

Improving the Research Software Review Process of License Scanner Output

Lukas Haß

Master's Thesis

July 9, 2026

Software Engineering Group
Department of Computer Science
Kiel University

Advised by
Prof. Dr. Wilhelm Hasselbring
Carina Haupt

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Weiterhin erkläre ich, dass die digitale Fassung dieser Arbeit, die dem Prüfungsamt per E-Mail zugegangen ist, der vorliegenden schriftlichen Fassung entspricht.

Kiel,

Abstract

Software has become an integral part of modern research. In more and more disciplines, researchers develop their own software to conduct their scientific work. This research software rarely stands alone but builds on numerous open-source packages, whose licenses impose legally binding obligations whenever the software is distributed. Since researchers frequently want to share their software alongside their results, they must comply with the licenses of all included packages. As domain experts rather than licensing specialists, they face a demanding task, and the Research Software Engineers (RSEs) who support them are a scarce resource in research organizations. License scanners promise relief by automating the detection of licenses in source code, but their raw output still requires extensive manual review, and existing tools offer little support for this step. This thesis addresses that gap with the *License Scanner Review Tool*, a prototype of an interactive review layer on top of the OSS Review Toolkit. Grounded in interviews with practitioners experienced in research software license compliance, the tool enriches each license finding with the context reviewers need to make a decision and feeds those decisions back into the automated analysis pipeline. We evaluated the prototype in a usability study with ten participants. It achieved a mean System Usability Scale score of 69.5, and nine out of ten participants agreed that it would reduce their effort compared to their current approach. These results indicate that a dedicated review tool can reduce the effort of license analysis for research software, easing the load on scarce RSEs.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Goals	2
1.3	Document structure	4
2	Foundations	7
2.1	German Aerospace Center (DLR)	7
2.2	License analysis	7
2.3	SPDX specification	8
2.3.1	License List and expressions	9
2.3.2	Declared, concluded, and detected licenses	9
2.4	Existing Tools	9
2.4.1	OSS Review Toolkit	10
2.4.2	ScanCode	11
2.4.3	FOSSology	12
3	Requirements Analysis	13
3.1	Expert interview design	13
3.2	Thematic analysis of interview transcripts	16
3.3	Results of thematic analysis	17
3.4	System requirements	26
4	Technical Design and Implementation	31
4.1	Tool comparison and selection	31
4.2	Integration with ORT	33
4.3	The License Scanner Review Tool	34
4.3.1	User interface structure	36
4.3.2	Package metadata and dependency graph	36
4.3.3	Concluding a package's license	38
4.3.4	Reviewing license findings	39
4.3.5	Comparing altered license texts	42
4.3.6	User guidance through tooltips	42
4.3.7	The server component	43
4.3.8	Technology and usability	44

Contents

5	Evaluation	47
5.1	Evaluation Setup	47
5.2	Procedure	48
5.3	Data Analysis	50
5.4	Results	52
5.4.1	Participants	52
5.4.2	Usability	52
5.4.3	Process Fit	58
6	Discussion	59
6.1	Design	59
6.2	Implementation	60
6.3	Evaluation	61
6.4	Threats to Validity	62
7	Conclusion	65
7.1	Revisiting the Goals	65
7.2	Future Work	66
	Acronyms	69
A	Usage of LLM-gen-AI	71
	Bibliography	73

Introduction

Software has become an integral part of modern research. In an increasing number of scientific disciplines, researchers develop their own software to conduct, automate, and reproduce their work. Such research software is rarely written from scratch. Instead, it typically depends on numerous third-party packages, the vast majority of which are distributed under open-source licenses. While these licenses grant broad rights to use, modify, and redistribute the software, they also impose obligations on users and, in particular, on anyone who distributes the software further. Consequently, when researchers wish to publish or otherwise distribute their software, they must ensure compliance with all applicable licenses, including those of transitive dependencies they may not even be directly aware of. This is a demanding task that requires expertise in both software engineering and the legal aspects of licensing. While license compliance is already challenging for professional software developers, they tend to have at least some familiarity with these issues, and the companies they work for frequently employ legal experts dedicated to such questions. The situation differs markedly in academia: for a software company, the software is the actual product, whereas in research it is usually a byproduct of the underlying scientific work. Research institutions and universities therefore rarely provide dedicated legal support, leaving researchers, who are domain experts rather than licensing specialists, to navigate these obligations largely on their own. Against this background, this thesis develops a tool that integrates into existing open-source license analysis tools to support developers, and researchers in particular, in handling license compliance tasks more efficiently.

1.1 Motivation

The dependence on third-party packages outlined above is not incidental but a deliberate practice. Reusing existing software allows researchers to build on established, well-tested functionality and to concentrate on their actual scientific contribution rather than reimplementing what is already available. This is also where open-source licenses come into play. Their popularity stems from the fact that they explicitly grant the rights needed for such reuse while keeping the accompanying obligations comparatively clear and manageable. However, common licenses require these obligations to be fulfilled not at the point of reuse, but whenever the software is distributed further. Researchers distribute their software,

1. Introduction

for a variety of reasons including financing their research by selling licenses to industry partners, sharing the software with project collaborators at other institutions [Czerniak et al. 2025], and publishing it alongside research papers to support the reproducibility and reusability of their results [Hasselbring et al. 2020]. In doing so, researchers typically act as both consumers and distributors of software. They incorporate third-party packages into their own code and, in turn, pass the resulting software on to others. It is precisely this dual role that turns the license obligations introduced above into a concrete compliance burden.

The reason this burden cannot simply be ignored is that licenses are legally binding documents defining the terms under which software may be used, modified, and distributed. When distributing software that incorporates other packages, compliance with the licenses of all included packages is therefore not optional. Non-compliance can expose the researcher or their institution to legal risks such as lawsuits or financial penalties [Czerniak et al. 2025]. Achieving compliance, however, is far from trivial. As a first step, all included packages and their licenses must be identified. This concerns both direct dependencies, which the researcher includes explicitly, and transitive dependencies, which are pulled in by those direct dependencies in turn. Identifying the corresponding licenses can range from simply reading the license declared for each package to analyzing the actual license texts embedded in its source files. The required effort grows with the number of dependencies and the desired thoroughness of the analysis. Once all licenses are known, their compatibility must be assessed, which in turn requires an understanding of licensing concepts and their legal implications. This is where the gap becomes apparent. Most researchers possess neither sufficient legal knowledge about licensing nor dedicated training in software engineering, since for them software development is often merely a means to an end [Johanson and Hasselbring 2018]. They therefore turn to Research Software Engineers (RSEs) [Felderer et al. 2025] within their organization for help. RSEs are researchers with specific knowledge in software engineering tasks like dependency management and license compliance [Felderer et al. 2025]. While optimally explicitly hired for this purpose of assisting the development and distribution of research software as suggested by Czerniak et al. [2025], RSEs are often a scarce resource in research institutions. Previous work focused on training and knowledge sharing to improve the situation [Schreiber and Haupt 2017] and suggests providing better technical infrastructure and tools to researchers.

1.2 Goals

The aim of this thesis is to take a first step toward improving the license analysis process in large research organizations such as the German Aerospace Center (Deutsches Zentrum für Luft- und Raumfahrt) (DLR) without putting a disproportionate strain on small teams of specialized RSEs who currently support it. We envision the ideal license analysis process as follows: Researchers should consider license compliance from the beginning of the development process, rather than addressing it only once distribution is imminent. To

1.2. Goals

enable this, preconfigured, easy-to-use tools and accompanying guidelines must be made available to them. License analysis should be integrated into the continuous integration (CI) pipeline of software projects, providing researchers with early feedback on potential license issues and preventing compliance problems before they arise. By reviewing and curating license analysis results continuously, the burden of a large, last-minute compliance review before distribution is significantly reduced. As a result, the number of cases requiring RSEs assistance should decrease, allowing RSEs to focus on the genuinely complex cases that remain.

While this vision targets all researchers, this thesis adopts a deliberately narrower scope: improving the efficiency of RSEs in their license analysis work by identifying gaps in the existing tooling and addressing them with a prototype. RSEs are currently the primary point of contact for license compliance in research organizations, making them a suitable starting point for better tooling. Moreover, lessons learned from RSE-focused tooling can inform future tools aimed directly at all researchers. The following goals reflect this focus:

- G1 – Identify user needs:* Understand what information RSEs need when reviewing license analysis results, and what currently makes this process inefficient. This provides the empirical basis for the tool design.
- G2 – Design and implement a prototype:* Design and implement a prototype tool that integrates with an existing license scanner and presents its output in a way that supports efficient review and decision-making, directly building on the suggestion to provide better technical infrastructure to researchers [Schreiber and Haupt 2017].
- G3 – Evaluate the prototype:* Assess whether the prototype reduces the manual effort required for license analysis and improves the efficiency of RSEs in their compliance decisions.

Several boundary conditions shape the scope of this thesis. Rather than being developed from scratch, the prototype shall be built on top of an existing license scanner, reusing its mature scanning capabilities so that the work can concentrate on the identified tooling gaps instead of reimplementing established functionality. It shall be released as Free and Open Source Software (FOSS), to allow the DLR and the wider research community to adopt, inspect, and extend it. To keep the tool maintainable for the developers at the DLR, it shall be implemented primarily in Python. The prototype shall run on a desktop computer while remaining operating-system agnostic, so that it can be used regardless of the individual researcher's working environment. Finally, the prototype is developed by a single person within a time frame of six months. The resulting limit on available developer-hours necessarily constrains the number of features that can be realized. Accordingly, a small set of high-quality, well-considered features is preferred over breadth of coverage.

The prototype is developed with usability as the primary quality concern, as the vision of a tool aimed at all researchers makes learnability and operability essential. Beyond usability, the prototype should not significantly increase the overall license analysis time, and should provide accurate information to prevent users from making decisions that

1. Introduction

could create legal liabilities. Since this is a research prototype intended to run in controlled environments, reliability, security, and compatibility with multiple tools are explicitly not primary concerns. Maintainability is considered to the extent necessary to enable continued development beyond this thesis.

1.3 Document structure

The remainder of this thesis is structured as follows, with Figure 1.1 giving an overview of the process that mirrors the three core chapters of this thesis:

- ▷ Foundations and related work are presented in Chapter 2. The chapter introduces the DLR as the organizational context of this thesis, important terminology, and existing license analysis tools.
- ▷ The requirements analysis is described in Chapter 3. It covers the design of the expert interviews, the thematic analysis of the interview transcripts and its results, and the system requirements derived from them.
- ▷ The technical design and implementation of the prototype is detailed in Chapter 4. The chapter compares the existing tools and selects one to build upon, describes the integration with the OSS Review Toolkit (ORT), and presents the *License Scanner Review Tool* itself.
- ▷ The evaluation of the tool including the results can be found in Chapter 5. It describes the design of the user study and reports the results on usability and the fit into existing license analysis workflows.
- ▷ A discussion of the approach and results is provided in Chapter 6, covering the design, implementation, and evaluation, as well as the threats to validity.
- ▷ The conclusion is available in Chapter 7. It revisits the goals of this thesis and outlines future work.

1.3. Document structure

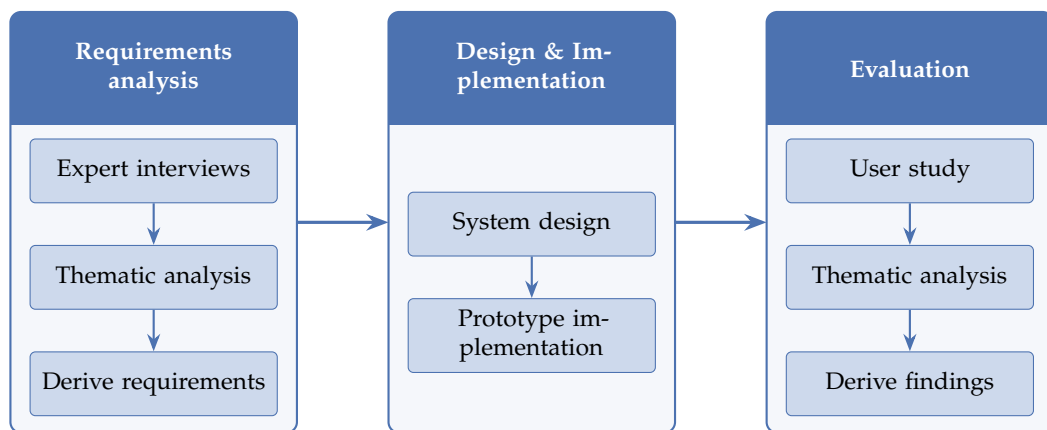


Figure 1.1. Overview of the process followed in the core chapters of this thesis.

Foundations

This chapter provides information about the context in which this thesis is conducted, introduces important terminology, and gives an overview of existing tools for license analysis.

2.1 German Aerospace Center (DLR)

The thesis is conducted in cooperation with the Institute of Software Technology at the DLR. The DLR is a governmental research organization with 11,786 employees [DLR 2024] of whom, based on a survey from 2014, about 3,000 develop software [Schreiber et al. 2014]. The current numbers are expected to be much higher. The team that is responsible for assisting researchers with software license compliance tasks consists of two RSEs and one lawyer, none of whom work on license compliance full-time. Rather, they take on these tasks alongside a range of other responsibilities, which limits their available capacity for individual compliance requests. Previous to this thesis proposal, the DLR introduced an informational document about open-source software [Brochhagen et al. 2022], started training courses about open-source licensing and provided collaboration tools and knowledge sharing workshops for researchers, where the topic of licensing is addressed [Schreiber and Haupt 2017]. Thus, the needs and requirements of the DLR directly influence the goals and approach of the thesis.

2.2 License analysis

To facilitate the re-use of software, open-source licenses define the legal terms under which software can be used, modified, and distributed. Most open-source licenses come with certain obligations for the user of the software, such as providing attribution to the original author or distributing the source code of modified versions. To avoid legal risks when distributing software that includes open-source components, it is important to analyze the licenses of all included components to ensure compliance with their terms. Li et al. [2025] divide the process of open-source license analysis into three main steps: license identification, license risk assessment, and license risk mitigation. This thesis focuses on the first step, license identification.

2. Foundations

License identification is the process of identifying the license under which a software package is distributed. Licenses can be identified in two ways: by their name or by their terms. Name-based identification determines the name of the license a software package is distributed under. The easiest way to do this is to look for a license in the package metadata or in a LICENSE file. If no license is found there, license scanners can analyze the source code files of the package to identify license texts based on a database of known license texts. Depending on where the license information originates, the identified licenses can be categorized into declared, detected, and concluded licenses. We introduce these terms in Section 2.3.

Term-based identification is a more specific approach: instead of matching a license as a whole by its name, it identifies the individual terms contained in a license text. The identified terms can then be used to determine the obligations and permissions of the license. Because it focuses on the actual content of the license text rather than its name, this approach enables a more fine-grained analysis of compatibility. This is particularly interesting when dealing with custom licenses or modified versions of existing licenses. These cases would fail in name-based identification, but term-based identification might still be able to provide useful information for compatibility analysis.

License risk assessment analyzes the compatibility of the identified licenses to assess the risk of license non-compliance when distributing the software. There are various approaches to analyze license compatibility, ranging from simple lookup tables to complex rule-based systems.

License risk mitigation employs strategies to reduce the risk of non-compliance once license incompatibilities or other risks are identified. Possible strategies include replacing incompatible components with compatible ones, obtaining additional permissions from the original authors, or re-licensing the software under a different license.

2.3 SPDX specification

The System Package Data Exchange (SPDX) is an open standard for communicating software Bill of Materials (BOM) information, including provenance, licenses, and security references [SPDX Project 2024]. It was originally developed under the Linux Foundation to provide a common, machine-readable format for exchanging license and copyright data between tools and organizations, and has since been published as an international standard [International Organization for Standardization 2021]. Because it defines a shared vocabulary for licenses, SPDX is widely adopted by license analysis tools. The tools introduced in Section 2.4 can all express their results using SPDX identifiers. For this thesis, three aspects of the specification are particularly relevant: the SPDX License List, license expressions, and the distinction between declared, concluded, and detected licenses.

2.3.1 License List and expressions

The SPDX License List is a curated set of commonly used open-source licenses, each assigned a unique, human-readable short identifier such as MIT, Apache-2.0, or GPL-2.0-only [SPDX Project 2026]. These identifiers avoid the ambiguity of referring to licenses by their full names and enable an unambiguous, tool-independent way to reference a license.

To describe cases where more than one license applies, SPDX defines a syntax for license expressions [SPDX Project 2024]. A license expression combines license identifiers using the operators AND, OR, and WITH. The AND operator indicates that the terms of all connected licenses apply simultaneously, while the OR operator indicates that the recipient may choose between the connected licenses. The WITH operator attaches a license exception to a license identifier, for example GPL-2.0-only WITH Classpath-exception-2.0. Licenses that are not part of the SPDX License List can still be referenced using the LicenseRef- prefix, which allows tools to represent custom or otherwise unrecognized licenses.

In addition to license identifiers, SPDX defines two special values. The value NONE indicates that no license was found, whereas NOASSERTION indicates that no assertion about the license could be made, for example because the license could not be determined, or the analysis was inconclusive. This distinction is important because it separates the absence of a license from the absence of knowledge about the license.

2.3.2 Declared, concluded, and detected licenses

SPDX distinguishes between the declared and the concluded license of a package. The declared license is the license as stated by the package authors, for example in the package metadata or a LICENSE file. The concluded license is the license determined by an analyst or tool after reviewing the available evidence, which may differ from the declared license when the declaration is missing, incomplete, or contradicted by the actual license texts found in the source code. SPDX does not define a dedicated term for the individual license findings that a scanner reports for individual files. The ORT refers to these as detected licenses [ORT Project 2026b], and they correspond to the per-file license information recorded in SPDX. The detected licenses serve as the evidence from which a concluded license is derived.

2.4 Existing Tools

This section summarizes the most relevant existing tools and their capabilities, focusing on the license identification step of the license analysis process. Rather than reimplementing this established functionality, we build our prototype on top of one of these tools. We restrict the overview to open-source tools and leave out commercial ones, since the foundation of the prototype must be freely available and extensible.

2. Foundations

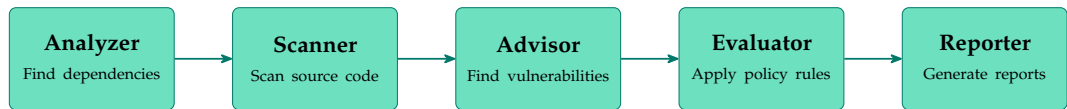


Figure 2.1. Overview of the ORT pipeline, adapted from [ORT Project 2026c]. A project is passed through the pipeline, whose analyzer resolves the dependencies, whose scanner identifies the licenses in the source code, whose advisor retrieves known security vulnerabilities, whose evaluator applies policy rules, and whose reporter generates the output. All components except the advisor are relevant for license analysis.

2.4.1 OSS Review Toolkit

The ORT offers a pipeline for analyzing software packages and their dependencies [ORT Project 2026c]. It includes components for dependency identification, downloading source code, license identification, finding vulnerabilities, evaluation of license compliance, and reporting. Figure 2.1 shows an overview of the ORT pipeline. The only required component is the analyzer since it provides the dependency identification. All other components are optional and can be used as needed. The analyzer also includes the collection of package metadata and thus the collection of declared licenses. The license identification component is a unified interface for various license scanners, including ScanCode and Licensee [ORT Project 2026d]. The ORT defines a common output format for all of its components, including the license scanners. An example output of the scanner component is shown in Listing 2.1. This example shows a part of the detected licenses for a package, specifically three findings of the MIT license in different files with different confidence scores. It also provides a file format for defining curations, which are manual corrections to the output of the license scanners. Listing 2.2 shows an ORT curation that sets the concluded license for all GPL-2.0-only findings in C++ source files to Apache-2.0. Curations can have a reason and a comment attached to them to document why the curation was made.

