We see it as the libraries only request data with the given output.

6 Evaluation

In this chapter, we evaluate the implementation. First, we present general results about the statistics of the data. Then, we assess whether the implementation meets the functional and non-functional requirements.

6.1 General Results

We used three data sources to build a foundation of mappings for different license names to canonical identifiers. We successfully integrated SPDX, ScanCode LicenseDB, and OSI. In total, we have over 8500 mappings with more than 2300 licenses, and both numbers are increasing. We also added custom entries for several popular licenses that were not provided by the data sources. For ScanCode LicenseDB, we had to create an environment file with exceptions because some licenses share similar names but use different keys and must be reviewed manually. However, this affected only 24 licenses.

6.2 Functional Requirements Evaluation

In Chapter 6, we identified two actors which are the developer and the CI/CD Pipeline, and three use cases for these actors.

The first use case is that the developer triggers a data update. There are two ways for the developer to trigger a data update. The first is to run the update classes, i.e., *SpdxDataUpdate*, *ScancodeLicensedbDataUpdate*, and *OsiDataUpdate*. By running these scripts, the most recent license lists from SPDX, ScanCode LicenseDB, and OSI are pulled, and the data is updated if applicable. The second solution is to manually add entries to the *custom* field in the *aliases* object. Both solutions should then be contributed as pull requests because the use case depends on validating the data in the CI/CD Pipeline.

Validating the data is the second use case, which is triggered by the CI/CD Pipeline when changes are made to the data. We established rules for our data that must always be fulfilled; otherwise, the contribution is not valid. These rules are:

- File name must be equal to the canonical identifier.
- Globally unique alias.
- Canonical identifier and source must be equal.
- Length maximum and forbidden characters.
- No empty fields.
- Programming libraries.

These rules must be valid for all JSON fields, except the *custom* field, which can be empty. This exception is necessary because the *custom* field is always created when a license file is added to the data to give contributors a starting point.

The last use case is that the developer maps some license strings to canonical identifiers deterministically. This use case depends on the previous two use cases. To enable the developer to succeed, we provide programming libraries in Python, Java, and TypeScript. Additionally, we provide a programming language-agnostic API with HTTP requests to JSON files. The developer enters the license string they want to map, and the API returns an object with the canonical identifier and the source of the canonical identifier. This process is deterministic because some licenses may differ by only one character, and any uncertainty could return an incorrect result. Each license is stored as its own JSON file, containing the canonical identifier, source, and aliases. For the API, all aliases are merged into one JSON file, where each key is the alias and the value is the object with the canonical identifier and source. Storing all licenses in JSON files where each license file contains the canonical identifier and its aliases ensures that the mappings are deterministic.

6.3 Non-Functional Requirements Evaluation

We will review the four quality attribute scenarios and evaluate them with the response measures to determine if we have addressed the concerns.

The first scenario is time behavior. Our response measure is to record the time of one hundred API calls, with the average time being less than $100\mu s$

per API call. We conducted the measurements for all three libraries. Figure 6.1 shows the graphs for each library. The x-axis represents the API call from the first time LicenseLynx is called until the 100th time. The y-axis indicates the time elapsed for each API call. We also calculate the median and plot it as a dashed line.

The results show that all libraries require significant time to initialize and read the mappings. Java needs $290515.00\mu s \approx 0.290s$, Python $14275.60\mu s \approx 0.0142s$, and TypeScript $229.10\mu s \approx 0.000229s$. Although Java takes the longest for the initial setup, the median time for an API call is $0.4\mu s$, for Python $0.9\mu s$, and for TypeScript $8.3\mu s$.

If we ignore the initialization time and focus on the time for normal execution, it becomes clear that the runtime per API call fluctuates more for TypeScript. In Figure 6.2, we plot the graphs without the initialization cost in one graph. The Python and Java implementations are very close together, whereas the TypeScript implementation is slower by an order of magnitude of around nine compared to Python and by an order of magnitude of twenty compared to Java. However, looking at the bigger picture, these time discrepancies do not significantly impact real use cases. Our goal was to be under 100 microseconds, and we achieved it.

The second scenario is modularity. The response measure is the average number of affected modules per commit, as each folder in the monorepo is one module. In Figure 6.3, we show the affected modules per commit since the start of the project. We have a total average of 1.38 which is lower than expected but a number to be very content with, shown with the yellow dashed line. The orange line indicates the change in the average over time, where we can see that after the first minimum viable product (MVP), the graph decreases rapidly and hovers steadily around the total average. We have several spikes, notably three high spikes: one at the beginning at the second commit and two at the end at commits 122 and 145. The first spike is due to the initial commit where we committed the first MVP. The second spike was adding copyright notices in the source files before open-sourcing the project, and the third spike was modifying the configuration files of the modules for the CI/CD pipeline in GitHub. At the end, the affected modules fluctuate due to the migration from GitLab to GitHub. We also show the version tags, and we can see that before we decided to push all modules with one tag, most version tags for a specific module only affected their own module. By changing the versioning to one tag for all modules, we see that two modules are involved, which is the version change in the project configuration file for the Python library and

