The Journal of Supercomputing
https://doi.org/10.1007/s11227-024-06069-x

Analyzing FOSS license usage in publicly available software at scale via the SWH-analytics framework

Alessia Antelmi[1,3] · Massimo Torquati[2,3] · Giacomo Corridori[5] · Daniele Gregori[4] · Francesco Polzella[5] · Gianmarco Spinatelli[5] · Marco Aldinucci[1,3]

Accepted: 12 March 2024

Abstract
The Software Heritage (SWH) dataset represents an invaluable source of open-source code as it aims to collect, preserve, and share all publicly available software in source code form ever produced by humankind. Although designed to archive deduplicated small files thanks to the use of a Merkle tree as the underlying data structure, querying the SWH dataset presents challenges due to the nature of these structures, which organize content based on hash values rather than any locality principle. The magnitude of the repository, coupled with the resource-intensive nature of the download process, highlights the need for specialized infrastructure and computational resources to effectively handle and study the extensive dataset housed within SWH. Currently, there is a lack of infrastructures specifically tailored for running analytics on the SWH dataset, leaving users to handle these issues manually. To address these challenges, we implemented the SWH-Analytics (SWHA) framework, a development environment that transparently runs custom analytic applications on publicly available software data preserved over time by SWH. Specifically, this work shows how SWHA can be effectively exploited to study usage patterns of free and open-source software licenses, highlighting the need to improve license literacy among developers.

Keywords Software Heritage · Free and open-source software · License conflicts · License management · Large-scale analytics

1 Introduction

Over the past two decades, open-source software (OSS) has undergone remarkable development, now enjoying widespread adoption. What commenced as a grassroots movement marked by the introduction of the first free OSS operating system has since evolved into a pervasive trend within the developer community [1, 2]. This

Extended author information available on the last page of the article momentum, in turn, has led to the widespread integration of open-source solutions by enterprises worldwide, ultimately capturing the attention of major players within the software industry, as exemplified by IBM's acquisition of Red Hat [3]. As highlighted in the 2022 GitHub report [4], open source serves as the cornerstone of over 90% of the world's software infrastructure. According to the same report, the year 2022 alone witnessed the inception of an astounding 52 million new (possibly open-source) projects on the GitHub platform, underscoring the thriving ecosystem. Furthermore, according to estimates from the European Commission, the adoption of open-source software is contributing to annual savings of approximately 114 billion euros in direct development costs, thereby significantly bolstering the European economy [5].

At its essence, OSS refers to software whose source code is made available to the public, allowing for viewing, modification, and distribution by anyone without any cost [6]. To be effectively considered open-source, the software must be accompanied by a license that makes its source code legally available to end-users. In this context, the Software Heritage (SWH) initiative represents a valuable source as it aims to archive, preserve, and make accessible all software publicly available in source code form ever produced by humankind [7]. SWH enables a whole series of analyses to gain precious insights into the evolution of the open-source community and its practices over time. By leveraging SWH data, researchers can delve into how developers contribute to OSS and explore the distinctive aspects of public code contributions, thus enabling a comprehensive exploration of the dynamics and nuances inherent in the landscape of open-source development. The SWH dataset is experiencing rapid growth, reaching close to 1PB of archived software source code files and boasting a metadata graph of nearly 20TB as of July 2023, with a monthly expansion rate of several TB. While designed to archive deduplicated small files (with an average size of less than 4kB) thanks to the use of a Merkle tree as the underlying data structure, querying the SWH dataset presents challenges due to the nature of these structures, which organize content based on hash values rather than any locality principle. This storing approach makes efficient processing in Big Data MapReduce frameworks (e.g., Spark) or AI training/inference systems challenging. Traversing the 20TB metadata hash tree and navigating across 1PB of storage objects without spatial locality hampers the efficiency of iterative operations, as files from the same directory may not be stored in contiguous memory areas. Consequently, SWH queries, which can request entire projects or specific elements (e.g., all README files), may require significant time to retrieve the desired data. For this reason, SWH may not be directly suitable for systems prioritizing efficiency in data retrieval.