Listing 2.1. Example ORT scanner output for a package (abbreviated).

```
1 - provenance:
2   vcs_info:
3     url: "https://github.com/vuejs/core.git"
4   summary:
5     start_time: "2025-12-01T13:45:41.000746946Z"
6     end_time: "2025-12-01T13:47:12.000218007Z"
7     licenses:
8       - license: "MIT"
9         location:
10          path: "README.md"
11          start_line: 9
12          end_line: 9
13          score: 100.0
14       - license: "MIT"
15         location:
```

2.4. Existing Tools

```
16     path: "README.md"
17     start_line: 52
18     end_line: 54
19     score: 38.0
20   - license: "MIT"
21     location:
22       path: "packages/compiler-core/LICENSE"
23       start_line: 1
24       end_line: 1
25     score: 100.0
```

Listing 2.2. Example ORT curation.

```
1 curations:
2   license_findings:
3     - path: "src/**/*.cpp"
4       start_lines: "3"
5       line_count: 11
6       detected_license: "GPL-2.0-only"
7       reason: "CODE"
8       comment: "The scanner matches a variable named 'gpl'."
9       concluded_license: "Apache-2.0"
```

2.4.2 ScanCode

The ScanCode Toolkit [AboutCode 2026] provides a command-line tool for scanning codebases to identify open-source licenses, copyrights, and other relevant information. It uses a rule-based approach to identify licenses based on a large database of license texts and patterns. While ScanCode can be used as a standalone tool, it is also integrated into the ORT as one of the supported license scanners.

ScanCode Workbench is a desktop app that allows viewing results from ScanCode Toolkit. Crucially the provided interface is read-only and does not allow to make any changes to the scan results. Nonetheless, the way this tool presents the scan results is a relevant reference for the design of our prototype, because it shows much of the same information that our tool needs to present to the user. It provides four different views on the scan results. A Table View shows scan results by file and can be filtered by directory or single files using a file tree. The License Explorer View lets the user show license findings filtered by detected license expression. The Package Explorer View shows scan results ordered by package type and package name. This view also includes a hierarchy showing dependencies of packages. The Chart Summary View shows statistics like counts of detected licenses, counts of detected copyright authors and file counts grouped by programming language.

2. Foundations

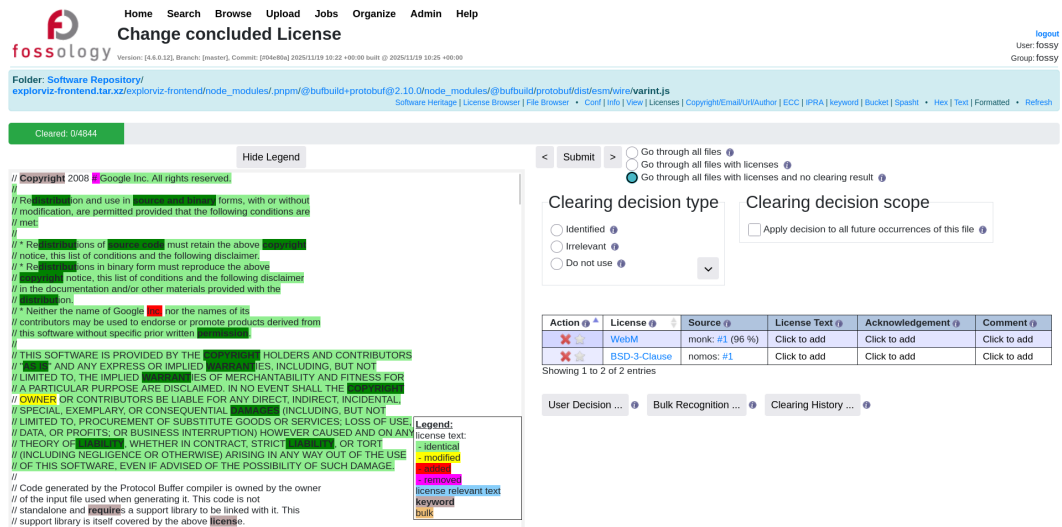


Figure 2.2. User interface of the clearing workflow in FOSSology.

2.4.3 FOSSology

FOSSology is an open-source license compliance software system [Gobeille 2008]. It is mainly used through a web-interface and provides multiple license identification implementations as well as an integration for running the ScanCode Toolkit from within FOSSology. Notably FOSSology includes a clearing workflow that is similar to the tool we designed in this thesis. Figure 2.2 shows this clearing workflow. In FOSSology every license has to be cleared, even the simple cases with no conflicting or low confidence scanner results. In the particular case shown in Figure 2.2 FOSSology detected two differing licenses in one file, because it uses more than one scanning algorithm. The left panel shows a highlighted comparison of the actual file content and the original license text of the identified license. The table on the right side lets the user choose between the detected licenses and submit a clearing decision.

Requirements Analysis

This chapter derives the requirements for the system from the needs of practitioners. To this end, we conducted semi-structured interviews with four license compliance experts at the DLR and analyzed the transcripts using thematic analysis. We first describe the design and execution of the interviews, then the thematic analysis method, its application to the transcripts, and the resulting themes. From these themes we finally derive 20 user stories that state the requirements the system should fulfill.

3.1 Expert interview design

To ensure that the system is useful for practitioners, we gathered requirements via expert interviews. The interviews explored how reviews are currently performed by hand, what contextual information informs these decisions, and how a system could best present such information to the reviewer. We conducted semi-structured interviews with four practitioners experienced with license compliance tasks. We recruited the interviewees from within the DLR, with three of four being part of the open-source team. Three of the four interviewees hold a degree in computer science, while the fourth holds a degree in law. By role, two of them work as software engineers and one in a management position, whereas the interviewee with the law degree works as a contract lawyer. Their self-reported experience with license analysis and its sub-tasks ranges from average to above average, whereas their experience with license analysis software is lower and varies more widely, from very inexperienced to experienced as shown in Figure 3.1. Across the panel, the interviewees report between two and 14 years of experience with license analysis. The chosen interview format ensures consistency across interviews, by following a predefined set of questions, the interview guidelines [Haß 2026], while still allowing the flexibility to explore noteworthy points that arise during the conversation. We conducted the interviews in German, the first language of both the interviewer and the interviewees. As the license analysis domain is complex enough on its own, using the participants' first language avoids an additional language barrier and allows for a better mutual understanding.

The guidelines cover five topic areas:

Interviewee profile: The opening questions serve to get to know the participant, their role in the DLR and their experience with license analysis. Combined with a questionnaire [Haß 2026] covering the background of the participants and a self-reported measure of

3. Requirements Analysis

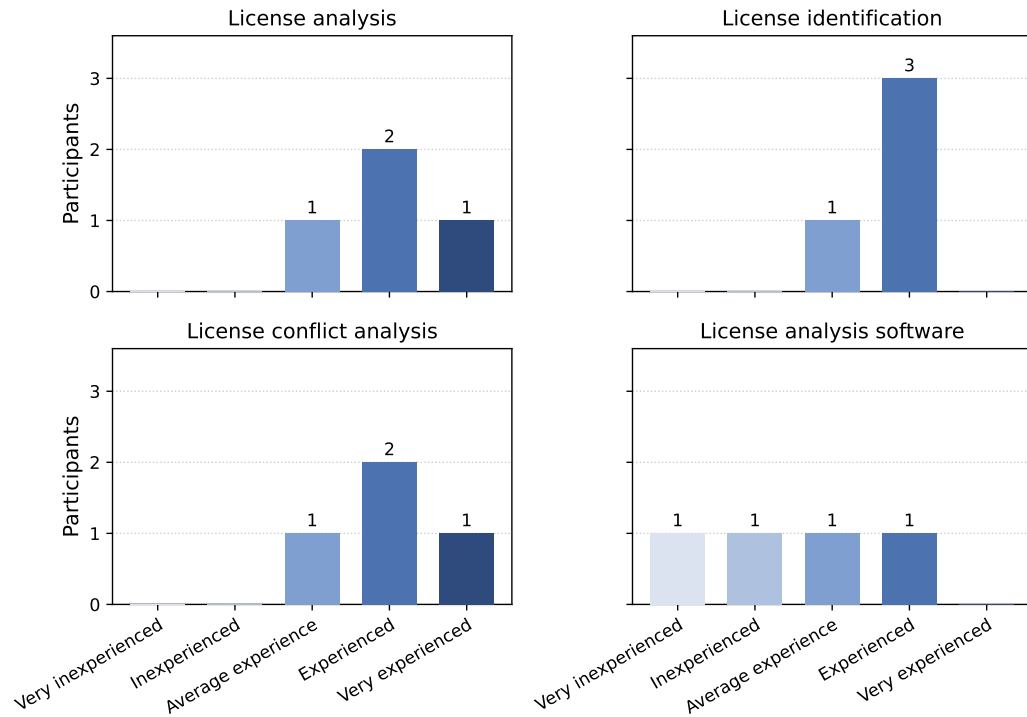


Figure 3.1. Self-reported experience distribution of interviewed experts with license analysis, license identification, license conflict analysis and license analysis software. Interviewees report an average to above average experience with license analysis and its sub-tasks. Experience with license analysis software is the lowest and varies widely from very inexperienced to experienced.

domain expertise, these questions allow the subsequent responses to be interpreted in light of each participant’s background and level of expertise.

License analysis goals: This section addresses the rationale behind performing license analysis before software is distributed, as well as its objectives. The main interest lies in the motivation and perceived stakes behind the task.

License analysis process: One of the main aspects of the interview is to understand how license compliance is currently ensured, and what that process looks like in detail. The questions aim to elicit the roles of people involved, the information required, and its sources. Here, we also ask the participants to assess the current process in order to understand what works well, what could be improved, and what such an improvement would look like in their view.

Tooling interaction and presentation: This section addresses the tools currently used for license

3.1. Expert interview design

analysis, their strengths, and shortcomings, and which parts of the process participants would like to automate. It further explores how a future tool should present contextual information to support review decisions. The responses inform both functional and non-functional requirements for the system.

User groups: The final section broadens the perspective beyond the individual interviewee to consider the wider range of potential users. The questions address who requires tooling support for license analysis, how those users can be grouped, and what distinct requirements each group might have.

To ensure the quality of the interview design, an expert in qualitative research methods reviewed the interview guidelines. The estimated interview length based on the guidelines is 60 minutes.

To allow the interviewer to focus on asking questions and following up on noteworthy points rather than taking notes, we recorded the audio of the interviews with the interviewees' consent. An ethics approval was obtained before conducting the interviews, to ensure that the study complied with institutional guidelines on research involving human participants, including informed consent, the right to withdraw at any time, and the secure and confidential handling of personal data according to the General Data Protection Regulation (GDPR). The resulting audio recordings sum up to a total of three hours and 35 minutes.

We transcribed the recordings using a Whisper speech-to-text model [Radford et al. 2022], deployed on the DLR intranet, ensuring that no interview content was transmitted outside the organization. To avoid timeouts of the transcription interface, recordings were split into segments of at most 30 minutes before processing. The transcription ran in approximately real time and, as it could proceed in the background, demanded no active involvement from the interviewer. The automatic transcriptions required manual review and correction before use. This review process involved the following steps:

- ▷ Separating speakers, as the model output contains no speaker identification and presents all spoken content as a single paragraph.
- ▷ Inserting timestamps every one to three minutes to support the correction process and to allow transcription errors to be located within the recording.
- ▷ Correcting domain-specific terminology that the model transcribed incorrectly.
- ▷ Fixing punctuation, spelling, and other transcription errors.
- ▷ Manually transcribing short passages where the model failed to produce a transcription and instead inserted "... " or a repeating word into the output; this occurred six times across all recordings.

In one recording, the interviewer's voice was significantly louder than the interviewee's. While this did not noticeably affect the transcription quality, it made the recording difficult

3. Requirements Analysis

to listen to during review. We addressed this by normalizing the volume of the affected recording using *FFmpeg* before conducting the review.

The manual review took approximately twice the duration of the recordings, bringing the total transcription effort to roughly eight hours. While still substantial, this represents a considerable saving compared to a fully manual transcription, which we estimated would have taken approximately four times the recording duration, or around 16 hours in total.

The final transcriptions are saved in a simple text based format. Each line in the file is either an utterance of the interviewer marked with the prefix Q:, an utterance of the interviewee marked with the prefix A: or a timestamp marked with the prefix T:. Listing 3.1 shows an example of this file format.

3.2 Thematic analysis of interview transcripts

We analyze the interview transcripts using thematic analysis, a qualitative research method for identifying and interpreting patterns of meaning across a dataset [Braun and Clarke 2006]. We adapt thematic analysis for our context following the approach of Jung et al. [2022], who apply thematic analysis to expert interviews with domain practitioners in a scientific setting analogous to ours. A central concept in thematic analysis is the *theme*: a pattern that captures something meaningful about the data in relation to the research question. Themes can be *semantic*, reflecting explicit and surface-level meanings in the data, or *latent*, capturing underlying assumptions and ideas that are not directly included in the data. In our analysis, we use a combination of inductive and deductive coding. Inductive coding allows deriving codes from the data without trying to fit them into a pre-existing coding frame. The aim of inductive coding is to let the themes emerge from the data and possibly discover unexpected themes. It is a bottom-up approach, that is useful to avoid influencing the analysis with existing bias or for analyzing data in a new domain. Deductive coding, on the other hand, allows us to use our existing research question to shape the themes and codes. This method is a top-down approach, that is useful to focus the analysis on specific aspects of the data that are relevant for the research question.

Braun and Clarke [2006] describe thematic analysis as a six-phase process:

Phase 1 – Familiarization with the data: The researcher reads and re-reads the dataset to develop a thorough understanding of its content before any formal coding begins.

Phase 2 – Generating initial codes: The researcher systematically codes features of the data that appear relevant to the research questions, producing an initial set of codes across the full dataset.

Phase 3 – Searching for themes: Related codes are collated into candidate themes, identifying broader patterns of meaning across the dataset.

3.3. Results of thematic analysis

Phase 4 – Reviewing themes: The candidate themes are refined by checking them against the coded data and the full dataset, merging, splitting, or discarding themes as necessary to ensure they accurately represent the data.

Phase 5 – Defining and naming themes: The scope and focus of each theme is defined, and names are assigned that clearly convey their meaning in relation to the research questions.

Phase 6 – Writing the report: The themes and their implications are reported in relation to the research questions.

3.3 Results of thematic analysis

We applied the thematic analysis process described in Section 3.2 to the transcripts of the four expert interviews, with the goal of identifying themes that inform the requirements for the design of the system. During the manual review and correction of the transcripts generated using speech to text software, we were already able to familiarize ourselves with the data, effectively completing Phase 1. A subsequent dedicated reading of the transcripts allowed us to generate initial codes, as described in Phase 2. This resulted in a total of 327 initial codes, with interview I1 contributing 141, interview I2 contributing 66, interview I3 contributing 78 and interview I4 contributing 42 codes. While the number of codes is related to the length of the interviews shown in Table 3.1, this does not fully explain why interview I1 has nearly twice the amount of codes after the initial coding while only being four minutes longer than interview I3. The order in which we conducted the coding might offer a better explanation. We coded the interviews in ascending order interview I1 being the first and interview I4 the last. As interview I1 was coded first, all of its information was new to us, so nearly every relevant statement produced a code. In the interviews coded afterwards information is repeated and thus not always written down as a code. Consequently, the set of codes might be biased towards new, unheard information. Another reason for these numbers could be the chosen mixture of inductive and deductive coding. Our existing research question focused the coding on the usage of tools to improve the license analysis and the answers in interview I1 match this focus more than the other interviews. A pure inductive coding approach likely would have resulted in a more balanced code distribution. To record the codes adjacent to the interviewees answers, we extend the file format used for the transcriptions by adding the codes directly after the answers between angle brackets, separated by commas. The resulting interview transcription files look as shown in Listing 3.1.

Listing 3.1. Excerpt from one of the interview transcriptions, showing the file format used to identify interviewer (Q) and interviewee (A) and showing timestamps (T) and codes extracted from the answers.

```
1 Q: What information and data do you need to do a license analysis?
```

3. Requirements Analysis

```
2 A: Information on what kind of research institute it is. Access to the source code. <  
   source code, type of organization>  
3 T: 13:37  
4 Q: Where do you get this information?  
5 A: From the requesting person. <requesting person>
```

We then grouped related codes into categories and identified overarching themes, corresponding to Phases 3 through 5. Where a code was relevant to multiple categories or themes, it was assigned to all of them. Because of this multiple assignment, the code counts reported per theme below add up to slightly more than the total number of unique codes. This process structured the codes into six themes and 25 categories. We assigned all codes to at least one theme, but not necessarily any category. Whether to introduce a category did not depend on a minimum number of codes, but on whether a code captured a distinction within its theme that the theme alone did not already convey. The result of grouping codes into categories and themes are multiple tree structures, one for each theme. In each tree the root node is the theme, intermediate nodes are categories and leaf nodes are codes. Figure 3.2 shows an excerpt of this structure for the *Context* theme, together with exemplary quotes from which the codes originate. To create, work with and write down this structure we use a text based file format shown in Listing 3.2. Each line is either a theme, category, or code. Themes are not indented, categories are indented followed by a line with more indentation and codes are lines with indentation, but followed by a line with the same or less indentation.