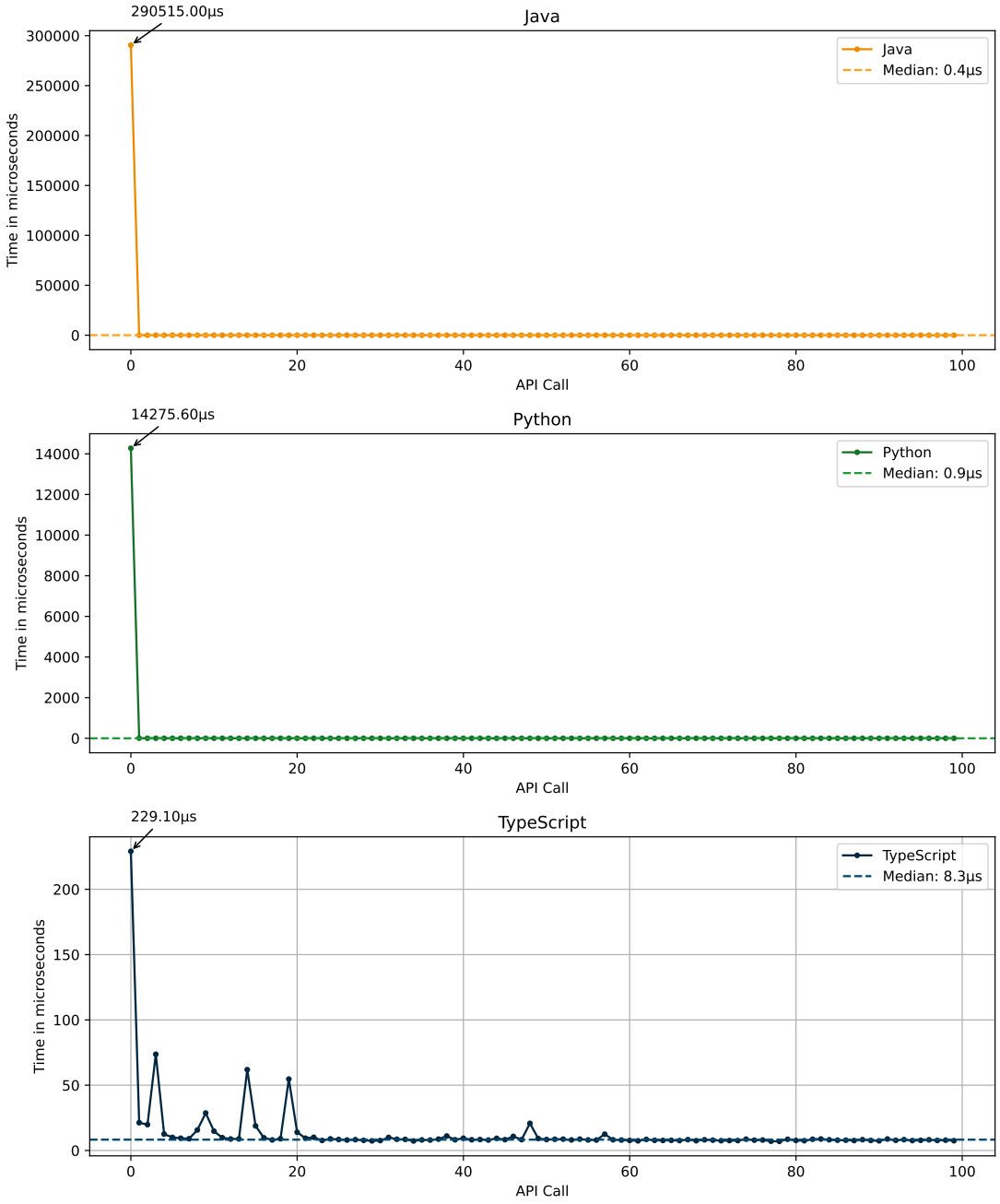


Figure 6.1: Time behavior in three separate plots for each library

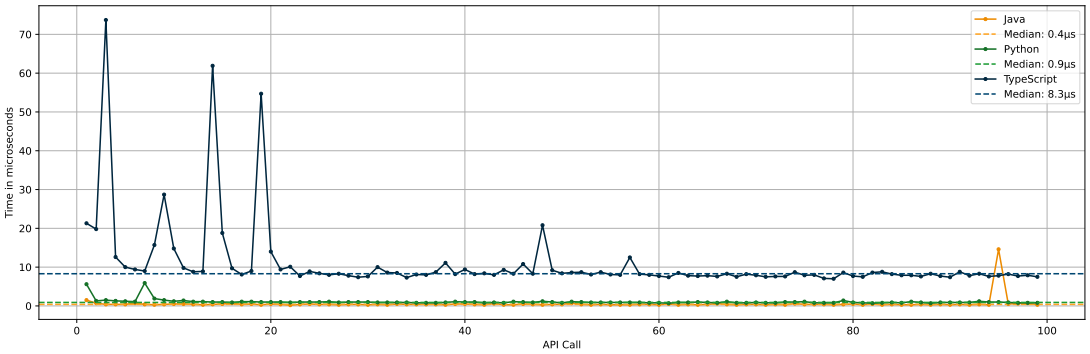


Figure 6.2: Time behavior without initialization

the TypeScript library. In Appendix C, we provide the code for how we conducted the measurements.