Listing 3.2. Excerpt from the thematic analysis file, used to group codes into categories and themes.

```
1 - Users (who is involved in license analysis and is thus a potential user)  
2   - Groups  
3     - scientists (2x)  
4     - ...  
5   - Requirements  
6     - scientists do not need the legal background info  
7     - ...  
8   - ...  
9 - Current process (what does the current process include and how is it viewed by the  
   interviewee)  
10  - trust licenses provided by scientists  
11  - Tools  
12    - structured questionnaire to determine if licensing is possible  
13    - ...  
14  - ...  
15 - ...
```

Notably in Phase 4 we restructured the themes from eight to six, merging two small themes into existing categories of other themes. All three codes of the *Obligations* theme were moved entirely into the *License* category of the *Context* theme. For our purposes

3.3. Results of thematic analysis

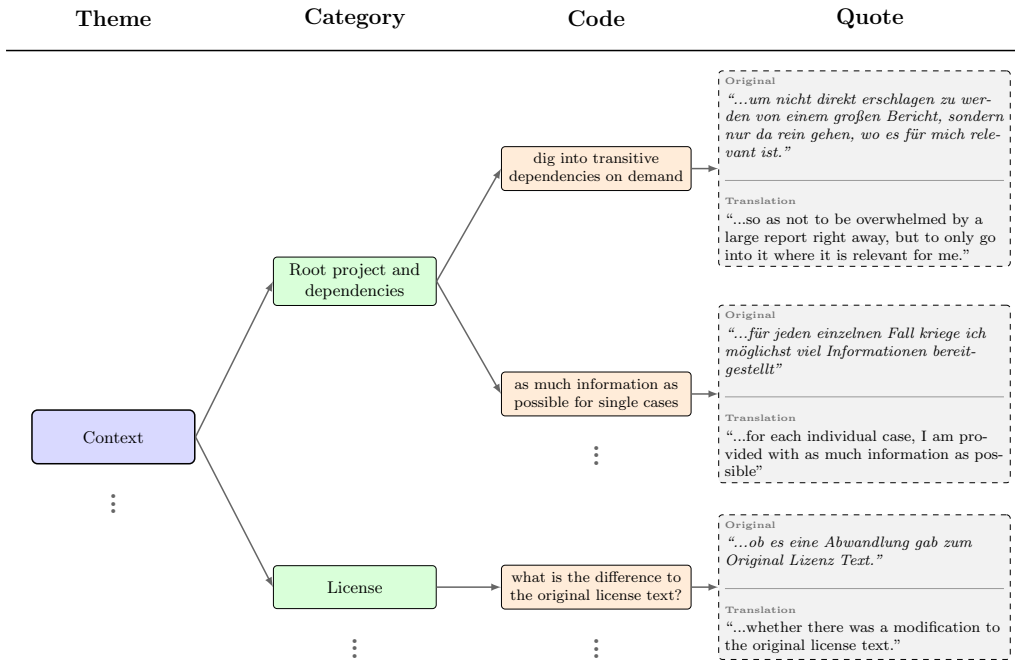


Figure 3.2. Excerpt of the thematic analysis results, showing the *Context* theme with exemplary, associated categories, codes, and quotes from the interviews with English translations. It illustrates how the thematic analysis proceeds from individual quotes via codes and categories to overarching themes.

obligations are most relevant in a way that we want to make the user aware of those obligations by giving context. The vaguely defined *Results* theme which included only two codes was dissolved and the codes moved into the *Motivation* theme. The fact that it was much harder to come up with a clear definition of the theme made the theme a primary candidate for removal. Another major change during refinement was renaming the *Process* theme to *Current process*, promoting the category *Future* to become theme *Future process* and integrating former theme *Automation* as one of its categories. We also refined the codes themselves, by consolidating codes with equal meaning, but differing notation into one. The coding process itself and especially this consolidation produced a number of duplicate codes, stemming from multiple interviewees independently raising the same points. To account for this in the theme tree, where the number of occurrences of a code would otherwise be lost, we add a count to each code with more than one occurrence in the interviews. The notation for this count is a suffixed (2x) for a code with two occurrences.

After identifying the final themes, we conducted a number of checks to ensure the consistency of the resulting tree structure of themes, categories, and codes. Working with

3. Requirements Analysis

the interview transcripts, codes, and themes involves a lot of moving around codes and copy-pasting between the used text files to find the final themes, so inconsistencies due to human error are to be expected. The checks ensure that every code identified in the interview transcription files is included in the final theme tree. We also make sure that no codes made their way into the theme tree, that are not in the interview transcription files. Another check ensures that the quantifier added to codes in the tree structure matches the number of occurrences in the transcription files.

Table 3.1. The number of codes per interview across coding stages in relation to the length of the interview in the order in which the coding has been done. Interviews coded first result in more codes. Codes in the deduplicated column are counted only for the first interview they originate from resulting in lower code counts for interviews coded later.

	Length in minutes	Initial coding	Final coding deduplicated
Interview I1	63	141	138
Interview I2	42	66	57
Interview I3	59	78	67
Interview I4	51	42	36
Total	215	327	298

The complete dataset of themes, categories, and codes is available at [Haß 2026]. We now describe the data, by defining the themes and categories and discussing an excerpt of the codes therein. The final six themes are *Motivation*, *Current process*, *Future process*, *Users*, *Tooling*, and *Context*:

Motivation The *Motivation* theme provides insights into what motivates people or organizations to do license analysis. It splits into five categories and includes a total of 41 codes. The motivation that interviewees attached to license analysis is anchored in legal compliance and the minimization of risk. Preventing legal repercussions was the most frequently cited concern. To this end the license analysis is about ensuring that the organization actually holds the rights to the software it ships. This includes the aspect of ensuring license compatibility when combining packages with different licenses, as well as collecting the agreements of all external personnel to a particular license. Alongside these defensive goals, several participants articulated more constructive motivations: promoting the open-source community, acting fairly toward upstream authors, and following good research practices. In their role as consultants the interviewees do not want to control the compliance with licenses, but enable researchers to comply with their chosen licenses by giving advice. The distribution and reuse of software is a goal that the license analysis works towards by ensuring the legality.

The stakes attached to these goals are substantial. Interviewees described potential consequences of breaching license conditions ranging from lawsuits and six-figure penalties to financial reimbursement, project delays, and the considerable unpaid effort required

3.3. Results of thematic analysis

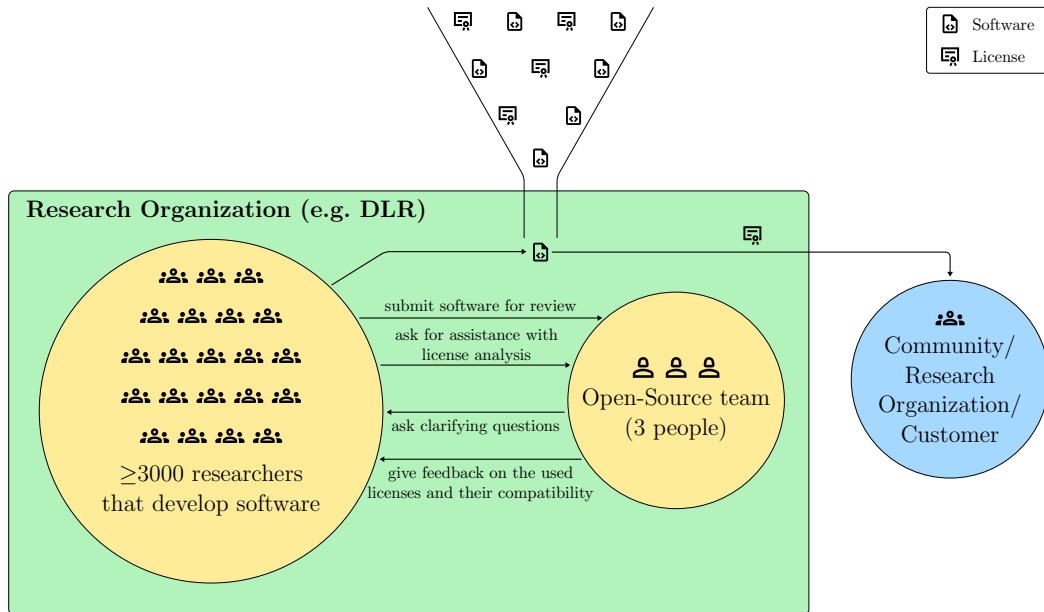


Figure 3.3. Visualization of the license compliance process described by the interviewees, shown with the number of researchers and license compliance experts reported at the DLR. Over 3,000 researchers in the research organization develop software that includes licensed third-party software and is in turn distributed under a license to the community, other research organizations, or customers. Before distribution, researchers submit their software for review and ask the three-person open-source team for assistance with license analysis. The team responds with clarifying questions and feedback on the used licenses and their compatibility. Because the researchers greatly outnumber the license compliance experts and lack licensing knowledge themselves, this communication becomes a time-consuming bottleneck.

to remove or replace for example GPL-licensed code in order to restore compliance. Apart from these legal consequences bad publicity and a damaged reputation were also mentioned. Crucially, participants say that such breaches are seldom willful. They typically originate from unawareness or misinterpretation of license terms. In this case license analysis acts as a safeguard against honest mistakes rather than a defense against deliberate misconduct. This also explains why some breaches are resolved cooperatively through the open-source community hinting toward a correction rather than through litigation.

Beyond the goals and consequences, several specific risk vectors sustain the need for license analysis. Among these are patent trolls and parties who deliberately set traps through cease-and-desist actions. The direct dependencies of a software are considered the most significant source of risks. Given these risks, the confidence the

3. Requirements Analysis

interviewees desire before publicly releasing software still varies and depends on the context. While complete confidence is sometimes desirable for some, the required confidence is not quantifiable for others and varies, for example, with the funding of a project. The aspiration is therefore to be as confident as possible. As a lower bound, the interviewees consider a best-effort analysis or a confidence of 90–95% acceptable, while acknowledging that a residual risk can never be fully eliminated.

Current process The *Current process* theme describes how compliance with license terms is ensured today and how this practice is viewed. It splits into six categories and includes a total of 92 codes. A defining characteristic of the current practice is that no central and no strict process exists. Instead, many publishing processes coexist depending on the financing and the goals of a project, and the consulting is largely voluntary. The work is predominantly manual and time-intensive, so that a full analysis is not feasible for every project, even though virtually every project carries license issues. Analysis is currently performed on finished projects rather than continuously during development.

The roles in the process are not well-defined and can vary from situation to situation, but generally the process usually starts with someone that is interested in distributing a software. This might be someone that is also responsible for the software, for example a department leader, software developer or scientist. In some cases they search for help from the open-source team, which is the most mentioned role in our data caused by the interviewees being recruited from that team. For commercial distribution the technology transfer and additional lawyers might be involved.

Even without the support of the open-source team there are tools available like the open source brochure [Brochhagen et al. 2022] and a software questionnaire. This questionnaire provides a checklist that helps with assessing whether a software is ready to be published. Employees often do not have the knowledge to answer the questionnaire themselves, so the open-source team also consults its use.

The analysis itself follows no fixed procedure, but a number of recurring steps can be identified. It begins with determining the intended recipient and the distribution format, since the licensing obligations depend on the composition and the runtime of the software. The dependency list that the analysis builds on is not assembled by the consultant: its creation is delegated to the software developer, who sends a list of the direct dependencies to the open-source team. The analysis then concentrates on these direct dependencies, whereas transitive dependencies are examined only for more critical projects or, in an ideal case, in full. For each dependency the licenses are checked against the provided dependency reference, and it is verified that a license applies to the correct part of a package. Because the licenses declared in package metadata are of imperfect quality, the analyst additionally checks whether a declared license is actually correct. Dependencies that are not distributed, such as test dependencies, are separated out, as they are a frequent cause of false positives. Uncommon or unknown licenses require manual research to interpret their terms. To keep the workload manageable,

3.3. Results of thematic analysis

often only a sample of the findings is examined rather than the complete set. Questions and results of the license analysis are communicated with those seeking help as unstructured text over email or a ticket system.

A salient characteristic of the process is how much of it rests on trust. The team trusts that the delegated dependency list is accurate and relies on the declared direct dependencies, on the package metadata, and on the licenses provided by the scientists. How much trust is extended depends on the size of the project.

The effort involved varies with the quality of the initial information, the size of the project and the licenses it includes. Obtaining the necessary information from the software developer and analyzing transitive dependencies are both reported as time-consuming, and commercially licensed parts require additional work. This effort competes with the other responsibilities that members of the open-source team carry. Throughout, the process is barely supported by dedicated tooling: standard tooling is described as missing, the setup of existing tools as challenging, and projects without a package manager complicate the analysis further.

Reflecting this variability, the reported duration of a single analysis ranges from a quick answer or about thirty minutes to several weeks, and in non-continuous work up to six months. License analysis is requested on the order of five to ten times per year, and of the one to three monthly consulting requests roughly every third to fourth involves license analysis, while the real demand is assumed to be higher still. This inequality in demand for help and available consultants is also shown in Figure 3.3.

Despite its informality, the current practice is valued for the legally sound and individualized feedback it provides. Its main drawbacks are the high workload, the need to look up details on uncommon licenses, the false positives caused by non-distributed dependencies and by scanning tools, the imperfect metadata quality, the absence of a report giving an overview of the results, and the fact that consultants often have no access to the source code.

Users The *Users* theme describes the potential user groups of our system and their respective requirements. It splits into two categories and includes a total of 22 codes. The three broad user groups that were mentioned are the consultant from the open-source team, software authors meaning developers or scientists and stakeholders. The consultants have both technical and legal knowledge, while the software authors are expected to only have technical knowledge and stakeholders more legal than technical knowledge. Both consultants and software developers favor usability over confidence. While consultants are interested in the legal background that leads to licensing decisions, developers and scientist need less in-depth information in favor of more basic information that is necessary to make a good decision. The stakeholders and management users need a rough overview with results that provide the best confidence possible and are transparently documented.

3. Requirements Analysis

Tooling The *Tooling* theme describes the tools used today and the shortcomings of existing tools. It splits into two categories and includes a total of 14 codes. Tools that are used today include the non-software tools like the open-source brochure and the software questionnaire. The interviewees also mention tools like REUSE and package managers in general, that create a structured information basis for license analysis to build on. Used for the license analysis itself are software tools that list dependencies and the licenses included in their package metadata like package managers themselves or for example `pip-licenses`. License scanners were also mentioned but do not seem to be widely used, yet.

The most mentioned criticism of existing software tools is the unreliability of their outputs and the connected lack of trust in the results. They are also reported as being difficult to use and poorly documented. The functionality to store the results of manual reviews is missing.

Future process The *Future process* theme describes the aspirations for an improved process and the requirements that should be met in the future. It splits into four categories and includes a total of 81 codes. The interviewees would like the process to be more defined, streamlined and centralized. The process should enable people to for example analyze a software project during development. They suggest replacing the aforementioned software questionnaire with a survey tool.

The wish for automation ranges from single tasks like license identification, checking license compatibility and suggesting replacements for incompatible packages, to automating the complete process from start to finish. There is awareness about difficulties when automating complex cases. Licenses are defined in natural language and thus open for interpretation and their processing viewed as hard to automate. Additionally, programmers do not always follow best practices. Combined with a heterogeneous programming language and tooling ecosystem this makes determining dependency lists and finding more information on dependencies difficult to automate. A recurring suggestion that could improve handling these difficulties is to automate the process while keeping a human in the loop. This hints towards a need to make results verifiable.

We asked the interviewees what functionality a tool that would support this future process should have. It should provide more context information that helps analyze a project's licenses than the current ones, while simultaneously limiting the information to a comprehensible amount. In an ideal world separate research would be unnecessary because all context information is already provided by the tool. This could be done by being able to increase the information details on demand. More specifically direct dependencies should be prioritized during the analysis and when needed digging deeper into transitive dependencies should be possible. The tool should integrate into existing software development workflows like a version control and continuous integration systems. Collaborative work should be supported, and any decisions made during license analysis should be saved along with the time and author of that decision.

3.3. Results of thematic analysis

New license findings occurring when a software or dependency is updated should be handled well. Findings that were already reviewed in earlier versions of a package should be handled in a way that reduces duplicate work. Licenses should be shown using a standardized license identifier and information sources should be made available to allow verification. Snippet scanning should be possible using the tool, to prepare the tool for the eventuality of more unattributed, but licensed code snippets being used in the future. Lastly the tool should be able to output a report that can be sent to the researcher that requested support or to stakeholders.

Good usability is an important requirement in the eyes of the interviewees, including a good overview of the analysis and its progress. Besides that, the system shall improve the accuracy of the license analysis, provide legally sound and auditable information, and produce its results deterministically. The system should be performant. What that means in pure numbers varies from accepting 30 minutes of runtime to ten hours of runtime. Running the tool on development machines with limited resources should be possible, allowing to integrate it into early software development, while also allowing to integrate it into Continuous Integration (CI). The system should also be well documented, stable i.e., not suddenly become unavailable and be reasonably secure, especially preventing data of the analyzed software from leaking outside the organization.

Context The *Context* theme describes information that is needed or helps when doing license analysis. It splits into six categories and includes a total of 52 codes. For the overall project important context is the source code, information on the software complexity, a list of authors especially third party authors, the organization the software project belongs to and information on funding. A list of licenses classified into open-source and commercial that are used in the project is deemed helpful. Dependencies should be listed with their respective license and use-case. If a dependency is used for testing that information is valuable to decide if its license is important for the analysis. The directories in which a dependency is used is a good indicator for this. Direct and transitive dependencies should be separated, given that direct dependencies are more critical to the analysis. For dependencies themselves, the popularity derived from the number of GitHub star and uses in other project gives a good impression on their trustworthiness. The interviewees also suggest context for licenses, like a classification into permissive and non-permissive. Showing the full license text is suggested as well as showing the difference of detected license texts to the original license texts. Hints for common license obligations like the inclusion of the license text in the distribution could also be useful.

3. Requirements Analysis

3.4 System requirements

We use user stories [Cohn 2004] to structure the requirements derived from the thematic analysis of the expert interviews. A user story is a short, semi-structured natural-language description of a requirement from the perspective of a stakeholder role. We follow the template “As a *⟨role⟩*, I want *⟨goal⟩*, so that *⟨benefit⟩*” [Cohn 2004]. The role identifies who needs the capability, the goal states the desired functionality, and the benefit makes the underlying motivation explicit. Table 3.2 shows the resulting 20 user stories. We define the “user” role as a member of the open-source team, since that group was mentioned most frequently in our data, and we therefore aim to match its requirements. The user stories do not capture every requirement the interviewees raised. Instead, we select those that are achievable, taking into account the existing tools that the system can build upon, which we discuss in Section 4.1. We categorize each user story into either analysis, layout, context or decision. The analysis category covers the core activity of determining the licenses of a project and its dependencies. It comprises both the automated identification of dependencies and their licenses and the manual review of the resulting findings. The first three user stories describe the automated foundation: identifying licenses, resolving direct and transitive dependencies, and expressing licenses under standardized identifiers. The next two analysis user stories concern the manual review of the findings that this automation surfaces. As such, they significantly influence the user flow through the interface and its central navigation elements. A final analysis user story asks for this automated analysis to run inside a CI pipeline, so that findings surface continuously during development.

The layout category includes user stories related to navigation and orientation throughout the interface. In contrast to the analysis category, the focus here is on overall interface navigation rather than on the review process itself.

The context category represents requirements to contextualize the project under review and its dependency packages with information not already contained in the metadata of the scan results. Context data can be sourced externally, by querying an online database, or internally, by aggregating data from the scan results. This category overlaps with the analysis category, since some context directly supports the review of the licenses of packages and findings.

Lastly, the decision category refers to the need to persist the decisions users make during the review process.

3.4. System requirements

Table 3.2. The 20 user stories stating the requirements for the system, derived from the thematic analysis of the expert interviews. Each user story is assigned to one of the four categories analysis, layout, context, and decision. The “user” role refers to a member of the open-source team.

ID	User Story	Category
US-01	As a user, I want to automatically identify the licenses present in a project and its dependencies, so that I obtain a complete basis for the analysis without compiling it manually.	analysis
US-02	As a user, I want the direct and transitive dependencies of a project to be determined automatically, so that I do not have to assemble the dependency list by hand.	analysis
US-03	As a user, I want licenses to be expressed using standardized license identifiers, so that they are unambiguous and comparable across packages.	analysis
US-04	As a user, I want to review the most critical license findings first, so that I can make decisions quickly and spot unexpected licenses early.	analysis
US-05	As a user, I want to review dependencies recursively, so that I can assess transitive dependencies when necessary.	analysis
US-06	As a user, I want to run the license analysis as part of a CI pipeline, so that new license issues are surfaced continuously during development rather than only before distribution.	analysis
US-07	As a user, I want to see as much information for a single package as necessary, so that I can make a decision for that package.	layout
US-08	As a user, I want explanations of the tool’s features and of any unclear terminology, so that I do not have to rely on external instructions.	layout
US-09	As a user, I want to see the file and the surrounding context of a license finding, and to expand that context on demand, so that I can check the finding in its surroundings.	context
US-10	As a user, I want to see a diff to the original license text when a finding is a whole license text, so that I can immediately spot any differences.	context

continued on next page

3. Requirements Analysis

Table 3.2 – *continued from previous page*

ID	User Story	Category
US-11	As a user, I want to easily estimate a project's complexity, so that I can gauge the effort its analysis will require.	context
US-12	As a user, I want to estimate the number of other projects that use a package, so that I get a sense of how widely used it is.	context
US-13	As a user, I want to see the number of GitHub stars, so that I get a sense of how popular a package is.	context
US-14	As a user, I want to see the level of transitivity of a dependency, so that I get a sense of how important a decision is.	context
US-15	As a user, I want to see the various licenses associated with a project, so that I understand its licensing situation.	context
US-16	As a user, I want to see whether a package's license has been decided without having to open the package details, so that I can easily see the status of the parent package.	context
US-17	As a user, I want to record the decision that a dependency or license finding is not relevant for license analysis, so that irrelevant dependencies (e.g. test dependencies) or findings in test or documentation files can be skipped in future analyses.	decision
US-18	As a user, I want to record which license, if any, is actually present in a finding of the automatic identification, so that I do not have to review the same finding again.	decision
US-19	As a user, I want to record the final license decision for a package after reviewing its findings, so that the review of that package is completed.	decision
US-20	As a user, I want to see when and by whom a decision was last edited, so that the review remains accountable and decisions can be traced back to their author.	decision

Having defined the categories, we now trace each user story to the findings of our thematic analysis presented in Section 3.3 it derives from.