The third scenario is reusability. The response measure is the number of supported languages and a language-agnostic interface. We implemented the programming libraries for Python, Java, and TypeScript and published them on public registries. We also provide installation and usage guides for each programming library. For the language-agnostic case, we provide a web API that returns JSON files as responses via HTTP GET requests. The web API is deployed with GitHub Pages, so the requests must be URL encoded.

The last scenario is testability. Our response measure is the code coverage of the tests and the pipeline status when code is pushed to GitHub. We successfully implemented unit tests for all three programming libraries and the scripts. In the root README file of LicenseLynx, we added badges for each module's code coverage and pipeline status. This provides a quick status overview of the modules. We configured GitHub Actions so that automated tests are always run when changes are made to the affected module when these changes are pushed to the main branch or a pull request is opened. For the scripts we have coverage of 93.4%, for the Python and TypeScript component 100% and for Java 96.4%.

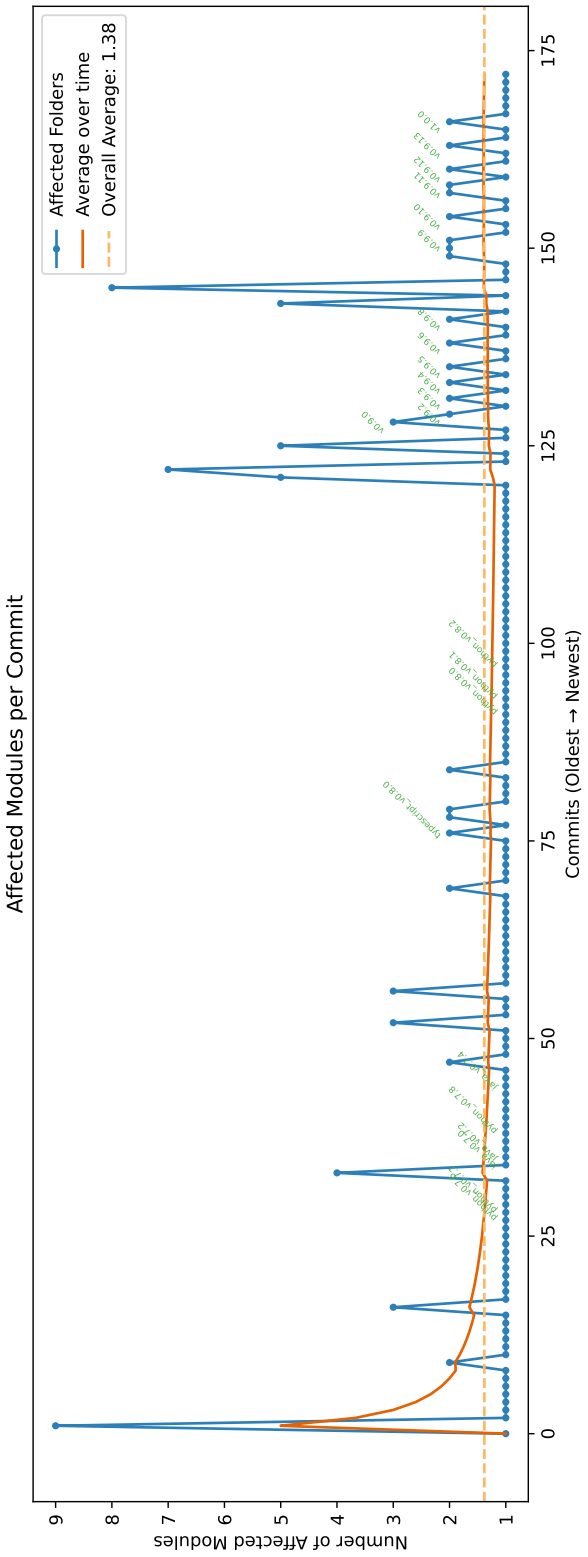


Figure 6.3: Affected modules per commit

7 Conclusions

The last chapter summarizes our key findings, outlines the limitations of LicenseLynx and provides a future prospect for the project.