Identifying the licenses of a project and its dependencies (US-01) follows from the aspiration in the *Future process* theme to automate license identification and to collect

3.4. System requirements

licenses by scanning source files, and from the *Tooling* theme, which names scanners for licenses in code files as an existing building block. It responds to the *Current process* finding that the work today is predominantly manual and time-intensive. Determining the dependencies automatically (US-02) is motivated by the *Current process* theme, in which the dependency list is delegated to the software developer and subsequently trusted rather than verified, and by the *Context* theme, which asks for direct and transitive dependencies to be listed and separated. The *Future process* theme records that determining the dependency list automatically is difficult, which we read not as a reason to drop the requirement but as motivation to reuse established tooling for it. Expressing licenses through standardized identifiers (US-03) corresponds to the *Future process* requirement that licenses be shown using a standardized license identifier.

The remaining analysis user stories concern the manual review of the findings that this automation produces. Reviewing the most critical findings first (US-04) reflects the prioritization of direct dependencies in the *Future process* theme and the consistent view across the *Motivation* and *Context* themes that direct dependencies are the most significant source of risk. Reviewing dependencies recursively (US-05) follows from the *Context* theme's separation of direct and transitive dependencies and from the *Future process* aspiration to dig deeper into transitive dependencies when needed. Running the analysis inside a CI pipeline (US-06) follows from the *Future process* theme, which asks for the tool to integrate into existing software development workflows such as version control and continuous integration systems, so that license issues can be surfaced continuously during development rather than only before distribution.

Turning to the layout stories, presenting as much information as necessary for a single package (US-07) draws on the *Future process* theme, which envisions a tool that provides enough context to make separate research unnecessary while keeping the information comprehensible and expandable on demand. Providing explanations of the tool's features and terminology (US-08) responds to the *Tooling* theme, which reports existing tools as difficult to use and poorly documented, and to the *Future process* requirement that the system be well documented.

Among the context stories, inspecting a finding within its surrounding file context and expanding that context on demand (US-09) combines the *Context* theme's emphasis on license texts from source files with the *Future process* aspiration to increase the level of detail on demand. Comparing a detected license text against the original by means of a diff (US-10) is grounded directly in the *Context* theme, which suggests showing the difference of detected license texts to the original texts. Estimating a project's complexity to gauge the required analysis effort (US-11) relates to the *Context* theme's call for information on software complexity and to the *Current process* finding that the analysis effort depends on the size and complexity of a project. Estimating how many other projects use a package (US-12) and showing its number of GitHub stars (US-13) both follow from the *Context* theme, which derives a dependency's trustworthiness from its popularity, the number of GitHub stars and its uses in other projects. Showing the level of transitiveness of a

3. Requirements Analysis

dependency (US-14) reflects the same separation of direct and transitive dependencies and the emphasis that direct dependencies are more critical to the analysis. Showing the various licenses associated with a project (US-15) draws on the *Context* theme's suggestion to list and classify the licenses used in a project. Indicating whether a package's license has already been decided without opening its details (US-16) is anchored in the *Future process* theme's call for a good overview.

Among the decision stories, recording that a dependency is not relevant for the analysis (US-17) follows from the *Current process* theme, in which non-distributed dependencies such as test dependencies must be separated out because they are a frequent cause of false positives, and from the *Context* theme's use of the directories of a dependency to judge its relevance. Recording which license, if any, is actually present in a finding (US-18) responds to the *Current process* theme, in which scanner findings contain a high number of false positives and uncommon or unknown licenses require manual research to interpret. Persisting the final license decision for a package after reviewing its findings (US-19) builds on these review activities and on the *Motivation* theme's goal of establishing a sound legal basis for the licensing of the software. Tracking when and by whom a decision was last edited (US-20) corresponds directly to the *Future process* requirement that decisions be saved together with the time and author of the decision, and to the auditability named in the same theme.

Besides these functional user stories, the interviewees voiced a number of non-functional requirements that concern the system as a whole rather than any single feature. The most emphasized is usability: the system should be easy to use and provide a good overview, including for the recurring review of updated projects. It should be performant, with acceptable runtimes ranging from about thirty minutes for a large project to at most a few hours, and it should run on ordinary development machines so that it can be used during early development as well as in CI. It should produce its results deterministically, remain available and stable, and be reasonably secure, in particular preventing data of the analyzed software from leaving the organization.

Technical Design and Implementation

This chapter describes the technical design and implementation of the system developed in this thesis. Since the requirements identified in Chapter 3 cover capabilities that established tools already provide, we first compare existing tools and select a foundation to build on. We then show how the contributed review tool combines with this foundation into a complete system, before presenting the implementation of the review tool itself. We develop the tool as a prototype. It should implement the requirements to the extent needed to evaluate the review approach in Chapter 5, rather than aiming for a production-ready system.

4.1 Tool comparison and selection

The foundational user stories US-01–US-03 describe capabilities that the analysis requires but that this thesis does not aim to reimplement: identifying the licenses present in the source files of a project, resolving its direct and transitive dependencies, and reporting the results under standardized identifiers. We therefore select an existing tool to provide this foundation and derive the selection criteria from the requirements. Table 4.1 summarizes the candidate tools against these criteria. The tool should identify licenses in source files (US-01, US-09) and analyze direct and transitive dependencies (US-02). Furthermore, it should report licenses under standardized identifiers (US-03), it should be integrable into continuous integration pipelines (US-06), and it should be extensible, so that the analysis can later be improved without replacing the foundation. A further criterion follows from the decision user stories US-17–US-20: the foundation should offer a means to persist the decisions made during analysis and reapply them on subsequent runs.

The ORT satisfies these criteria most completely. It covers the full range of required functionality, from dependency analysis through license identification in source files to reporting. These functions are extensible: the scanning stage is an orchestration layer around interchangeable backends rather than a fixed implementation. This means the foundation can be extended with additional, and possibly better, scanners later without changing the surrounding system. The ORT runs as a command-line tool, which makes it straightforward to integrate into any CI pipeline. It also provides mechanisms to store the decisions made during a license analysis and to apply them automatically on the next run, which addresses the persistence required.

4. Technical Design and Implementation

Table 4.1. Comparison of candidate tools against the selection criteria. ✓ = supported, (✓) = partial or limited, – = not supported. The highlighted row marks the ORT CLI, which we select as the foundation.

Tool	Dependency analysis	Source-file scanning	CLI / CI	Decision persistence	Review UI
ORT CLI	✓	✓	✓	✓	–
ScanCode Toolkit	–	✓	✓	–	–
ScanCode Workbench	–	✓	–	–	(✓)
FOSSology	(✓)	✓	–	✓	✓
ORT Server	✓	✓	(✓)	✓	(✓)
Licensee	–	–	✓	–	–
ClearlyDefined	–	(✓)	–	✓	✓

These persistence mechanisms, however, are not exposed through any interface designed for human use. They exist as a technical foundation in the form of configuration files. This is the gap that the present work addresses: The ORT provides the analysis and the underlying means to record decisions, but not a review interface through which a human can inspect findings and make those decisions efficiently. We therefore build on the ORT for the analysis foundation and contribute the review component on top of it.

ScanCode Toolkit is a capable license scanner, but it focuses on identifying licenses in a single project at a time and does not perform dependency analysis. It is, however, one of the scanner backends that the ORT supports. We therefore use ScanCode as the scanner implementation within the ORT, so that the ORT orchestrates the analysis while ScanCode performs the license identification.

FOSSology already provides a web interface with a review and approval process for package licenses and integrates a range of scanners, including ScanCode. It is, however, designed to be operated through its web interface, which makes it difficult to integrate into an existing CI pipeline.

Licensee likewise analyzes only one project at a time and so lacks dependency analysis [Licensee 2026]. It additionally identifies licenses only in project-level files, so it does not support the detailed inspection of licenses in source files that US-09 requires.

Both ScanCode and the ORT offer interface projects adjacent to their command-line tools, but neither fits the result-review needs of this work. ScanCode Workbench is a read-only viewer for scan results and therefore cannot support the recording of decisions. The ORT Server is geared towards scaling the execution of the ORT pipeline across multiple machines

by providing an interface to start jobs and view their results. Using the ORT through this interface moves away from the lightweight CI integration we aim for. Integrating a review interface into either of these projects would be possible in principle, but it would require more integration effort, particularly because neither is built for this purpose, which is more than is warranted for a prototype.

ClearlyDefined, finally, is better suited as a source of information than as a foundation, since it does not focus on an individual project but records manual decisions for the open-source ecosystem as a whole.

4.2 Integration with ORT

The goal of the system is to support the license analysis of a project and its dependencies, from the automated identification of licenses through to the human decisions that resolve the cases automation cannot. Within this system, the contributed review layer specifically aims to make the review of low-confidence license identifications easier to comprehend, faster, and less error-prone. Building such a system from scratch would mean reinventing functionality that established tools already provide, which runs counter to the practice of reusing existing open-source software rather than rebuilding it. As established in Section 4.1, the ORT already covers the automated part of the analysis, and even offers a mechanism to persist analysis decisions, but it exposes no interface through which a human can make those decisions efficiently. We therefore do not build a standalone tool but a review layer on top of the ORT pipeline, which together form the system described in this chapter.

Figure 4.1 shows the resulting system, consisting of the ORT pipeline and the contributed *License Scanner Review Tool*, and the dataflow between their components. To analyze a project, it is passed through the ORT pipeline, whose analyzer resolves the dependencies and whose scanner identifies the licenses in the source code. The ORT provides further pipeline components, among them an evaluator that applies policy rules and a reporter that generates output. These serve the broader goal of aiding license analysis, but our tool builds only on the output of the analyzer and the scanner, and we do not make further use of them. The review tool loads, aggregates and filters the output of the scanner component and augments it with information from external sources, in order to provide context for each case requiring review. This context is presented to the human reviewer, who decides on the license for each case. The ORT persists such decisions in a data structure it calls a *curation*, the same mechanism referred to as decisions in the requirements of Chapter 3. The tool writes the reviewer's decisions back as curations, which the next run of the ORT pipeline can then apply automatically to improve its output. While the ORT also allows curations to be stored elsewhere, for example in a central configuration directory, in our design they are part of the project to analyze, so the tool alters the project itself when writing curations. Storing the curations in the project keeps them close to the packages they curate and allows them to be shared across collaborators together with the project.

The system thus extends the ORT pipeline with an interactive feedback loop. The

4. Technical Design and Implementation

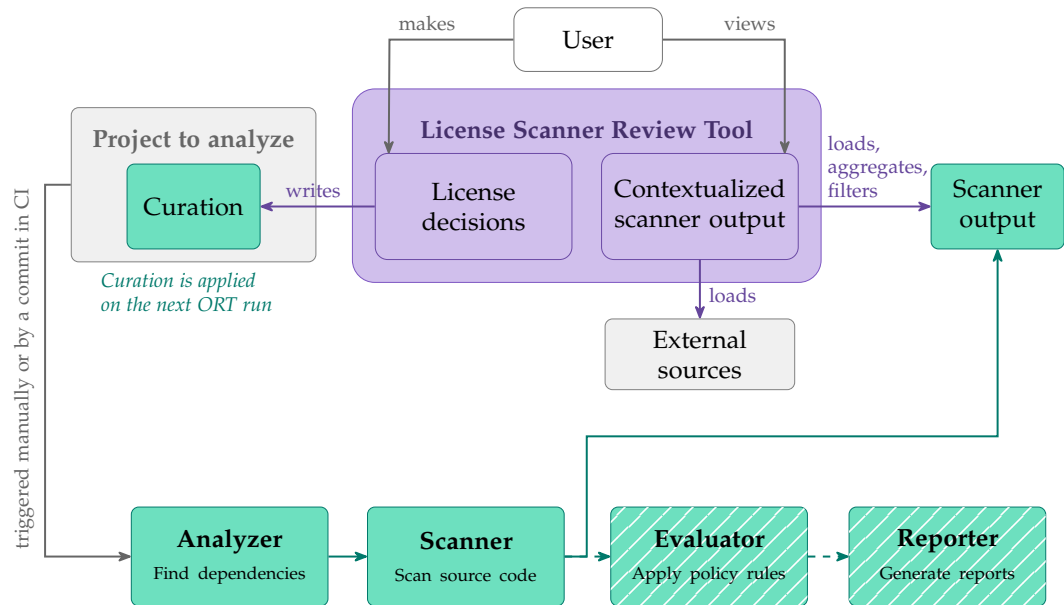


Figure 4.1. The data flow through the system consisting of the ORT (turquoise) and the contributed *License Scanner Review Tool* (purple). To analyze a project it is passed into the ORT pipeline consisting of analyzer, scanner, evaluator, reporter, and further components not shown. Our tool loads, aggregates and filters the output of the scanner component and augments it with information from other external sources to provide context for the user. The user uses this context to decide which licenses were found by the scanner and which licenses apply to the analyzed project and its dependencies. The tool persists these decisions into a data structure defined by the ORT named curations. In the subsequent runs of the system these curations can be applied by the ORT to improve its output.

components of the ORT pipeline themselves remain unchanged and run without user interaction. Such a run is currently triggered manually, but could in the future also be triggered by a commit, in case the ORT pipeline is used in CI. Only the review step is interactive: The automated pipeline produces the findings, the human reviews the low-confidence ones with the help of the provided context, and the resulting curations feed back into the next automated run. In this way the manual effort is confined to the decisions that genuinely require human judgment, while everything around them stays automated.

4.3 The License Scanner Review Tool

This section describes the details and implementation of the *License Scanner Review Tool*. It highlights the considerations that went into its design and explains noteworthy changes made during the development process.

Throughout this section we illustrate the tool using the test project that we also used

4.3. The License Scanner Review Tool

The screenshot displays the License Scanner Review Tool interface for the 'requests' package. The interface is divided into several sections, each labeled with a red box and a letter (F1-F7).

- F1: Page header and breadcrumb navigation.** Shows the tool name 'License Scanner Review Tool' and the project path 'Your project: PyPi:requests:2.32.5'.
- F2: Package metadata panel.** Displays details for 'PyPi:requests:2.32.5', including PURL, Registry, Description, Homepage, VCS, Authors, Declared licenses (Apache-2.0), Detected licenses (Apache-2.0, BSD-2-Clause, ISC, LGPL-2.0-or-later, MIT, NOASSERTION), Dependencies (PyPi:certifi:2020.2.25, PyPi:charset-normalizer:3.4.4, PyPi:idna:3.11, PyPi:urlgrab3:2.6.3), Weekly downloads (378,509,327), and GitHub stars (54,046).
- F3: Dependency Graph.** A visual representation of the package's dependencies, showing a network of nodes and edges.
- F4: Conclude license panel.** Contains a 'Trust declared license' button and a 'Conclude license' button. Below the buttons, it shows 'Included licenses after review: Apache-2.0'.
- F5: License findings panel.** Displays 'Suggested declared license: SPDX expression' and '9 hidden findings: Apache-2.0 (9)'. It includes navigation buttons 'Prev' and 'Next'.
- F6: File context panel.** Shows the current finding (Finding 3 of 10) for 'BSD-2-Clause' in the file 'docs/themes/LICENSE-35-45'. The text includes a copyright notice and a disclaimer.
- F7: Diff panel.** Shows the difference between the detected license text and the canonical BSD-2-Clause license text. The detected text is highlighted in red, and the canonical text is in green.

Figure 4.2. The tool review page, showing details of the requests package. The annotated regions are: (F1) the page header and breadcrumb navigation, (F2) the package metadata panel, (F3) the dependency graph acting as a minimap and progress indicator, (F4) the conclude-license panel with its two review actions, and (F5) the license findings panel, which contains (F6) the file context panel for the current finding, including the navigation to the previous and next finding, and (F7) the diff of the detected license text against its canonical original.

4. Technical Design and Implementation

during development [Haß 2026]. It is assembled to represent a realistic set of dependencies typical of a scientific Python project, with the direct dependencies `numpy`, `pandas`, `matplotlib`, `scipy`, `seaborn`, and `requests`. Together with their transitive dependencies, the project depends on a total of 25 Python packages. The figures in this chapter show the tool reviewing this project.

The tool is made up of two main components, the frontend, and the server. The frontend provides the user interface that is based on web technologies and thus runs in the browser.

4.3.1 User interface structure

The user interface is made up of three pages, the project overview page, the package review page, and the about page. The initial page the user sees shows the project overview with a link to the source repository of the project under review and a list of the project's direct dependencies. Each of the direct dependencies links to its review page shown in Figure 4.2. Throughout this section we annotate the screenshots with numbered labels of the form F_n , numbered continuously across all figures, which we use to refer to specific regions of the interface in the text. This page lets the user review the declared license and license findings of a dependency. Just below the header is a breadcrumb navigation ($F1$) that includes links to the review page of each package in the chain of parent packages from the currently selected package to the project overview. More specifically we use location breadcrumb trails that tell the user "where they are" [Instone 2002], giving them orientation and a way to go back up the dependency tree structure. The rest of the page splits into panels titled Package metadata ($F2$), Dependency Graph ($F3$), Conclude license ($F4$), and License findings ($F5$) that visually group information. The following subsections describe these panels in turn.

The about page contains a short description of the tool's purpose, lists what the user can do with the tool, and how the tool integrates into the ORT. At the top of all pages is a header that shows the tool's name, linking to the project overview, and a link to the about page.

4.3.2 Package metadata and dependency graph

The dependency graph ($F3$) in the upper right corner of the page acts as a minimap that helps users orient themselves in the dependency tree. Besides orientation, the size of the graph aims to give a visual sense of the project's overall size and thus a rough indication of its complexity, implementing US-11, while the depth of a node in the tree conveys the level of transitiveness of the corresponding dependency, implementing US-14. We draw the dependency tree of the project under review using Cytoscape.js [Franz et al. 2016], a graph theory library built for creating visualizations and interactive graphs. The nodes correspond to packages and the edges are directed from parent to child packages. The currently selected package is visualized by adding a blue border to its node and nodes that are filled green show that the package's license has already been concluded, giving the user

4.3. The License Scanner Review Tool

Package metadata ⓘ

ID

PyPI::requests:2.32.5

F8

PURL

pkg:pypi/requests@2.32.5

Registry

<https://pypi.org/project/requests/>

Description

Python HTTP for Humans.