7.1 Summary of Key Findings

In this thesis, we developed LicenseLynx, a deterministic solution for mapping software licenses to canonical identifiers. Our research led to several key findings. Firstly, we successfully integrated three major data sources: the SPDX License List, ScanCode LicenseDB, and OSI Approved License List. This integration resulted in a comprehensive database containing over 2300 licenses and 8500 mappings. Secondly, we implemented robust data handling and validation mechanisms to ensure the accuracy and consistency of the license data. This process includes downloading, processing, and validating license data from multiple sources. Additionally, we developed programming libraries in Python, Java, and TypeScript to facilitate the use of LicenseLynx in various software projects. These libraries provide a simple API for mapping license strings to canonical identifiers. Furthermore, we created a language-agnostic web API to provide access to the license data, allowing users to retrieve license information via HTTP requests. Lastly, our system was designed to be efficient and modular, with performance tests showing that the average time for an API call is well below the target of 100 microseconds. The modular design ensures that changes to one component do not affect others.

The whole project is open source and the code is on GitHub. Additionally, the project home page is on its own domain, https://licenselynx.org.

7.2 Limitations of LicenseLynx

Despite the successful implementation and promising results, there are several limitations to this study. One significant limitation is the case sensitivity of LicenseLynx. The current implementation requires the input string to exactly match the stored alias to retrieve the canonical identifier, which can lead to issues if there are variations in the case of license names.

Another limitation is the coverage of data sources. While we integrated three major data sources, there are still many licenses, especially commercial and flavored licenses, that are not covered, limiting the comprehensiveness of the data.

Additionally, some licenses require manual verification due to ambiguities or similarities with other licenses, which can be time-consuming and may introduce delays in updating the database. Lastly, the accuracy and completeness of LicenseLynx depend on the data provided by external sources. Any changes or inaccuracies in these sources can affect the quality of our database.

7.3 Future work

To further enhance the capabilities and coverage of LicenseLynx, several recommendations are proposed. Firstly, to avoid ambiguity and overwrites, we could extend the data structure with a field for rejected mappings as a countermeasure for wrongly mapped licenses. Secondly, expanding the integration of additional data sources, including commercial and less common licenses, would increase the comprehensiveness of the data. We could integrate the LDBcollector to enhance our data. Lastly, we want to provide for the `json/latest/mapping.json` different versions corresponding to the published components, e.g. `json/1.0.0/mapping.json`. Under this aspect publishing the Javadoc for each version is beneficial for the Java community. Furthermore, the monorepo structure of LicenseLynx could be separated in own repository for more clarity.

Appendices

A Code Implementation for data validation

```
def check_json_filename():
    for filename in os.listdir(DATA_DIR):
        if filename.endswith(JSON_EXTENSION):
            filepath = os.path.join(DATA_DIR, filename)
            with open(filepath, 'r') as f:
                data = json.load(f)
                canonical_name = data.get("canonical")
                if canonical_name != filename[:-5]:
                    logger.error(f"JSON filename '{filename}' does
                       not match canonical name '{canonical_name}'")
```

```
def check_unique_aliases():
    all_aliases = {}
    for filename in os.listdir(DATA_DIR):
        if filename.endswith(".json"):
            filepath = os.path.join(DATA_DIR, filename)
            with open(filepath, 'r') as f:
                data = json.load(f)
                aliases = data.get("aliases", [])
                access_aliases(aliases, all_aliases, filename)

    for alias, filenames in all_aliases.items():
        if len(filenames) > 1:
            logger.error(f"Alias '{alias}' is not unique globally.
               Affected file: {filenames}")
```

```
def check_src_and_canonical(spdx_license_list: list,
    spdx_exception_list: list):
    for filename in os.listdir(DATA_DIR):
        if filename.endswith(JSON_EXTENSION):
            filepath = os.path.join(DATA_DIR, filename)
            with (open(filepath, 'r') as f):
                data = json.load(f)
                canonical_name = data.get("canonical")
                if (canonical_name in spdx_license_list or
                    canonical_name in spdx_exception_list) and data["
                    src"] != "spdx":
                    logger.error(f"If src is SPDX, canonical name '{
                       canonical_name}' must be in SPDX license list"
```

```
                    )
10              elif (canonical_name not in spdx_license_list and
11                    canonical_name not in spdx_exception_list) and
                        data["src"] == "spdx":
12                  logger.error(f"Canonical name '{canonical_name}'
                        is in SPDX license list but source is not '
                        spdx'.")
```