Homepage

<https://requests.readthedocs.io>

VCS

<https://github.com/psf/requests.git>

Authors

Kenneth Reitz <me@kennethreitz.org>

Declared licenses ⓘ

Apache-2.0

F9

Detected licenses ⓘ

Apache-2.0

BSD-2-Clause

ISC

LGPL-2.0-or-later

MIT

NOASSERTION

Dependencies

PyPI::certifi:2026.2.25

PyPI::charset-normalizer:3.4.4

PyPI::idna:3.11

PyPI::urllib3:2.6.3

F10

Weekly downloads ⓘ

379,864,005

F11

GitHub stars ⓘ

54,187

Figure 4.3. The package metadata panel showing details of the requests package. It shows (F8) the ORT package identifier, the Package Uniform Resource Locator (PURL), authors, description, and links to the registry, homepage and Version Control System (VCS) of the package, (F9) its declared and detected licenses, (F10) its direct dependencies, each linking to its own review page, and (F11) the weekly download count and number of GitHub stars, enriched by the tool as indicators of the package’s popularity.

a visual indication of their progress through the review process. The name of a package is only shown when hovering over the respective node to allow for a tightly packed graph that does not take up a lot of screen space.

The package metadata panel (Figure 4.3) aims to gather the information needed to decide on a package’s license in a single place, implementing US-07. It shows the package identifier used by the ORT, the PURL, the authors, the description, and links to the registry, the package’s homepage, and its VCS (F8). Although the tool aims to provide any information necessary for the license analysis without leaving it, the links give easy access to further information in case the user is missing information. The declared and

4. Technical Design and Implementation

detected licenses are shown here as well (F9), aiming to give the user an overview of the licenses associated with the package, implementing US-15. The panel also lists the direct dependencies of the selected package (F10), each linking to its own review page. This list lets users navigate to packages deeper in the dependency tree, implementing US-05. For each listed dependency the list also shows whether its license has already been concluded and, if so, the concluded license, so that the review status of a child package is visible without opening it, implementing US-16. All the previously mentioned metadata comes from the ORT analyzer component. To help users gauge a package's popularity, the panel adds two statistics not present in the analyzer output. The number of downloads from the package's public registry in the last week, implementing US-12, and the number of GitHub stars (F11), implementing US-13.

4.3.3 Concluding a package's license

The license conclusion panel houses the central action of concluding the license of the currently selected package. To catch the user's attention it has a blue border that is also thicker than those of the other panels. It includes the instructions on what to do on this page. Either the user should first review the license findings and licenses of child dependencies and then return to this panel to finish the review by recording the conclusion or trust the declared license based on the package's popularity metadata.

Figure 4.4. The form opened by the “Conclude license” button. It allows to enter a concluded license in the form of an SPDX expression and optionally a comment. Above the form the licenses that should be considered when deciding on the concluded license are listed along with a hint to ensure their compatibility with the chosen concluded license. The hint includes links to license compatibility resources.

There are two buttons corresponding to these possible actions that both open a form that lets the user enter a concluded license and a comment. The difference between those actions is mostly semantic, because they create the same data structure. When using the “Trust declared license” button, the form is pre-filled with the declared license and a comment stating that the declared license is deemed correct. This aims to speed up the review process for popular or well-known packages that can be trusted. The “Conclude license” button opens the form shown in Figure 4.4 with empty fields, together with a hint to ensure that

4.3. The License Scanner Review Tool

the concluded license the user chooses is compatible with the licenses included in the package. Recording a conclusion this way implements US-19, letting the user complete the review of a package once its findings have been examined. These included licenses, namely those confirmed during review as present in the package’s files and those applying to its child dependencies, are shown in this panel to help the user arrive at the correct conclusion. The compatibility of these licenses has to be considered when deciding on the license that applies to the package. To this end, additional information in the form of the DLR open-source brochure [Brochhagen et al. 2022] and the license compatibility checker provided by the European Union [Interoperable Europe 2026] are linked above the form.

4.3.4 Reviewing license findings

The license findings panel (Figure 4.5) provides the interface for reviewing licenses that the scanner found in any of the package’s source files. The number of license findings varies widely between packages: in our test project it ranges from five to 2,709 findings. This presents the challenge to create an interface that both enables the user to handle large numbers of findings in an acceptable amount of time while not overloading them with too much information shown at once. To solve this our interface shows only one finding for review at a time and exposes functionality that is designed to help deal with large numbers of findings. To navigate through the findings the interface provides a button to show the previous finding and another one to show the next finding.

The tool orders the findings by a priority based on the filename and the detected license. Because the panel exposes only one finding at a time, this order determines the sequence in which the user encounters the findings when stepping through them with the navigation buttons. Ordering the findings this way implements US-04. It is intended to let the user encounter the most authoritative and most review-worthy findings early and stop once confident that no further unexpected licenses remain. Equation 4.1 defines this priority and the following introduces its components and the rationale behind it. Let f denote a license finding defined by a detected license, a confidence score, a file path, and start and end lines in that file. We write $b(f)$ for the lower-cased base name of that file path and $\ell(f)$ for the detected license. Let L be the set of license identifiers occurring in the package’s declared SPDX expression, which represents what the package claims about its own licensing. Let $\mathcal{M}$ be the set of recognized manifest filenames, and let $\mathcal{D}$ be the set of filenames matching the LICENSE, NOTICE, and README declaration filenames. The priority of a finding is then defined as

$$\text{priority}(f) = \begin{cases} 0 & \text{if } b(f) \in \mathcal{M}, \\ 1 & \text{if } b(f) \in \mathcal{D}, \\ 2 & \text{if } \ell(f) \notin L, \\ 3 & \text{otherwise,} \end{cases} \quad (4.1)$$

where the conditions are evaluated from top to bottom and the first satisfied condition determines the priority. The findings are then ordered by ascending priority, so that lower

4. Technical Design and Implementation

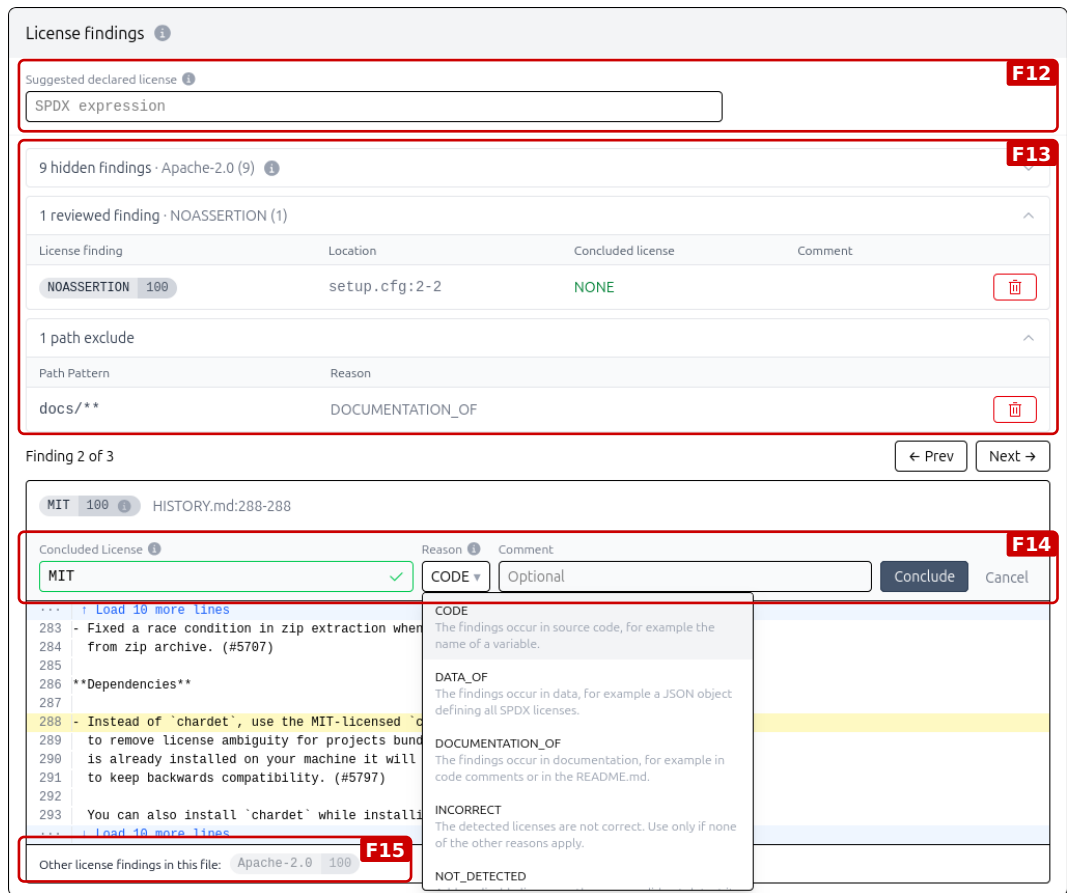


Figure 4.5. The license findings panel. (F12) The suggested-declared-license input overrides the declared license used for hiding findings. (F13) Hidden findings, reviewed findings, and path exclusions are shown in collapsible tables. (F14) To allow concluding the license of a finding a form is shown in the file context panel. (F15) Other license findings in the same file are shown below the file content.

priorities are shown first. Priority 0 contains project manifest files such as `pyproject.toml`, `package.json`, and `setup.cfg` files. These are treated as the most authoritative source, because they carry the machine-readable license declaration that the maintainers define and publish alongside the package. Priority 1 contains `LICENSE`, `NOTICE`, and `README` files, which constitute the canonical human-readable license declarations and are the documents a reviewer would consult to confirm a package’s licensing. Priority 2 contains findings whose detected license is not part of the declared SPDX expression, that is, $\ell(f) \notin L$. Such undeclared licenses are prioritized because they represent a discrepancy between what the

4.3. The License Scanner Review Tool

package declares and what the scanner detects, which is precisely the situation that warrants manual review. Priority 3 contains all remaining findings, which are typically matches in individual source files whose license is already covered by the declared expression and therefore add little new information. The priority serves as the primary sort key. Findings sharing a priority are ordered by the finding's confidence score in descending order, so that within each priority the matches in which the scanner is most confident are shown first. The interface groups this information into a panel nested inside the license findings panel (F6). The tool shows the file contents with five lines before and after the finding, aiming to let the user check a finding in its surrounding context, implementing US-09. Buttons above and below the file contents allow loading more lines from the file in batches of ten lines, allowing the user to view more context on demand. Below the file contents, the other license findings located in the same file are listed (F15), giving an overview of the file's licensing. The detected license and the confidence score are shown directly above the file contents. Three buttons in the panel header let the user exclude the file path, or any of its parent paths, to record that the findings under a given path are not relevant for the analysis, implementing US-17, or conclude the license actually present in the finding, implementing US-18. One of the conclude buttons suggests a likely license to conclude, opening the resulting form with that license pre-filled, and the other opens the empty form. The suggested license to conclude to is either the detected license of the finding or `NONE` in case of a `NOASSERTION`. Concluding to `NONE` means that there is no license at the location described by the finding [SPDX Project 2024], so it is the correct conclusion for false-positive findings. `NONE` does not appear in the list of included licenses in the license conclusion panel, because it is not a license, explicitly signals that no license was found and is thus not relevant for the overall conclusion of the package's license. The tool does not suggest concluding a finding as `NOASSERTION`, because it would mean the user is unsure about what license is present, defeating the tool's purpose of increasing the confidence of what licenses are included in the project under review. All three forms in the license finding panel require the user to choose from a list of reasons defined by the ORT and allow to add a comment to the exclusion or conclusion. The available reasons differ for exclusions and conclusions. For each reason the tool shows an explanation for when to use it. One such form with the extended reasons list is shown in Figure 4.5 marked F14. Exclusions and conclusions are shown in collapsible lists above the finding context panel after they have been created and can be deleted via those lists. The exclusion list contains the path pattern that is excluded and the respective reason. Concluded findings are shown with the detected license, the confidence score, the locations of the finding, the concluded license and the user defined comment. To handle the possibly large number of findings, the tool hides findings that match the declared license of the selected package with a confidence score of 100 in a list above the finding context panel. Because these findings repeat the declared license and thus do not help in finding any licensing problems like incompatibilities, hiding them should not impact the utility of the tool. All three of those lists shown in F13 are collapsible and are collapsed by default to not overload the user

4. Technical Design and Implementation

with too much information. Another way to reduce the number of license findings to review is to suggest a declared license that the tool uses instead of the declared license from the metadata (F12). In the analyzer results of our test project declared licenses were frequently missing because the ORT analyzer is unable to extract them [ORT Project 2026a]. This breaks the hiding of findings matching the declared license. The scanner on the other hand usually detects these, so combined with the sorting explained earlier the users can quickly deduce the declared license from one of the first findings shown to them and enter it as a suggestion, allowing the tool to appropriately hide findings. It also helps to review packages that declare the wrong license, by first correcting it via the suggestion mechanism and then reviewing the remaining license findings.

4.3.5 Comparing altered license texts

The license scanner does not only find license identifiers confined to a single line, but also complete license texts. When a license finding references a complete license text, chances are high that the license text has been altered in some way, and such changes can explain a confidence score below 100. To make them visible the tool shows a comparison, also called a diff, against the original license text (F7), implementing US-10. The diff marks each line as unchanged, added, or removed, so the nature and extent of a deviation, be it merely syntactic or a semantic difference introduced by added or removed terms, is visible at a glance rather than something the reviewer has to reconstruct by hand. Most license analysis tools, including the scanner our tool builds upon, rely on name-based rather than term-based identification which we introduced in Section 2.2, and we follow this approach. The diff nevertheless gives the reviewer a limited term-level view: by surfacing the added and removed lines it makes deviations in the license’s actual terms visible, so the reviewer can judge whether a change is cosmetic or alters the obligations. An alternative we considered was to place the detected text and the original side by side and leave the comparison to the reviewer. We rejected this because it shifts the work of locating and classifying the differences onto the reviewer, which is precisely the effort the diff is meant to eliminate, and because the burden grows with the length of the license text and the number of changes. To compute the diff we use an implementation of the difference algorithm proposed by Myers [1986]. We chose this algorithm in particular because it is the default diff implementation in Git [Nugroho et al. 2020], so developers are already accustomed to reading its output and need not learn an unfamiliar representation in order to interpret the comparison.

4.3.6 User guidance through tooltips

The tooltips shown throughout the interface aim to improve the tool’s learnability and to let users understand its features and terminology without relying on external instructions, implementing US-08. Tooltips provide explanations on demand, shown when the user hovers over an element, and thus add guidance without permanently occupying screen

4.3. The License Scanner Review Tool

space or interrupting the review flow [Isaksen et al. 2017]. They are made accessible through information icons placed next to the elements they describe, and the corresponding tooltip is shown when the user hovers over or clicks such an icon. The interface contains sixteen of these icons. The tooltips they reveal serve four purposes: explaining the panels that divide the interface, guiding the user on what to enter into an input field where this is not obvious, explaining domain-specific terminology, and clarifying other functionality and displayed data. Four tooltips explain the purpose of the package metadata, dependency graph, license findings, and diff panels. The conclude-license panel instead shows its explanatory text permanently, so no tooltip is added there. Domain terminology is explained where it appears: tooltips next to the declared and detected licenses explain the distinction between them, and a tooltip explains the meaning of a finding's confidence score. Several tooltips clarify the data the tool displays or derives, namely the source of the weekly download count and the number of GitHub stars, the meaning of the "included licenses after review" and "suggested declared license" features, and the reason for which certain findings are hidden. Input fields are explained where their expected content is not self-evident: the concluded license of a package and of a finding, the path pattern of an exclude-path curation, and the reasons selected for finding curations each carry an explanatory tooltip. A special case is the warning icon shown when a package's declared license is missing or invalid. Hovering over it reveals a tooltip that explains the warning and points the user towards the suggested-declared-license feature as the way to handle the situation. This connects the problem directly to its remedy at the point where the user encounters it.

4.3.7 The server component

The frontend communicates via HTTP with the server component, which is responsible for reading from and writing data to the file system. The server also handles loading data from external Application Programming Interfaces (APIs). The server exposes Create, Read, Update, and Delete (CRUD) operations for the path excludes, finding curations, and license curations that the user creates during review. These are read from and written back to the file system in the format expected by the ORT, so that a subsequent ORT run can apply them. A central endpoint returns the scan result that the frontend displays. For each package it combines the analyzer result with the scanner result, so that the frontend can present them together. The analyzer result provides the package metadata, while the scanner result provides the license findings. The analyzer and scanner results are stored in a YAML file that the server reads and parses. For our test project this parsing takes several seconds, which would make a per-request parse noticeably slow. The server therefore parses the result file once at startup and serves subsequent requests from the parsed data held in memory. This keeps the interface close to instantaneous in response to user interaction, at the cost of requiring a restart of the server when the result file changes. Further endpoints return the dependency graph and, for a given finding, the contents of the file it occurs in, which supplies the source context shown around the finding. The file-content endpoint reads these files directly from disk. It expects the source files to be present alongside the

4. Technical Design and Implementation

scanner output, rather than retrieving them itself. Placing them there is the task of the ORT's downloader component, which uses the analyzer output to download the correct version of each package's source code. The downloader has to be run manually after the analyzer, and before the source context of a finding can be shown in the tool. It is not shown in the pipeline of Figure 4.1 because it is not required to run the ORT on its own. We use it as a convenience to obtain the source files, avoiding the need to reimplement the download logic ourselves. Beyond the ORT output, the server retrieves information from external sources. These are the canonical license texts from the ScanCode license database, the number of GitHub stars from the GitHub API, and the download statistics, obtained from the npm API for npm packages and the pypistats.org API for Python packages. The frontend does not query these sources directly. Routing the requests through the server avoids cross-origin restrictions that could otherwise block them in the browser. The external APIs impose usage limits, and when a limit is reached the corresponding information is simply not shown. To stay within these limits, and more generally to reduce network traffic both for the user and for the external sources, the server caches the retrieved data. The cache lifetime is defined per source according to how often the data changes and what each source permits. GitHub stars are cached for one hour, which matches both their tendency to change regularly and the point at which the GitHub API permits the same request again [GitHub 2026]. The npm and Python Package Index (PyPI) download counts are cached for twenty-four hours. PyPI updates its statistics only once per day [Python Software Foundation 2026], so caching a weekly figure for a day is a reasonable fit, and npm's usage limit is not documented. The license texts are cached for thirty days, as they rarely change if ever. An updated license is usually published under a new identifier rather than altered in place, so the texts can be held far longer. The cache itself can be kept in memory or persisted to a file, selectable through an environment variable.

4.3.8 Technology and usability

The server component is implemented in Python using the FastAPI framework, a choice aimed at keeping the tool maintainable. FastAPI generates an OpenAPI specification for the exposed endpoints out of the box, from which we in turn generate the API client code used by the frontend. The interface between the two components is thus maintained in a single place, the server, rather than kept in sync by hand. Whenever an endpoint changes, regenerating the client propagates the change to the frontend, where mismatches then surface as type errors instead of failing requests at runtime. This should reduce both the maintenance effort and the risk of the components drifting apart as the tool evolves. The frontend is built with Vue.js as a single-page application. Navigating between the project overview and the review pages of individual packages therefore does not reload the page, giving a smooth user experience when changing views, which matters in a review process that moves through many packages. This choice of a JavaScript frontend deviates from the boundary condition of implementing the tool primarily in Python, and we reflect on this deviation in Section 6.2. In addition, Vue's reactivity system makes it

4.3. The License Scanner Review Tool

easy to propagate status changes across the interface. When the user concludes a package's license, for example, the corresponding node in the dependency graph turns green and the status shown in the dependency list updates, without any additional wiring between these components. Since the interface is web-based, the established Web Content Accessibility Guidelines [Caldwell et al. 2008] apply to it, and we aim to improve its usability by implementing basic ones, such as keyboard navigation of all interactive elements and a minimum text contrast ratio as defined by the AA rating. Vue's component system supports this, as recurring interface elements like tooltips and collapsible lists implement the keyboard interaction once and reuse it wherever they appear.