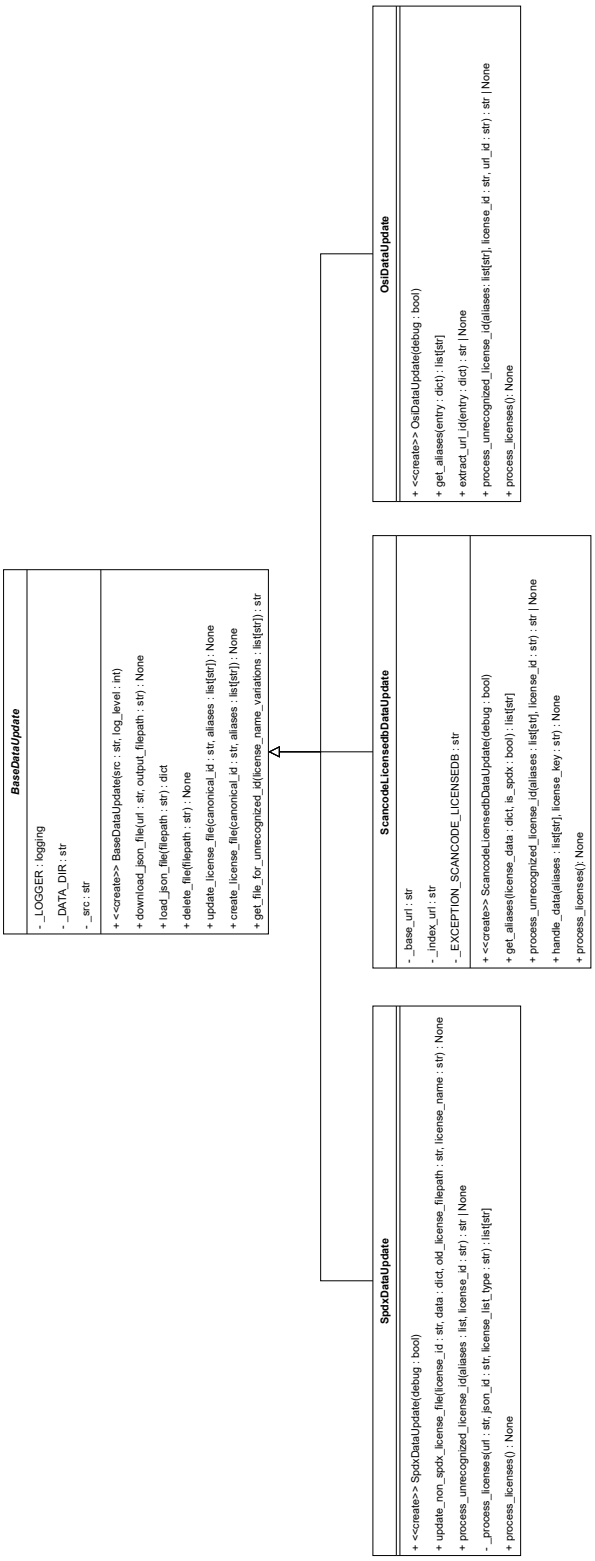
```
1  def check_length_and_characters():
2      forbidden_characters_canonical = {"#", "$", "%", "=", "[", "]", "
           ?", "<", ">", ":", "/", "\\", "|", "*", " "}
3
4      max_length = 100  # Adjust the maximum length limit as needed
5      for filename in os.listdir(DATA_DIR):
6          if filename.endswith(JSON_EXTENSION):
7              filepath = os.path.join(DATA_DIR, filename)
8              with open(filepath, 'r') as f:
9                  data = json.load(f)
10                 canonical_name = data.get("canonical")
11                 aliases = data.get("aliases")
12                 src = data.get("src")
13
14                 # Max length check
15                 if len(canonical_name) > max_length:
16                     logger.error(f"Canonical name '{canonical_name}'"
17                                  f"exceeds maximum length limit of {
                                        max_length} characters")
18                 if any(len(alias) > max_length for alias in aliases):
19                     logger.error(f"At least one of the aliases
                           exceeds maximum length limit of {max_length}
                           characters "
20                                  f"in the file {filename}")
21                 if len(src) > max_length:
22                     logger.error(f"Source {src} exceeds maximum
                           length limit of {max_length} characters")
23
24                 # Forbidden char check
25                 if any(char in forbidden_characters_canonical for
                       char in canonical_name):
26                     logger.error(f"Canonical name '{canonical_name}'
                           contains forbidden characters")
```

```
1  def check_no_empty_field_except_custom():
```

```
    for filename in os.listdir(DATA_DIR):
        filepath = os.path.join(DATA_DIR, filename)
        with open(filepath, 'r') as f:
            data = json.load(f)

            if not data["canonical"]:
                logger.error(f"Field 'canonical' in '{filename}' is
                    empty.")
            if not data["aliases"]:
                logger.error(f"Field 'aliases' in '{filename}' is
                    empty.")
            for key, value in data["aliases"].items():
                if not value and key != "custom":
                    logger.error(f"Alias list in '{filename}' for
                        field '{key}' is empty.")
            if not data["src"]:
                logger.error(f"Field 'src' in '{filename}' is empty."
                    )
```

B Full class diagram for data update



C Code for modularity analysis

```
import subprocess
import matplotlib.pyplot as plt
import numpy as np


def get_git_commits():
    """Retrieve a list of all commit hashes in the repository."""
    result = subprocess.run(["git", "log", "--pretty=%H"], capture_output=True, text=True)
    # Git returns newest commit first; reverse it to have oldest commit first.
    return result.stdout.strip().split("\n")[::-1]


def get_changed_files(commit):
    """Retrieve a list of changed files for a given commit."""
    result = subprocess.run(["git", "show", "--name-only", "--pretty=format:", commit], capture_output=True, text=True)
    return [line for line in result.stdout.strip().split("\n") if line]


def get_affected_folders(files, folder_groups):
    """Determine which folders are affected by a list of changed files."""
    affected_folders = set()
    for file in files:
        folder = file.split("/")[0]  # Extract the top-level folder
        for group_name, paths in folder_groups.items():
            if folder in paths:
                affected_folders.add(group_name)
                break
        else:
            affected_folders.add(folder)  # If not in a group, add as individual folder
    return affected_folders


def get_git_tags():
    """Retrieve a dictionary mapping commit hashes to tags."""
```

```
35      result = subprocess.run(["git", "tag", "--list", "--format=%(
            objectname) %(refname:short)"],
36                              capture_output=True, text=True)
37      tags = {}
38      for line in result.stdout.strip().split("\n"):
39          parts = line.split(" ", 1)
40          if len(parts) == 2:
41              tags[parts[0]] = parts[1]
42      return tags
43
44
45  def analyze_commits():
46      """Analyze all commits and store the number of affected folders.
            """
47      folder_groups = {
48          "Python": {'python'},
49          'Typescript': {'typescript'},
50          "Java": {"java"},
51          "Data": {"data"},
52          "Scripts": {"scripts"},
53          "Website": {"website"}
54      }
55
56      commits = get_git_commits()
57      commit_counts = []
58      avg_commit_counts = []
59
60      # Compute statistics in ascending order (oldest to newest)
61      for i, commit in enumerate(commits):
62          changed_files = get_changed_files(commit)
63          affected_folders = get_affected_folders(changed_files,
                folder_groups)
64          commit_counts.append(len(affected_folders))
65          avg_commit_counts.append(np.mean(commit_counts))  # Moving
                average over time
66
67      return commits, commit_counts, avg_commit_counts
68
69
70  def plot_results(commits, commit_counts, avg_commit_counts, tags):
71      """Plot the number of affected folders per commit."""
72      overall_average = np.mean(commit_counts)
73
```

```
74      print(f"Average number of affected folders: {overall_average}")
75
76      plt.figure(figsize=(15, 5))
77      plt.plot(range(len(commits)), commit_counts, marker='.',
            linestyle='-', label='Affected Folders', color='#2c7fb8')
78      plt.plot(range(len(commits)), avg_commit_counts, linestyle='-',
            color='#e66101', label='Average over time')
79      plt.axhline(y=overall_average, color='#fdb863', linestyle='--',
            label=f'Overall Average: {overall_average:.2f}')
80
81      # Add labels for tags with 45-degree rotation
82      for i, commit in enumerate(commits):
83          if commit in tags:
84              plt.annotate(tags[commit], (i, commit_counts[i]),
85                           textcoords="offset points", xytext=(0, 5),
86                           ha='center', fontsize=5, color='#4daf4a',
                                 rotation=45)
87      plt.xlabel("Commits (Oldest -> Newest)")
88      plt.ylabel("Number of Affected Folders")
89      plt.title("Affected Folders per Commit")
90      plt.legend()
91      plt.savefig('commit-history.pdf', bbox_inches='tight')
92      plt.show()
93
94
95  if __name__ == "__main__":
96      commits, commit_counts, avg_commit_counts = analyze_commits()
97      tags = get_git_tags()
98      plot_results(commits, commit_counts, avg_commit_counts, tags)
```

D Time-behavior measurement

```python
# Python
for key in random_keys:
    req_start = time.perf_counter()
    LicenseLynx.map(key)
    req_end = time.perf_counter()

    response_times.append((req_end - req_start) * 1_000_000) # Convert to microseconds
```

```java
// Java
for (String licenseName : randomLicenses) {
    double startTime = System.nanoTime();
    LicenseObject licenseObject = LicenseLynx.map(licenseName);
    double endTime = System.nanoTime();

    responseTimes.add((endTime - startTime) / 1000); // Convert to microseconds
}
```

```typescript
// TypeScript
for (const key of randomKeys) {
    const start = performance.now();
    await map(key);
    const end = performance.now();

    responseTimes.push((end - start) * 1000); // Convert ms to microseconds
}
```