Evaluation

This chapter describes the evaluation of the prototype in a usability study with ten practitioners. We let the participants use the prototype and collect their feedback both while they use it, by asking them to think aloud, and directly afterwards, through a questionnaire. We first describe the preparations for the study, in particular the scanned test project and the deployment of the prototype, and then detail the course of an evaluation session, including the tasks and the questionnaire. After explaining how we analyze the collected data, we report the results of the evaluation.

5.1 Evaluation Setup

This section describes the preparations required before the evaluation sessions can take place: obtaining the scan results that the participants review and making the prototype accessible to them. Since the purpose of the prototype is to review the output of an ORT scanner run, this run has to happen before the prototype can be used and evaluated. To run the scanner we need a project to run the scanner on. To this end we use the test project [Haß 2026] that we also used during development. Running the ORT scan on this test project, downloads all 25 packages and scans every file in all of these packages. This scan step takes several hours when started without any existing cache on a computer with 16 CPU cores and 16 GB of RAM. For this reason it is not feasible to run the scan in every evaluation session, because it would result in long evaluation sessions. The prototype itself does not take several hours to run, since it is an interactive web-interface and thus by its nature not a long-running task the participants would have to wait for. It does however require docker installed on the computer that should run it or a working Python installation and some manual setup. Since the participants of this evaluation are not all located in the same place, the easiest and most resource-efficient way to run the evaluation is remotely in online meetings. This also means that it is not easy to provide the participants with a device that has the prototype set up to easily use it. We ask the participants to use their own devices instead. To circumvent the time intensive scanning process, we run the scan once before starting the evaluations and let every participant use the prepared scan results [Haß 2026] during the evaluation. We adjusted the prepared scan results slightly to create test cases that become progressively more complex and challenging to review, while still being realistic. With these adjustments we also added back some missing declared licenses that

5. Evaluation

are caused by the ORT analyzer deficiency described in Chapter 4. To let the participants use their own devices without needing to install the prototype on them, we deploy the prototype to a server that the participants can access via the Internet. With this setup the participants only need a web browser to use the prototype. This means that the evaluation only tests the usability of the prototype and not the usability of the complete system.

5.2 Procedure

This section describes how an evaluation session unfolds, from the introduction through the tasks to the concluding questionnaire, and closes with the pretest of the study design.

Each evaluation session follows a written guideline to keep the procedure consistent across participants. A session starts with an introduction that explains the goal of the evaluation, outlines the two parts of the session, and obtains the participants' consent to record the session. The introduction explicitly states that there are no right or wrong answers and that the evaluation targets the software and not the participant. This framing follows Boren and Ramey [2000], who recommend casting the system under test as the object of study and the participant as a domain expert to avoid self-conscious or apologetic behavior. The participants then share their screen and receive a document containing the scenario, the tasks, and the questionnaire [Haß 2026], which guides them through the remainder of the session. During the tasks we keep our own contributions to a minimum and announce beforehand that we will mostly stay silent apart from reminders to think aloud. We limit our interventions to occasional hints when participants do not record their decisions in the tool or run out of time on a task. When participants encounter defects of the prototype, we do not assist and record the resulting situation instead. In the introduction to the evaluation we present this scenario to the participants as described in Figure 5.1.

We define a set of tasks that the participants are asked to work on. Figure 5.2 shows the tasks given to the participants. The tasks generally require them to conclude licenses of specific packages in the dependency tree of the test project. Task 1 is the exception since it asks the participants to explore the tool and its functionality freely. We also ask them to not click on the package that they are supposed to review in the consecutive tasks. It does not have any success criteria, since we want the participant to explore until they think they know enough about the tool to use it. We only limit the time to a maximum of ten minutes, to keep the evaluation session brief. Task 2 asks the participants to navigate to a particular package, review its license and conclude the license to what they think is the correct license. This package is a leaf in the dependency tree of the test project, so it has no other dependencies. It also has a valid declared license and the scan results include only license findings with a confidence score of 100 that match the declared license. This makes the package an easy case where no special consideration should be necessary before concluding the license. Task 3 has the same structure as Task 2. It requires the participants to review another package, though. The scan results for this package include five license findings that have a confidence score below 100 or do not match the declared license,

Scenario presented to participants

Imagine you received a software project from a researcher, and they asked you to check the dependencies for license compatibility. You decided to use the OSS Review Toolkit and ScanCode to scan the project for any included licenses, by running

```
docker run --rm -v $(pwd):/app -w /app ghcr.io/oss-review-toolkit/ort:82.0.0 scan -i ort-out/analyzer-result.yml -o ort-out
```

in the project directory downloaded to your computer. You receive a large scan-result.yml with over 100,000 lines. To review these results you start the License Scanner Review Tool by running

```
docker run --rm -p 8000:8000 -v $(pwd)/ort-out:/data/ort-out lukashass/license-scanner-review-tool:0.0.3
```

For the purpose of this evaluation we assume that the scan is already finished and the License Scanner Review Tool is already started. You can access it at <https://example.com/>.

Figure 5.1. Scenario presented to participants in the introduction of the evaluation.

making this case more difficult to review. One of the cases of a confidence score below 100 for a finding matching the declared license is caused by a changed license text compared to the canonical license, resulting in the user seeing the diff feature (F7) for changed license texts. The other is a simple reference to the license. The expected action for both these cases is the confirmation of the detected license. The remaining three cases are split into one NOASSERTION which we expect to be concluded to NONE as it is only a reference to a LICENSE file and two cases of license findings in test or example files that are not included in the distribution and can be excluded by the “Exclude path” feature. Task 4 is the most complex, asking participants to review and conclude the licenses of the complete subtree of `PyPI::pytest-cov:7.0.0`. The previous tasks are designed so that two packages in this subtree should already be reviewed and concluded. The remaining five packages shall be reviewed and concluded as part of this task. For two of those five packages the scan results are missing a declared license, allowing us to evaluate the usability of the tool in this case and the feature of suggesting a declared license that is designed to make this case easier to handle for the user. One package stands out by having not just a single license as declared license, but a disjunctive SPDX expression instead. This case is interesting because there are license findings that detect only one of the operands of the disjunction with a confidence score of 100. Another interesting property of two packages in the subtree is that the declared license is the BSD-2-Clause license and numerous license findings detect

5. Evaluation

the BSD-3-Clause license with a confidence score of 99. The cause of this false detection is the inaccurate declaration of the license in source files as simply “BSD”. Because Task 4 includes more than one package and not all of them are leaf packages, the participants also have to consider the order in which they review the packages. The ideal order is from leaf to root. To conclude the license of a package we need to include the licenses of its dependencies in the decision, in case we do not trust the authors to have chosen a license that is compatible with the licenses of the package’s dependencies. The tool only makes the licenses of dependencies visible once they are concluded.

To collect feedback during the usage of the prototype, we ask the participants to think aloud, one of the most widely used methods in usability testing [Boren and Ramey 2000]. Task 0 lets the participants practice thinking aloud on a simple arithmetic problem before they start working with the prototype, a practice step that is recommended in the think-aloud literature [Boren and Ramey 2000]. After the participants finish the tasks, they answer a questionnaire consisting of 13 statements and four questions. The first ten statements comprise the System Usability Scale (SUS) questionnaire and focus on the general usability of the prototype. The SUS gives a view of subjective assessment of usability [Brooke 1996]. The statements use a Likert Scale [Likert 1932], with five answers ranging from 1 - strongly disagree to 5 - strongly agree. To prevent response biases, half of the statements expect the common answer to be strong agreement and the other half strong disagreement. Additionally, the positive and negative statements are alternating to ensure participants read carefully and think about the statements before answering. The remaining seven items — three further Likert statements and four free-text questions — are more specific to our system and its integration into the participants existing license analysis workflows. The three statements continue using the Likert Scale, also used by the SUS, while the four questions ask for free-text answers.

To identify potential errors with tasks and survey questions it is good practice to apply pretesting methodologies before conducting an evaluation [Geisen and Bergstrom 2017]. In our case a review from an expert in usability survey design found no problems with our evaluation design. Additionally, we conducted one test run to find as many potential errors before starting the actual evaluations. During the test run we looked for any problems related to the usability and understanding of the tasks and questionnaire, as well as logistical difficulties that could arise during the evaluations [Geisen and Bergstrom 2017]. As a result of this pretest, we clarified to which package participants should navigate before starting the license review in tasks 2 and 3.

5.3 Data Analysis

The evaluation sessions produce two kinds of data: the recorded think-aloud protocols and the answers to the questionnaire. This section describes how we analyze each of them.

To analyze the think-aloud protocols, we use thematic analysis described in more detail in Section 3.2. We apply thematic analysis to the think-aloud protocols in the same way

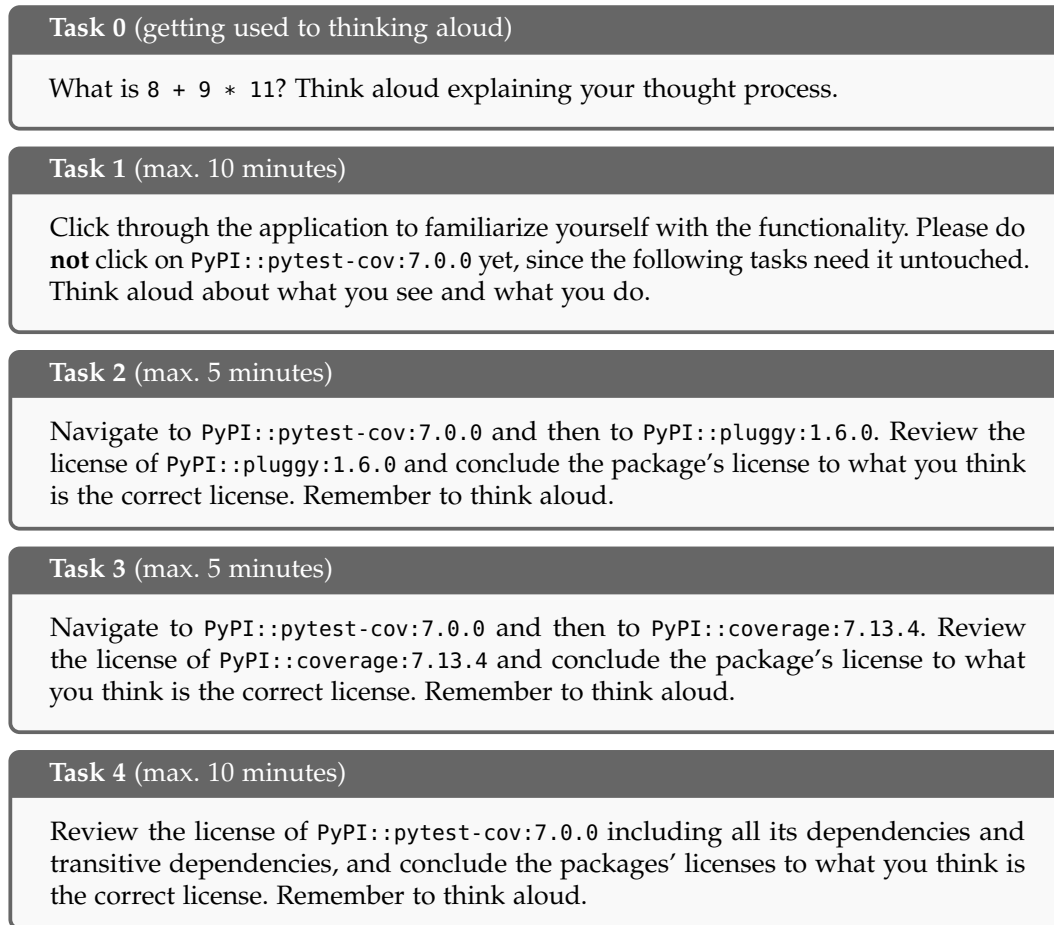


Figure 5.2. The tasks presented to participants during the evaluation. Task 0 allows for the participants to get used to thinking aloud and to dial in on an appropriate detail of uttered thoughts. Task 1 allows the participants to discover what they can do with the system. Task 2, 3 and 4 ask the user to conclude licenses of packages and become consecutively more complex.

as to the interview transcripts described in Section 3.3. While the method of inducing participants utterances differs between directly asking questions in the interviews and prompting them to vocalize what they see, do and think, both result in textual statements that can be analyzed using thematic analysis.

To calculate the SUS score from the answers to the first ten statements of the questionnaire, we sum the score contribution of each statement and afterwards multiply by 2.5. The score contribution depends on the statement's sentiment. For positive statements the

5. Evaluation

contribution is the scale position minus 1 and for negative statements the contribution is 5 minus the scale position.

5.4 Results

This section reports the results of the evaluation. We first describe the participants and their experience with license analysis, then the usability of the prototype based on the SUS scores and the thematic analysis of the think-aloud protocols, and finally how well the tool fits into the participants' existing license analysis processes.

5.4.1 Participants

We recruited a total of ten participants through three channels: the DLR open-source team, a DLR-wide call for participation targeting people with experience and/or knowledge with open-source licenses, and personal connections. Out of the ten participants, nine have a technical background with a degree in computer science or aerospace engineering and one has a legal background with a law degree. Eight of the participants currently work in a technical role, one in a legal role, and one in a management role. As a result of trying to find a larger number of participants, they have on average less experience with license analysis, compared to the four participants of the requirements interviews. Their years of experience vary widely from one year to 25 years with a median experience of five years. The self-reported level of experience shown in Figure 5.3 widely differs between specific tasks within license analysis. While most participants assess their experience with the overall process of license analysis as average or above average, they do not have the same level of experience for all sub-tasks. They report a similar level of experience with identifying what license a software package includes and is distributed under. Experience with finding and analyzing conflicts between licenses and determining their compatibility is reportedly lower with nearly half of the participants being inexperienced or very inexperienced. Most of the participants report a low level of experience with using software in the license analysis process. This is a noteworthy result, given that using software for license analysis is exactly what we ask them to do in our evaluation.

5.4.2 Usability

In this section we describe the usability of our tool based on the SUS scores and the analysis of the think-aloud protocols.

The mean SUS score is 69.5 on the 0 to 100 scale or close to "good" on the adjective rating scale created by Bangor et al. [2009]. Bangor et al. [2009] found a highly correlated adjective rating that helps to make sense of raw numeric scores. They assigned mean SUS scores to the seven adjectives "Worst Imaginable", "Awful", "Poor", "OK", "Good", "Excellent" and

5.4. Results

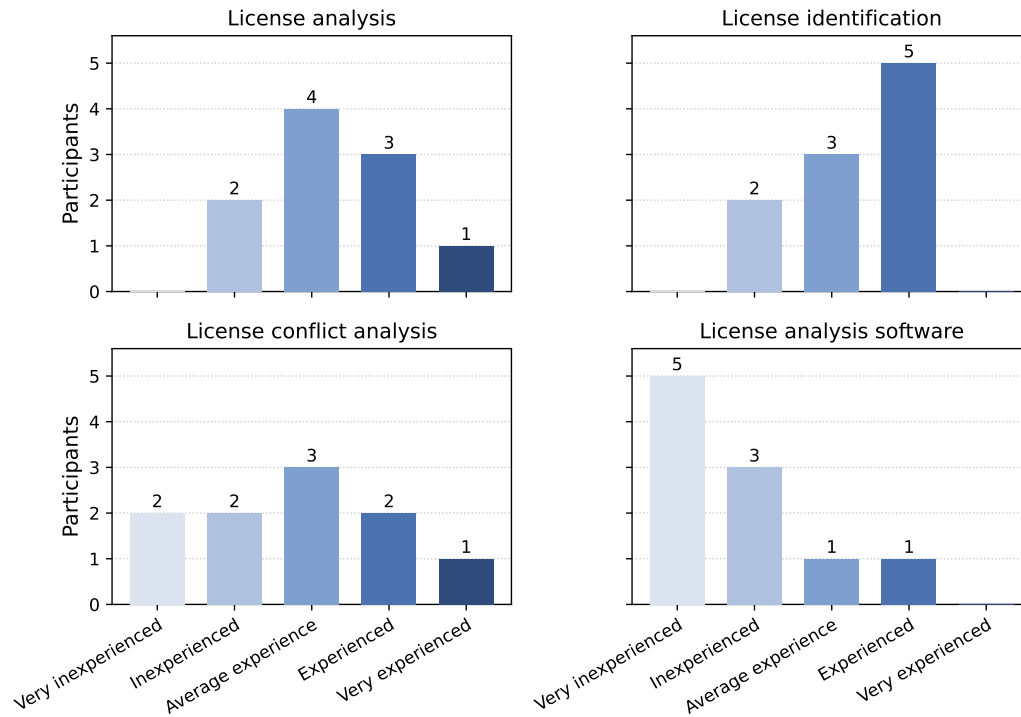


Figure 5.3. Distributions of self-reported experience with license analysis, license identification, license conflict analysis and license analysis software. Most participants report average and above average experience with license analysis in general. Experience with license identification is similar, but missing a very experienced participant. License conflict analysis experience is broadly distributed across the spectrum. The participants report the least experience with using software to do license analysis.

“Best Imaginable”. “Good” correlates to a mean SUS score of 71.4 with a standard deviation of 11.6.

While our mean score is close to “good”, it also has a standard deviation of 13.5 and the scores of the individual participants vary notably as can be seen in Figure 5.4. The lowest score 47.5 can be described as almost “OK” and the highest score 90 is close to the “Best Imaginable” usability [Bangor et al. 2009]. The three lowest scores — identified by P1-3 in Figure 5.4 — stem from participants that either reported low experience with license analysis or a non-technical background, which may indicate that these factors contributed to the lower scores.

Possible reasons, as to why the SUS score is not perfect can be found in the thoughts uttered by participants while they used the tool. We analyze the think-aloud protocols using thematic analysis as described in Section 5.3. The complete dataset of themes, categories,

5. Evaluation

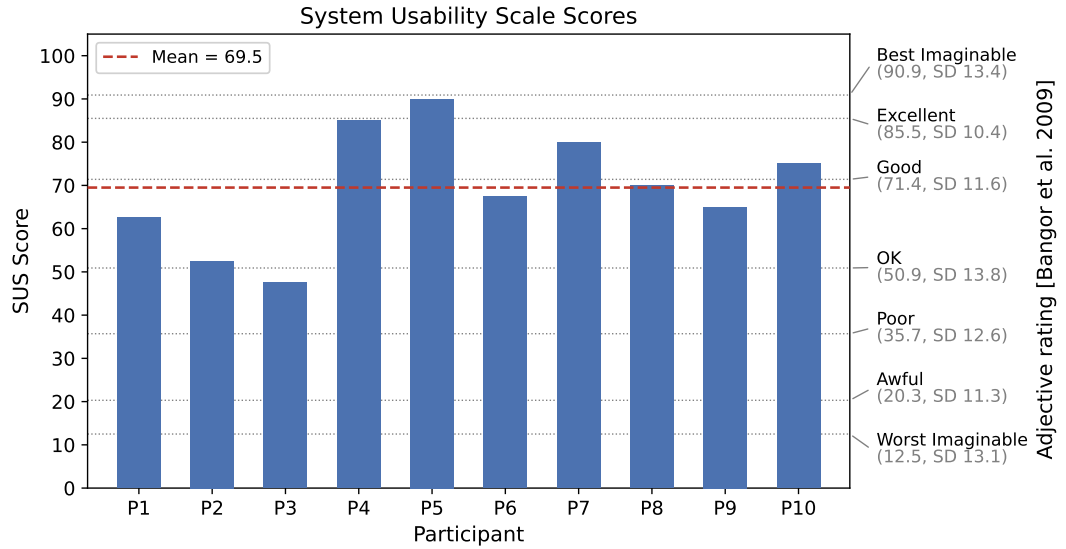


Figure 5.4. System Usability Scale scores per participant. The mean of 69.5 corresponds to a rating close to “good”; individual scores range from 47.5 to 90. The three lowest scores (P1-3) come from participants who reported low license-analysis experience or a non-technical background.

and codes is available at [Haß 2026]. We now describe the data, by defining the themes and categories and discussing noteworthy codes therein. The resulting thematic analysis tree is made up of eight themes, a total of 17 categories and 216 codes. The themes are Correct understandings, Actions, Assumptions, Insecurities, Task specifics, Bugs, Sentiments, and Wishes:

Correct understandings The *Correct understandings* theme provides insights into what parts of the tool participants understand correctly. It includes a total of 21 codes, which do not describe the full picture of understanding that the participants showed by mostly using the tool the way we intended. We will not go into details of the included codes since they describe the intended usage already laid out in Chapter 4.

Actions The *Actions* theme describes what participants did while solving the tasks. It splits into three categories and includes a total of 40 codes. We categorize actions into expected, unexpected and external actions. External actions include looking into the source repository of the project under review. The two reasons for this are getting more context and searching for the declared license to double-check what the tool shows. Participants also commonly left the tool to look up unknown licenses and unknown terminology used by the tool. Actions that we did not expect the participants to take include, using the “Exclude path” feature to handle reference findings and trusting the declared license of a package based on findings of the same license in prominent

metadata files. These actions are in line with and can be explained by the insecurities the participants uttered. Expected actions mirror the intended usage described in Chapter 4.

Assumptions The *Assumptions* theme is relatively small and includes only six codes. The codes in this theme, for example the assumption that the project metadata includes the ground truth for license decisions, help to explain some of the unexpected actions and insecurities.

Insecurities The *Insecurities* theme describes what parts of the tool caused confusion for the participants, where they were unsure or hesitant and what misunderstandings arose while using the tool. It is the largest theme with 67 codes that are split into six categories. Participants were commonly unsure what to do when seeing license identifiers they did not recognize. While this happened with identifiers of common licenses like the ISC license, the most confusion was caused by identifiers that are specific to the scanner ScanCode used in our evaluation system. They were also often unsure about the compatibility of shown licenses, which they need to know about to make accurate decisions about which license applies to a package.

Another factor contributing to the participant's confusion is the missing details on the sources of displayed information. The introduction to the task mentions that our tool is based on the ORT and ScanCode, but there are no indicators that show which piece of information originates from what component in the tool. Participants wondered about the source of the detected license and why it is possible that none is found when licenses were detected. Additionally, the tool does not show information on where license findings come from. In this case it is not clear if the participants would be satisfied with knowing that license findings are created using ScanCode or would rather want to have detailed information on how ScanCode created a license finding.

The evaluation also made apparent that the tool lacks explanation for terminology that is central to understanding and using it correctly. Participants are unsure about the meaning of SPDX expressions, the word conclude in connections with licenses and packages, and the special license values NOASSERTION and NONE. This terminology originates from the SPDX specification that we explain in more detail in Section 2.3.

The lack of explanations for this terminology also results in a lack of understanding of the tool's purpose. Participants were unsure what to do with the tool, asking for example why they should decide the license instead of letting the tool decide and why they should enter the concluded license into the tool.

A few participants had trouble finding and navigating to dependencies of direct dependencies. This surfaces a problem with layout clarity that we already noticed during pretesting and tried to remedy by a more detailed task description. Having one of the main navigation elements of the tool inside the metadata table is not immediately obvious to all users.

5. Evaluation

The misunderstandings category records mistaken beliefs that participants formed about the tool and its concepts. A recurring one was reading the link to the source repository as a link to the tool's own source code. Several beliefs concerned entering decisions into the tool: that a license decision counts as done without being entered into the tool, that trusting the declared license is merely a faster way of entering the concluded license, and that the conclusion comment is produced by the tool rather than written by the user. Other misjudgments include assuming that a missing declared license means no license exists, that `NONE` is a normal license identifier that would appear as an included license, or that findings are hidden when their score is low.

Beyond the six categories, a number of codes sit directly under the theme. The most frequent of these is being surprised or overwhelmed by the large number of detected licenses. Several participants noticed that the list of direct dependencies shown by the tool does not match the dependencies they found in the repository, and some did not immediately locate the fourth direct dependency in the graph. The conclusion reasons were a further source of uncertainty, both in choosing a reason for a correct finding and in distinguishing the two different reason lists. Reference findings caused repeated hesitation about how to handle them, whether the "Exclude path" feature applies to a whole directory or only a single file, and how to mark a `NOASSERTION` as a false positive when none of the offered reasons fit. The diff panel confused participants when it showed no difference. When the diff panel was not shown at all for findings of identifiers in files that also include license text, the participants also reacted with confusion. A recurring undercurrent was a reluctance to decide on the available information alone, expressed as wanting to consult the project source before trusting a declared license. Participants also found it hard to tell which part of a project a license finding applies to, which would require better insight into the internal structure of the package under review. Smaller points of confusion include two findings appearing for a single OR expression, and the poetry label used for the project root.

Task specifics The *Task specifics* theme captures reactions that stem from the design of the evaluation tasks rather than from the tool itself. It consists of nine codes that all concern time pressure and the volume of findings. The time limit per task left several participants feeling rushed and unable to review every finding as thoroughly as they would have liked. The number of findings in the packages of the reviewed `pytest-cov` subtree reinforced this, ranging from none in some packages to several hundred in one case, which left participants overwhelmed and reluctant to review all findings. Because these reactions are tied to the artificial conditions of the study, they are relevant for interpreting the other themes.

Bugs The *Bugs* theme collects defects in the tool that surfaced while participants solved the tasks and contains seven codes. The most consequential defect is the missing validation of the conclude form. Confirming the form without entering a license produced no visible result, which left participants unsure whether they had finished and later forced

5.4. Results

them to conclude the package a second time. A second recurring problem was flickering tooltips, which both startled participants and prevented them from clicking the links inside a tooltip. The remaining codes record a could-not-load-file error shown for a finding, a download link for the open-source brochure that does not work, and the hidden licenses not using the same filtering logic as the shown findings, leading to them showing up as both hidden and to-be-reviewed.

Sentiments The *Sentiments* theme records the participants' attitudes towards the tool and divides into neutral, positive, and negative reactions across 35 codes. The five neutral codes contain remarks such as the package metadata being unexciting and working with license findings requiring more practice. Positive reactions are the most numerous and group into praise for the dependency graph, praise for the navigation, and a number of individual remarks. The dependency graph was repeatedly appreciated for conveying the complexity of the reviewed project at a glance and for providing good orientation. Navigating through findings and dependencies was perceived as practical, and the tree-like structure as well as the working browser navigation were called out positively. Individually praised aspects include the license diff, the tooltips, the validation of entered SPDX identifiers, the explanation of declared and detected licenses, and the ability to edit a conclusion. Negative reactions concern mostly layout and missing information. It was not obvious that the lower two panels belong to the selected package, the delete button in the conclusion panel was unclear, and the reason of a finding's conclusion is missing from the list of reviewed findings. Participants also disliked that the LICENSE file is not shown directly and instead requires a detour to view.

Wishes The *Wishes* theme gathers the features and changes participants asked for and splits into additional context, explanations and quality-of-life improvements across 30 codes. Most wishes ask for additional context that the tool currently withholds. Participants wanted to inspect hidden findings, to see the full name or text behind a license identifier, to open the relevant LICENSE file, and to know which SPDX identifiers exist. One request was to highlight copyleft licenses that affect the licenses they are combined with. The explanation wishes mirror the terminology gaps reported under Insecurities, most prominently a request to explain NOASSERTION and to attach tooltips to the action buttons. The quality-of-life wishes target the efficiency of the review. By far the most frequent single request was to navigate the dependency tree directly through the nodes of the graph, followed by making the graph zoomable. Participants further asked for ways to record decisions in bulk, to prefill a conclusion with the declared license, and to jump to the next package immediately after a conclusion. Some suggested reordering the panels so that their arrangement follows the order of the review process.

5. Evaluation

5.4.3 Process Fit

The three Likert statements following the SUS address how the tool relates to the participants' license analysis process rather than its general usability. They ask whether the tool fits into the participant's existing license analysis workflow, whether it would reduce the effort compared to their normal approach, and whether the participant felt confident that the information provided by the tool was accurate enough for compliance decisions. Figure 5.5 shows the distribution of answers for each statement.

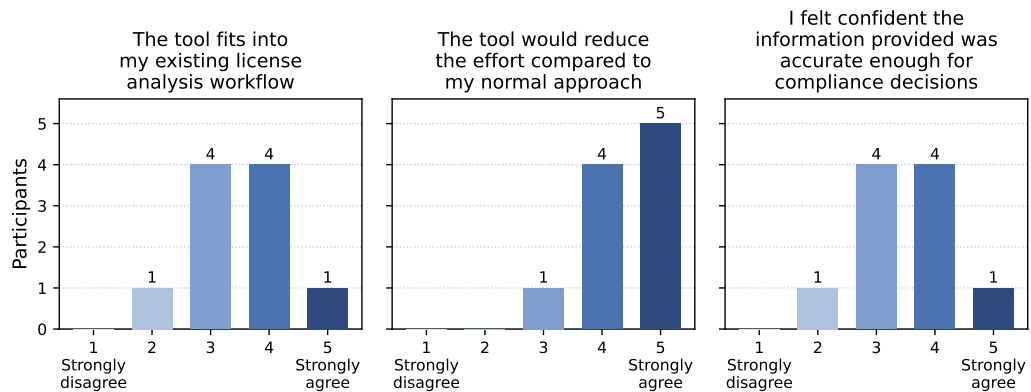


Figure 5.5. Distributions of answers to the three statements on process fit. Participants widely agree that the tool would reduce the effort compared to their normal approach. Answers on workflow fit and on confidence in the accuracy of the provided information are mostly positive, but a notable share of participants answers neutrally.

The clearest result concerns the reduction of effort. Nine out of ten participants agree or strongly agree that the tool would reduce the effort compared to their normal approach, while the remaining participant answers neutrally and no participant disagrees. This suggests that the participants expect the tool to make the review of scan results less effortful, although this remains a self-reported expectation after a single session and not a measured reduction. The answers regarding workflow fit are positive but less pronounced. Five participants agree or strongly agree that the tool fits into their existing license analysis workflow, four answer neutrally and one disagrees. The statement on confidence in the accuracy of the provided information receives similarly reserved answers. Half of the participants agree or strongly agree, four answer neutrally and one disagrees.

We do not report the answers to the four free-text questions separately, because the notable information they contain is already covered by the thematic analysis of the think-aloud protocols. The complete answers are available at [Haß 2026].

Discussion

The preceding chapters described the elicitation of the requirements, the design, and implementation of the prototype, and its evaluation. This chapter steps back from this description and interprets the results. We revisit the central design decisions, in particular the choice of a human-in-the-loop review process over automated classification of license findings, and reflect on the implementation, including a rejected user interface variant and the deliberately limited scope of some features. We then interpret two notable patterns in the evaluation results and draw lessons from the study design, before examining the threats to the validity of the two empirical methods, the expert interviews and the usability study, and the measures taken to mitigate them.

6.1 Design

This section revisits three design decisions of the prototype: the choice of a manual review process over automated classification, the integration with the ORT through its scan result file, and the deferral of decision traceability.

The prototype implements license review as a manual, human-in-the-loop process. The obvious alternative is the automated classification of license findings, for example by means of natural language processing. A certain uncertainty is inherent to license analysis regardless of the approach as license statements are natural language, and interpreting them is not exact, whether the interpretation is performed by a scanner, by an automated classification method, or by a human reviewer. The scanner output makes this uncertainty explicit through the confidence score attached to each finding. Since no approach can eliminate this uncertainty, the deciding question is who carries the responsibility for a compliance decision. Automated classification leaves this question open. A manual review keeps the responsibility with the reviewer, who takes every decision explicitly. For this reason, automated classification was rejected in favor of a human-in-the-loop design.

The second decision concerns the integration with the ORT: the tool reads the scan result file directly. During development, we discovered that this file format is an internal interface of the ORT and therefore offers no stability guarantees across versions [ORT Project 2023]. A cleaner integration would have been a dedicated reporter that exposes exactly the information required by the review tool. For a prototype, however, using the existing file format was the pragmatic choice: the format is readily available, and the implementation

6. Discussion

effort could be focused on the review tool itself instead of an additional reporter.

The third decision was to defer the decision traceability defined by US-20. The development focused on the first review cycle of a project, and accordingly the participants of the usability study reviewed a project that had not been reviewed before. In addition, the available time was spent on improving the general review process and the usability of the tool, which matter most in this first cycle, following the principle that few well-designed features are preferable to many insufficiently thought-through ones. Two ways to implement decision traceability in the future are conceivable. The first is to store author and timestamp explicitly for each decision, either by extending the curation data model or by storing this information in a separate file specific to the tool. This would require some way for the user to identify themselves inside the tool and, if required by organization policy, the tool would need to authenticate the user's identity. The second is to use a version control system to track changes to the curation file and to display the commit author and timestamp for each decision. This approach requires the user to keep the curation file under version control and to commit their changes, which is not always the case, and it may attribute decisions to the wrong author when the curation file is edited outside the tool.

Taken together, these decisions reflect two priorities: The first is keeping the responsibility for compliance decisions with a human reviewer, which motivated the manual review process. The second is concentrating the limited development capacity on the core review process, accepting pragmatic shortcuts like using an internal interface and deferring the decision traceability where the alternative would have shifted effort away from it.

6.2 Implementation

This section reflects on the implementation from five angles: the development process, a user interface variant that was explored and rejected, the choice of the frontend technology, the runtime behavior of the tool, and the scope limitations of the implemented features.

In retrospect, an agile development process with iterative user feedback would have been preferable. Obvious shortcomings like the missing terminology explanations and bugs like the missing validation of the Conclude license form could have been fixed before conducting a final evaluation. Design questions such as the guided review workflow discussed below could have been settled by letting users test variants early instead of by speculation during development. This was hindered by the limited availability of test participants, whose time had to be reserved for the usability study.

One such design question concerns a user interface variant that guides the user more strongly by asking whether the declared license can be trusted before showing the license findings. This guidance works in general, but the resulting visibility logic for the findings becomes convoluted. The findings are hidden while the question is unanswered, shown after the question is answered with no, and hidden after the question is answered with yes. For a concluded package the findings are hidden as well, but they must remain reviewable

without forcing the user to delete the existing conclusion first. This variant could be revisited in the future, but it requires a thorough consideration of the user experience and the review workflow.

Building the frontend with Vue.js is a deliberate deviation from the boundary condition of implementing the tool primarily in Python. Python is generally not used for frontend development, so a purely Python-based frontend with minimal JavaScript would have resulted in a multipage application that feels more like a website than an application. A JavaScript-heavy single-page application instead changes views without navigation delays, because navigating does not reload the page, at the minor upfront cost of a longer initial loading time [Jain 2022]. We judged this application-like experience to be worth the deviation, because the review process involves frequent navigation between many packages. To contain this deviation as much as possible, the API client code is generated from the OpenAPI specification that the Python backend provides, so the frontend does not need to implement any API calls manually.

Regarding performance, parsing the scan result file takes several seconds for the test project with 25 packages, whereas analyzing and scanning the same project with the ORT takes several hours. The review tool therefore adds a negligible amount of time to the license analysis process. Notably, the long execution time of the ORT is acceptable in the first place because analysis and scan can run in the background without user interaction. Since the scan result is loaded before the review starts, this loading may equally run without a user actively waiting for it, although this is unlikely in the current form of the tool, which is intended to run on the user’s machine. No significant effort was therefore spent on reducing the startup time, but this optimization remains possible in the future.

Finally, some features were implemented with a deliberately limited scope. Marking items as not relevant (US-17) was implemented only for license findings, so direct dependencies cannot be excluded for being development dependencies. This is partly because the ORT supports a curation for excluding findings, and partly because the focus lay on reviewing the transitive dependencies. For the same reason, the files of the project itself are neither scanned nor reviewed in the tool. In addition, the ORT downloader has to be run manually after the analyzer before the source context of a finding can be shown in the tool; integrating the downloader into the tool is future work.

In summary, the implementation reflects the same prioritization as the design: effort was concentrated on the core review workflow, while process, performance, and feature scope were kept as simple as the prototype allowed.

6.3 Evaluation

This section first interprets two notable patterns in the evaluation results — the neutral answers to the workflow fit statement and the reservations visible in the confidence results — and then reflects on the study design itself.

One possible explanation for the large share of neutral answers to the workflow fit

6. Discussion

statement lies in the prior experience of the participants. Most participants report a low level of experience with using software in the license analysis process (see Figure 5.3), so several participants may not have an established workflow that the tool could fit into or conflict with.

The reservation visible in the confidence results is consistent with the *Insecurities* theme of the think-aloud analysis. Participants missed indications of where displayed information originates, encountered license identifiers they did not recognize, and repeatedly expressed a reluctance to decide on the information shown in the tool alone, preferring to consult the source repository before trusting a declared license. Improving the transparency of information sources could therefore be a direct lever to increase this confidence. Concretely, indicating that the declared license originates from the analyzer component of the ORT while the detected licenses come from its scanner component could reduce this confusion.

Beyond the results themselves, the study design offers two lessons. First, the evaluation setup was deliberately open and included little guidance on how to use the tool. This openness was intended to observe how participants approach the tool without instruction, but combined with the lack of a built-in manual, it left participants struggling to make sense of why they were using the tool. Second, participants did not always know when a task was finished, and consequently some did not conclude the findings and packages. A task description that includes an indicator of task completion could have prevented this [Boren and Ramey 2000]. The handling of the time limits — tasks were allowed to run longer than the defined limit as long as a participant was still making useful progress — is discussed as a threat to validity in the next section.

6.4 Threats to Validity

This section discusses the threats to the validity of the two empirical methods, the expert interviews and the usability study, grouped into construct, internal, and external validity as well as reliability, following the classification of Runeson and Höst [2009].

A threat to construct validity is that participants of the usability study did not always recognize when a task was complete, so the completion of findings and packages measures both the usability of the tool and the clarity of the task description (see Section 6.3).

Both the interviews and the usability study were conducted by an inexperienced interviewer which may have influenced how questions were asked and how sessions were moderated. To counteract this, advice on conducting interviews and on the interview guidelines was obtained from domain experts within the DLR. The pretest of the usability study served both to check the study design and to gain first moderation experience. Furthermore, the time limits of the tasks were not strictly enforced: the combined task time was sometimes allowed to run up to ten minutes over the defined total of 30 minutes as long as a participant was still making useful progress. This makes completion-related observations less comparable across participants. Since participants knew they were evaluating a tool developed by the study conductor, questionnaire answers may be biased toward favorable

or neutral ratings.