Java	Python	TypeScript
290515	14275.59997	229.1
1.5	5.600042641	21.3
0.7	1.299893484	19.8
0.4	1.500127837	73.7
0.4	1.300126314	12.6
0.4	1.200009137	10
0.7	1.100124791	9.4
0.4	5.89992851	9
0.3	1.900130883	15.7
0.4	1.499895006	28.7
0.5	1.200009137	14.8
0.5	1.40001066	9.8
0.4	1.099891961	8.8
0.3	1.100124791	8.9
0.4	1.000007614	61.9
0.5	1.000007614	18.8
0.7	0.899890438	9.7
0.4	1.099891961	8.1
0.5	1.099891961	9
0.3	1.000007614	54.7
0.5	1.000007614	14
0.4	1.000007614	9.4
0.2	0.900123268	10.1
0.4	1.000007614	7.7
0.5	1.000007614	8.9
0.4	1.000007614	8.4
0.4	1.099891961	8
0.3	0.900123268	8.3
0.4	1.000007614	7.8
0.4	0.999774784	7.4
0.3	1.000007614	7.6
0.4	0.899890438	10
0.3	0.899890438	8.6
0.5	0.900123268	8.5
0.4	0.899890438	7.3
0.4	0.800006092	8.1

Continued on next page

Java	Python	TypeScript
0.4	0.800006092	8
0.4	0.800006092	8.7
0.3	0.900123268	11.1
0.5	1.100124791	8.2
0.5	1.000007614	9.4
0.4	1.000007614	8.2
0.3	0.800006092	8.4
0.6	0.899890438	8
0.3	0.800006092	9.3
0.3	1.099891961	8.3
0.5	1.000007614	10.8
0.4	0.900123268	8.3
0.3	1.200009137	20.8
0.4	1.000007614	9.2
0.4	0.800006092	8.4
0.7	1.099891961	8.6
0.4	1.000007614	8.7
0.3	0.899890438	8.1
0.4	0.899890438	8.7
0.3	0.899890438	8.1
0.3	0.899890438	8
0.3	0.900123268	12.5
0.4	0.900123268	8.2
0.4	0.800006092	8
0.3	0.800006092	7.7
0.5	0.699888915	7.4
0.3	0.900123268	8.5
0.4	0.900123268	7.8
0.3	1.000007614	7.7
0.4	0.899890438	7.8
0.6	0.800006092	7.6
0.4	1.099891961	8.3
0.3	0.800006092	7.5
0.4	0.800006092	8.2
0.4	0.899890438	7.9
0.3	0.800006092	7.5
0.4	0.800006092	7.6

Continued on next page

Java	**Python**	**TypeScript**
0.3	1.000007614	7.6
0.6	1.000007614	8.7
0.4	1.100124791	7.9
0.4	0.800006092	8
0.4	0.800006092	7.1
0.3	0.800006092	7
0.4	1.40001066	8.6
0.5	0.900123268	7.7
0.3	0.800006092	7.5
0.5	0.699888915	8.6
0.3	0.800006092	8.8
0.4	0.899890438	8.2
0.3	0.800006092	7.9
0.3	1.099891961	7.9
0.4	0.900123268	7.6
0.3	0.699888915	8.3
0.4	0.899890438	7.7
0.3	0.900123268	7.4
0.4	0.900123268	8.8
0.3	0.900123268	7.8
0.4	1.200009137	8.3
0.3	1.000007614	7.6
14.6	1.000007614	7.8
0.5	0.900123268	8.2
0.6	0.800006092	7.7
0.7	0.800006092	7.9
0.3	0.800006092	7.5

References

AG, S. (2016). Retrieved March 28, 2025, from https://www.fossology.org/wp-content/uploads/sites/39/2018/11/Module-C-Features-201610.pdf

Bass, L., Clements, P., & Kazman, R. (2003, January). Software architecture in practice, second edition.

Cornec, B. (2012, June). The fossology project. Retrieved March 28, 2025, from https://events.static.linuxfound.org/images/stories/pdf/lcna_co2012_cornec.pdf

Döring, R., & Morignot, A. (2025, March). The pyproject.toml file. Retrieved March 28, 2025, from https://python-poetry.org/docs/pyproject#license

European Commission. (2024, December). Shaping europe's digital future. Retrieved March 28, 2025, from https://digital-strategy.ec.europa.eu/en/policies/cyber-resilience-act

Fahey, E. (2024). The evolution of eu–us cybersecurity law and policy: On drivers of convergence. *Journal of European Integration*, *46*(7), 1073–1088. https://doi.org/10.1080/07036337.2024.2411240

GitHub. (2025, March). Retrieved March 28, 2025, from https://api.github.com/repos/maxhbr/LDBcollector

Gobeille, R. (2008). The fossology project. *Proceedings of the 2008 international working conference on Mining software repositories*, 47–50. https://doi.org/10.1145/1370750.1370763

Hashemi, M. (2019, July). Calendar versioning. Retrieved March 28, 2025, from https://calver.org/overview.html

House - Oversight and Reform; Science, Space, and Technology. (2020, December). Internet of things cybersecurity improvement act of 2020. Retrieved March 28, 2025, from https://www.congress.gov/116/plaws/publ207/PLAW-116publ207.pdf

Huber, M. (2021, December). Ldbcollector/aliases/aliases.csv. Retrieved March 28, 2025, from https://github.com/maxhbr/LDBcollector/blob/generated/aliases/aliases.csv

Huber, M. (2022, September). Stats. Retrieved March 28, 2025, from https://github.com/maxhbr/LDBcollector/blob/generated/_stats/stats.txt

Huber, M. (2024, April). Ldbcollector. Retrieved March 28, 2025, from https://github.com/maxhbr/LDBcollector/blob/main/README.md

International Organization for Standardization. (2023, November). Iso/iec 25010:2023 systems and software engineering — systems and software quality requirements and evaluation (square) — product quality model. Retrieved March 28, 2025, from https://www.iso.org/standard/78176.html

Jaeger, M., & Wang, K. (2015, August). License ref table. Retrieved March 28, 2025, from https://github.com/fossology/fossology/wiki/License-Ref-Table

Jensen, T., & Greinacher, F. (2024, June). Siemens/cyclonedx-property-taxonomy: Cyclonedx property taxonomy for the “siemens” namespace. Retrieved March 28, 2025, from https://github.com/siemens/cyclonedx-property-taxonomy

LicenseDB, S. (n.d.). Scancode licensedb - help. Retrieved March 28, 2025, from https://scancode-licensedb.aboutcode.org/help.html

Lovejoy, J. (2023, February). The spdx license list. Retrieved March 28, 2025, from https://github.com/spdx/license-list-XML/blob/main/DOCS/faq.md

Lovejoy, J. (2024a, February). Explanation of spdx license list fields. Retrieved March 28, 2025, from https://github.com/spdx/license-list-XML/blob/main/DOCS/license-fields.md

Lovejoy, J. (2024b, March). A historical timeline for the spdx license list. Retrieved March 28, 2025, from https://github.com/spdx/license-list-XML/blob/main/DOCS/history.md

Lovejoy, J. (2024c, March). How to request a new license or exception to the spdx license list. Retrieved March 28, 2025, from https://github.com/spdx/license-list-XML/blob/main/DOCS/request-new-license.md

National Institute of Standards and Technology. (2022, July). Executive order 14028, improving the nation’s cybersecurity. Retrieved March 28, 2025, from https://www.nist.gov/itl/executive-order-14028-improving-nations-cybersecurity

Official Journal of the European Union. (2024, November). Regulation (eu) 2024/2847 of the european parliament and of the council of 23 october 2024 on horizontal cybersecurity requirements for products with digital elements and amending regulations (eu) no 168/2013 and (eu) no 2019/1020 and directive (eu) 2020/1828 (cyber resilience act). Re-

trieved March 28, 2025, from https://eur-lex.europa.eu/legal-content/EN/TXT/HTML/?uri=OJ%3AL_202402847

Ombredanne, P., Mahapatra, A. S., & Yeung, C. (2024, August). Scancode licensedb. Retrieved March 28, 2025, from https://github.com/aboutcode-org/scancode-licensedb/blob/main/README.rst

OSI. (2024a, February). The open source definition. Retrieved March 28, 2025, from https://opensource.org/osd

OSI. (2024b, March). The license review process. Retrieved March 28, 2025, from https://opensource.org/licenses/review-process

OSI. (2024c, July). About the open source initiative. Retrieved March 28, 2025, from https://opensource.org/about

Preston-Werner, T. (2023, January). Semantic versioning 2.0.0. Retrieved March 28, 2025, from https://semver.org/

Safyan, M. (n.d.). Singleton anti-pattern. Retrieved March 28, 2025, from https://www.michaelsafyan.com/tech/design/patterns/singleton

State of California Department of Justice. (2024, March). Retrieved March 28, 2025, from https://oag.ca.gov/privacy/ccpa

Stewart, K. (2015, November). Retrieved March 28, 2025, from https://events.static.linuxfound.org/sites/events/files/fossology-overview-20151109.1.pdf

Suriyawongkul, A. (2024, August). Spdx license list matching guidelines and templates (informative). Retrieved March 28, 2025, from https://github.com/spdx/license-list-XML/blob/main/DOCS/license-matching-guidelines-and-templates.md

The United States Government. (2021, May). Executive order on improving the nation’s cybersecurity. Retrieved March 28, 2025, from https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity

tl;drLegal. (n.d.-a). Software licenses explained in plain english. Retrieved March 28, 2025, from https://www.tldrlegal.com/

tl;drLegal. (n.d.-b). Verified license summaries on tl;drlegal. Retrieved March 28, 2025, from https://www.tldrlegal.com/verified

Wang, K., Mishra, G., & Jaeger, M. (2018, July). Nomos. Retrieved March 28, 2025, from https://github.com/fossology/fossology/wiki/Nomos

Weber, S., Jaeger, M., & Wang, K. (2018, April). Retrieved March 28, 2025, from https://github.com/fossology/fossology/wiki/Monk/

Winslow, S. (2023, March). License inclusion principles. Retrieved March 28, 2025, from https://github.com/spdx/license-list-XML/blob/main/DOCS/license-inclusion-principles.md