All participants were recruited from within the DLR, and most reported little experience with software support in the license analysis process. The results may therefore not transfer to other organizations or to more experienced compliance specialists. In addition, the usability study covered a single test project of moderate size; how the tool performs for substantially larger projects was not evaluated.

The thematic analysis of both the interviews and the think-aloud data was carried out by a single person, so its outcome may be subjective. To mitigate this, a thematic analysis could be performed by multiple researchers independently to cancel out individual bias [Nowell et al. 2017]. After producing a first set of codes, categories, and themes, the researchers would compare their results and adjust their analysis accordingly. This was not possible because the present work is a master's thesis and not a group project.

In summary, the discussion confirms the central design choice of the prototype while qualifying its results. The human-in-the-loop review process remains justified by the responsibility argument, and the evaluation shows that the tool integrates into the license analysis process at negligible additional cost. At the same time, the results must be read within their limits: they stem from a small group of participants from a single organization, most of them inexperienced in license analysis, reviewing a single project of moderate size. The threats to validity discussed above were mitigated where possible, but not eliminated; in particular, the single-researcher thematic analysis and the participants' relation to the study conductor remain inherent to the setting of a master's thesis. Within these limits, the findings support the conclusions drawn in the following chapter, and the open points identified here reappear there as future work.

Conclusion

This thesis set out to take a first step toward reducing the burden that license compliance places on the scarce RSEs of a research organization, by improving their efficiency in reviewing license scanner output. Starting from interviews with experts at the DLR, we derived what information RSEs need when reviewing scanner output and turned these needs into concrete requirements. We surveyed the landscape of existing license tooling, identified a gap in the way scanner output is presented for human review, and designed and implemented the *License Scanner Review Tool* to close it. Finally, we evaluated the prototype in a usability study to assess whether it makes the review process less effortful.

7.1 Revisiting the Goals

We now revisit the three goals stated in the introduction and summarize how each was addressed.

G1 – Identify user needs: We conducted four semi-structured interviews with practitioners at the DLR and analyzed the transcripts using thematic analysis. The analysis yielded six themes that describe the motivation for license analysis, the current and envisioned license analysis process, its users, tooling, and context. From these findings we formalized 20 user stories (see Table 3.2) that are the requirements for a system that supports RSEs in reviewing license scanner output. They provide the empirical basis for the tool design.

G2 – Design and implement a prototype: Given the requirements, we designed and implemented the *License Scanner Review Tool*, a web-based review layer on top of the ORT pipeline. We based the prototype on the ORT with ScanCode as its scanner backend, which we identified as the most suitable foundation after comparing seven open-source license tools (see Table 4.1). The central gap we identified is that the raw output of license scanners is not presented in a way that supports efficient review and decision-making. The *License Scanner Review Tool* aggregates and filters the analyzer and scanner output and enriches each package with context, such as the file context around a finding, a diff against canonical license texts, a dependency graph, and popularity data. License findings are then presented to a reviewer one at a time. The reviewer’s decisions are written back as ORT curations that the next pipeline run can apply automatically. This

7. Conclusion

forms an interactive feedback loop, while the rest of the pipeline remains automated and CI-friendly.

G3 – Evaluate the prototype: We evaluated the prototype in a think-aloud usability study with ten participants. The tool achieved a mean SUS score of 69.5, which is close to “good” on the adjective rating scale by Bangor et al. [2009]. Nine out of ten participants agreed that it would reduce their effort compared to their normal approach. The accompanying thematic analysis of the think-aloud protocols also identified concrete usability problems that point toward specific improvements. These include unrecognized license identifiers, unclear license compatibility, missing indication of information sources, and unexplained terminology.

Taken together, these results indicate that a dedicated review tool can improve the research software license analysis process. Participants perceived the review as less effortful and largely appreciated the context the tool surfaces. At the same time, the study points to usability refinements that are still needed before the tool is ready for everyday use. The results should also be read in light of the threats to validity discussed in Section 6.4, in particular the small sample of ten participants and the single test project.

7.2 Future Work

The evaluation and the design decisions made during this thesis open up several directions for future work.

The most immediate direction is to address the usability problems uncovered in the evaluation. The open study setup showed that users struggle without onboarding, so a written manual or a guided tour would help them understand the tool’s purpose and workflow. The reason codes that the ORT defines for findings caused confusion, because they do not cover the common case of a finding simply being correct. They should either be explained more clearly or hidden from the user. Beyond these, the concrete bugs and points of insecurity reported by participants, such as the missing validation on the conclude form, should be fixed.

A second direction concerns extending the tool’s functionality. Decision traceability was deliberately left out of the first review cycle. It could be implemented by recording the author and timestamp of each decision, or by deriving this information from the VCS history of the curation files. Integrating the ORT downloader into the tool would remove a manual step from the workflow. Clearly indicating the origin of displayed information could also raise reviewer confidence, for example by showing that the declared license comes from the analyzer while detected licenses come from the scanner (see Section 6.4).

A third direction is broader coverage and further validation. Support for more programming languages and package ecosystems is largely a matter of testing and handling ecosystem-specific quirks, such as the way monorepos are structured in the JavaScript ecosystem. The tool should also be tested for recurring reviews of a project that changes

7.2. Future Work

over time. This is where continuous integration of license analysis and features such as decision traceability become most valuable. A comparative usability study against an established tool such as FOSSology would put the measured scores into perspective. Finally, it remains an open question whether an automated classification of findings using natural language processing could match the accuracy of manual review. This was deliberately rejected here for reasons of responsibility (see Section 6.1), but the accuracy question remains open and worth investigating. Another way to integrate natural language processing would be to automatically suggest a curation based on the context of a finding, which the reviewer could then accept or reject.

Acronyms

API	Application Programming Interface
BOM	Bill of Materials
CLI	Command Line Interface
CI	Continuous Integration
CRUD	Create, Read, Update, and Delete
DLR	German Aerospace Center (Deutsches Zentrum für Luft- und Raumfahrt)
FOSS	Free and Open Source Software
GDPR	General Data Protection Regulation
ORT	OSS Review Toolkit
PURL	Package Uniform Resource Locator
PyPI	Python Package Index
RSE	Research Software Engineer
SPDX	System Package Data Exchange
SUS	System Usability Scale
UI	User Interface
VCS	Version Control System

Usage of LLM-gen-AI

Both this thesis and the accompanying prototype were created with the assistance of generative artificial intelligence systems backed by large language models (LLM-gen-AI). The Software Freedom Conservancy has published recommendations for contributors who use such systems in their work on FOSS [Software Freedom Conservancy 2026]. Although these recommendations address contributions to existing FOSS projects, they apply partially to this work as well, since the prototype is published under a FOSS license. This appendix is motivated by the recommendation to fully disclose how and when a LLM-gen-AI system assisted in the creation of a contribution. The most relevant recommendation for how we used these systems is the one to invest substantial time in reviewing assisted or generated contributions before submission, which we followed for all generated output.

For the implementation of the prototype, we used GitHub Copilot for code completion and next-line suggestions, and the Claude models Sonnet 4.6, Opus 4.7, and Opus 4.8, depending on the task, for code generation and refactoring through the Claude Code interface. Our workflow generally followed a consistent pattern: we prompted the model with, for example, a user story together with a description of the desired implementation details, refined the plan proposed by the model in follow-up prompts until it was satisfactory, and only then let the model generate the code for the plan. We then reviewed the generated code and dismissed or refined it as needed, so that, in line with the recommendations, no generated code entered the code base unreviewed. In line with the recommendation not to overuse LLM-gen-AI, a significant part of the implementation was still coded manually, and we used assistance only where we considered it useful. This shifted our time from writing code to planning and reviewing it, which we hope had a positive effect on the quality of the code and the usability of the prototype. Beyond code generation, we used the models for debugging assistance.

For writing this thesis, we used LLM-gen-AI more conservatively. We used the same models as for coding, as well as Claude Fable 5, for language improvements, grammar and spelling checks, rephrasing suggestions, and sanity checks of the document structure. Crucially, we did not use these systems to generate content; their role was to improve the quality and readability of text we had written.

Bibliography

- [AboutCode 2026] AboutCode. *ScanCode Toolkit*. July 2026. URL: <https://github.com/aboutcode-org/scancode-toolkit> (visited on 07/07/2026). (Cited on page 11)
- [Bangor et al. 2009] A. Bangor, P. Kortum, and J. Miller. Determining what individual SUS scores mean: adding an adjective rating scale. *Journal of Usability Studies* 4.3 (May 2009), pages 114–123. URL: <https://dl.acm.org/doi/10.5555/2835587.2835589>. (Cited on pages 52, 53, 66)
- [Boren and Ramey 2000] T. Boren and J. Ramey. Thinking aloud: reconciling theory and practice. *IEEE Transactions on Professional Communication* 43.3 (Sept. 2000), pages 261–278. DOI: 10.1109/47.867942. (Cited on pages 48, 50, 62)
- [Braun and Clarke 2006] V. Braun and V. Clarke. Using thematic analysis in psychology. *Qualitative Research in Psychology* 3.2 (2006), pages 77–101. DOI: 10.1191/1478088706qp0630a. (Cited on page 16)
- [Brochhagen et al. 2022] M. Brochhagen, C. Haupt, T. Jäger, V. Müller, T. Schlauch, and K. Stumvoll. *Nutzung von Open-Source-Software im DLR*. 2022. URL: <https://elib.dlr.de/193607/> (visited on 01/25/2026). (Cited on pages 7, 22, 39)
- [Brooke 1996] J. Brooke. “SUS: A ‘Quick and Dirty’ Usability Scale”. In: *Usability Evaluation In Industry*. CRC Press, 1996. (Cited on page 50)
- [Caldwell et al. 2008] B. Caldwell, M. Cooper, L. G. Reid, G. Vanderheiden, W. Chisholm, J. Slatin, and J. White. Web content accessibility guidelines (WCAG) 2.0. *WWW Consortium (W3C)* 290.1-34 (2008), pages 5–12. (Cited on page 45)
- [Cohn 2004] M. Cohn. *User stories applied: for agile software development*. USA: Addison Wesley Longman Publishing Co., Inc., 2004. (Cited on page 26)
- [Czerniak et al. 2025] A. Czerniak, A. Ehrenhofer, B. Fritzsche, M. Funk, F. Goth, R. Hähnle, C. Haupt, M. Konersmann, J. Linxweiler, F. Löffler, A. Lüpkes, S. Nielebock, B. Rumpe, I. Schieferdecker, T. Schlauch, R. Speck, A. Struck, J. P. Thiele, M. Tichy, and I. Ulusoy. *GI- und DE-RSE Muster-Leitlinie zur effizienten Entwicklung von Forschungssoftware*. 2025. DOI: 10.18420/2025-gi_de-rse. (Cited on page 2)
- [DLR 2024] DLR. *Zahlen und Fakten*. 2024. URL: <https://www.dlr.de/de/das-dlr/ueber-uns/das-dlr-in-zahlen/personal> (visited on 07/07/2026). (Cited on page 7)
- [Felderer et al. 2025] M. Felderer, M. Goedicke, L. Grunske, W. Hasselbring, A.-L. Lamprecht, and B. Rumpe. Investigating Research Software Engineering: Toward RSE Research. *Commun. ACM* 68.2 (Jan. 2025), pages 20–23. DOI: 10.1145/3685265. (Cited on page 2)

Bibliography

- [Franz et al. 2016] M. Franz, C. T. Lopes, G. Huck, Y. Dong, O. Sumer, and G. D. Bader. Cytoscape.js: a graph theory library for visualisation and analysis. *Bioinformatics* 32.2 (Jan. 2016), pages 309–311. DOI: 10.1093/bioinformatics/btv557. (Cited on page 36)
- [Geisen and Bergstrom 2017] E. Geisen and J. R. Bergstrom. *Usability testing for survey research*. Morgan Kaufmann, 2017. (Cited on page 50)
- [GitHub 2026] GitHub. *Rate limits for the REST API*. URL: <https://docs.github.com/en/rest/using-the-rest-api/rate-limits-for-the-rest-api?apiVersion=2026-03-10> (visited on 07/07/2026). (Cited on page 44)
- [Gobeille 2008] R. Gobeille. The FOSSology project. In: *Proceedings of the 2008 international working conference on Mining software repositories*. MSR '08. New York, NY, USA: Association for Computing Machinery, May 2008, pages 47–50. DOI: 10.1145/1370750.1370763. (Cited on page 12)
- [Haß 2026] L. Haß. *Improving the research software review process of license scanner output - artifacts*. Zenodo, June 2026. DOI: 10.5281/zenodo.20596507. (Cited on pages 13, 20, 36, 47, 48, 54, 58)
- [Hasselbring et al. 2020] W. Hasselbring, L. Carr, S. Hettrick, H. Packer, and T. Tiropanis. Open Source Research Software. *Computer* 53.8 (Aug. 2020), pages 84–88. DOI: 10.1109/MC.2020.2998235. (Cited on page 2)
- [Instone 2002] K. Instone. Location, path & attribute breadcrumbs. *Proceedings of the 3rd Annual Information Architecture Summit* (Jan. 2002), pages 16–17. (Cited on page 36)
- [International Organization for Standardization 2021] International Organization for Standardization. *ISO/IEC 5962:2021 Information technology — SPDX Specification V2.2.1*. 2021. URL: <https://www.iso.org/standard/81870.html> (visited on 07/07/2026). (Cited on page 8)
- [Interoperable Europe 2026] Interoperable Europe. *Compatibility Checker | Interoperable Europe Portal*. URL: <https://interoperable-europe.ec.europa.eu/collection/eupl/solution/licensing-assistant/compatibility-checker> (visited on 07/07/2026). (Cited on page 39)
- [Isaksen et al. 2017] H. Isaksen, M. Iversen, J. Kaasbøll, and C. Kanjo. Methods for Evaluation of Tooltips. In: *Human-Computer Interaction. User Interface Design, Development and Multimodality*. Edited by M. Kurosu. Cham: Springer International Publishing, 2017, pages 297–312. DOI: 10.1007/978-3-319-58071-5_23. (Cited on page 43)
- [Jain 2022] V. Jain. A comparative analysis of single page applications (SPAs) and multi page applications (MPAs). *International Journal of Core Engineering & Management* 7.4 (2022), pages 271–277. (Cited on page 61)
- [Johanson and Hasselbring 2018] A. Johanson and W. Hasselbring. Software Engineering for Computational Science: Past, Present, Future. *Computing in Science & Engineering* 20.2 (Mar. 2018), pages 90–109. DOI: 10.1109/MCSE.2018.021651343. (Cited on page 2)
- [Jung et al. 2022] R. Jung, S. Gundlach, and W. Hasselbring. Thematic domain analysis for ocean modeling. *Environmental Modelling & Software* 150 (Apr. 2022), page 105323. DOI: 10.1016/j.envsoft.2022.105323. (Cited on page 16)

- [Li et al. 2025] B. Li, C. Liu, L. Fan, S. Chen, Z. Zhang, and Z. Liu. Open Source, Hidden Costs: A Systematic Literature Review on OSS License Management. *IEEE Transactions on Software Engineering* 51.9 (Sept. 2025), pages 2432–2454. DOI: 10.1109/TSE.2025.3586411. (Cited on page 7)
- [Licensee 2026] Licensee. *Licensee*. July 2026. URL: <https://github.com/licensee/licensee> (visited on 07/07/2026). (Cited on page 32)
- [Likert 1932] R. Likert. *A technique for the measurement of attitudes*. Open Library ID: OL50383141M. New York: Archives of Psychology, Columbia University, 1932. (Cited on page 50)
- [Myers 1986] E. W. Myers. An O(ND) difference algorithm and its variations. *Algorithmica* 1.1-4 (Nov. 1986), pages 251–266. DOI: 10.1007/BF01840446. (Cited on page 42)
- [Nowell et al. 2017] L. S. Nowell, J. M. Norris, D. E. White, and N. J. Moules. Thematic analysis: striving to meet the trustworthiness criteria. *International Journal of Qualitative Methods* 16.1 (2017), page 1609406917733847. DOI: 10.1177/1609406917733847. (Cited on page 63)
- [Nugroho et al. 2020] Y. S. Nugroho, H. Hata, and K. Matsumoto. How different are different diff algorithms in Git? *Empirical Software Engineering* 25.1 (Jan. 2020), pages 790–823. DOI: 10.1007/s10664-019-09772-z. (Cited on page 42)
- [ORT Project 2023] ORT Project. *Ort dependency_graphs*. July 2023. URL: <https://github.com/oss-review-toolkit/ort/issues/7236> (visited on 07/07/2026). (Cited on page 59)
- [ORT Project 2026a] ORT Project. *Find an alternative to Python-Inspector*. Feb. 2026. URL: <https://github.com/oss-review-toolkit/ort/issues/11467> (visited on 07/07/2026). (Cited on page 42)
- [ORT Project 2026b] ORT Project. *License Handling | OSS Review Toolkit*. URL: <https://oss-review-toolkit.github.io/ort/docs/guides/license-handling> (visited on 07/07/2026). (Cited on page 9)
- [ORT Project 2026c] ORT Project. *OSS Review Toolkit*. URL: <https://oss-review-toolkit.github.io/ort/> (visited on 07/07/2026). (Cited on page 10)
- [ORT Project 2026d] ORT Project. *Scanner | OSS Review Toolkit*. URL: <https://oss-review-toolkit.github.io/ort/docs/tools/scanner> (visited on 07/07/2026). (Cited on page 10)
- [Python Software Foundation 2026] Python Software Foundation. *PyPI Download Stats*. URL: <https://pypistats.org/api/> (visited on 07/07/2026). (Cited on page 44)
- [Radford et al. 2022] A. Radford, J. W. Kim, T. Xu, G. Brockman, C. McLeavey, and I. Sutskever. Robust speech recognition via large-scale weak supervision (Dec. 2022). DOI: 10.48550/ARXIV.2212.04356. (Cited on page 15)
- [Runeson and Höst 2009] P. Runeson and M. Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering* 14.2 (Apr. 2009), pages 131–164. DOI: 10.1007/s10664-008-9102-8. (Cited on page 62)

Bibliography

- [Schreiber et al. 2014] A. Schreiber, R. Galoppini, M. Meinel, and T. Schlauch. An Open Source Software Directory for Aeronautics and Space. In: *Proceedings of The International Symposium on Open Collaboration*. OpenSym '14. New York, NY, USA: Association for Computing Machinery, Aug. 2014, pages 1–7. DOI: 10.1145/2641580.2641630. (Cited on page 7)
- [Schreiber and Haupt 2017] A. Schreiber and C. Haupt. Knowledge Exchange about Open Source at the German Aerospace Center. In: *Proceedings of the 13th International Symposium on Open Collaboration, OpenSym 2017*. Galway, Ireland: ACM New York, NY, USA, Aug. 2017. DOI: 10.1145/3125433.3125470. (Cited on pages 2, 3, 7)
- [Software Freedom Conservancy 2026] Software Freedom Conservancy. *Recommendations when using llm-backed generative ai systems for foss contributions*. June 2026. URL: <https://sfconservancy.org/llm-gen-ai/llm-backed-generative-ai-recommendations.html> (visited on 07/08/2026). (Cited on page 71)
- [SPDX Project 2024] SPDX Project. *SPDX Specification 3.0.1*. The Linux Foundation, Dec. 2024. URL: <https://spdx.github.io/spdx-spec/v3.0.1/> (visited on 07/07/2026). (Cited on pages 8, 9, 41)
- [SPDX Project 2026] SPDX Project. *SPDX License List*. The Linux Foundation, Feb. 2026. URL: <https://spdx.org/licenses/> (visited on 07/07/2026). (Cited on page 9)