Motivation. The vast scale of the SWH repository poses a formidable challenge for analysis using conventional tools. Attempting to download content from SWH not only demands a significant investment of time but also necessitates ample storage resources. Therefore, the prospect of downloading the entire dataset onto a proprietary machine and subsequently analyzing it on a personal laptop becomes impractical and unfeasible. The magnitude of the repository, coupled with the resource-intensive nature of the download process, highlights the need for specialized infrastructure and computational resources to effectively handle and study the extensive dataset housed within SWH. Currently, there is a lack of infrastructures specifically tailored for running analytics on the SWH dataset, leaving users to handle the various challenges mentioned manually. In this context, the Software Heritage Analytics framework (SWHA) comes into play. The first design goal of SWHA is to enhance the analytics process by providing a structured platform, thereby relieving users from the explicit management of the challenges mentioned earlier. The second objective is to facilitate the seamless updating of existing analyses, especially in monitoring the evolution of specific aspects. In scenarios where users need to track changes or updates, SWHA allows them to download and process only the new data effortlessly. This transparency in handling updates is made possible by the architecture of SWHA, allowing users to efficiently build upon their analyses without unnecessary duplication of efforts or extensive manual intervention.

Contribution. This article extends our previous work [8], where we introduced the SWHA framework, the first specialized development and runtime environment designed to facilitate the analysis of software archives preserved over time by SWH. SWHA is a free and open-source project available on GitHub[1], developed within the context of the ADMIRE European project[2]. In this work, we focus on presenting the application of SWHA in analyzing software archives stored by SWH. Specifically, we illustrate SWHA's functionalities by delving into a practical scenario: the examination of free and OSS (FOSS) licenses within publicly available software artifacts. Through this exploration, we offer insights into the efficacy and scalability of SWHA, showcasing its performance in real-world contexts. Our novel contributions can be summarized as follows.

- To illustrate the practical utility of SWHA, we present an application dedicated to examining FOSS license usage within publicly available software artifacts. Specifically, we considered publicly available GitHub repositories indexed by SWH. Focusing on software licenses is particularly significant, given that one of the defining characteristics of open-source software is the presence of a license that dictates the legal accessibility of its source code to end-users.
- We perform an in-depth analysis to assess how licenses are expressed in practice, quantifying the prevalence of multi-licensed projects and license conflicts detected in the most used GitHub repositories.
- We evaluate the performance of our license analytics application and the overall scalability performance of SWHA.

In line with the terminology used by SWH, when we use the term *(software) project* in this article, we are referring to a software artifact in its source code form. Given the complex data model underlying SWH, we take care to specify whether we are referring to SWH or GitHub projects throughout the article when not clear from the context. In cases where we are discussing GitHub projects, we also use the term *repository* interchangeably.

[1] https://github.com/alpha-unito/Software-Heritage-Analytics

[2] https://admire-eurohpc.eu

The remainder of the paper is organized as follows. Section 2 details the architecture of SWHA, and introduces the SWH initiative on which it is based. Section 3 overviews related works concerning the analysis of license usage. Section 4 presents the analytic application built on top of SWH to analyze FOSS license usage in publicly available software artifacts. Section 5 discusses the insights obtained, the application's performance and framework's scalability, and the limitations of our work. Section 6 concludes this work.

2 Background

In this section, we provide an overview of the Software Heritage dataset by delineating its main characteristics to provide the reader with the essential knowledge to understand specific architectural details of SWHA. We then describe the Software Heritage Analytics framework by detailing its components.

2.1 Software Heritage

Software Heritage [7, 9, 10] is a globally recognized nonprofit initiative dedicated to archiving, preserving, and making accessible all software publicly available in source code form ever produced by humankind. Launched in 2016, it currently contains around 16.6 billion unique source files and 3.5 billion unique commits from more than 258 million publicly available software projects, for a total of 1PB data[3], crawled from code repositories like GitHub and GitLab[4]. The SWH archive has been exploited to analyze geographic and gender diversity in public code contributions [11–13], license text variants [14], repository forks identification [15], and various code usage statistics, such as the most likely filenames [16], commits patterns [17], and average size of the most popular file types [15].

The SWH graph dataset. To facilitate the traceability of software artifacts and minimize storage requirements, SWH projects are stored as a Merkle directed acyclic graph (DAG) [18]. Specifically, a Merkle DAG is a DAG where each node has an identifier resulting from the hashing of the node's content and the list of identifiers of its children using a cryptographic hash function like SHA256. This inherent peculiarity of Merkle DAGs makes these structures a versatile and efficient solution for data integrity verification, deduplication, synchronization, and security in diverse applications.

The SWH DAG [19] is organized in six logical layers represented in Fig. 1. In more detail, the SWH data model supports:

- *Contents* or *blobs*, which represent the graph's leaves and contain the raw content of source code files;
- *Directories*, namely source code trees;

[3] As in July 2023.

[4] https://archive.softwareheritage.org

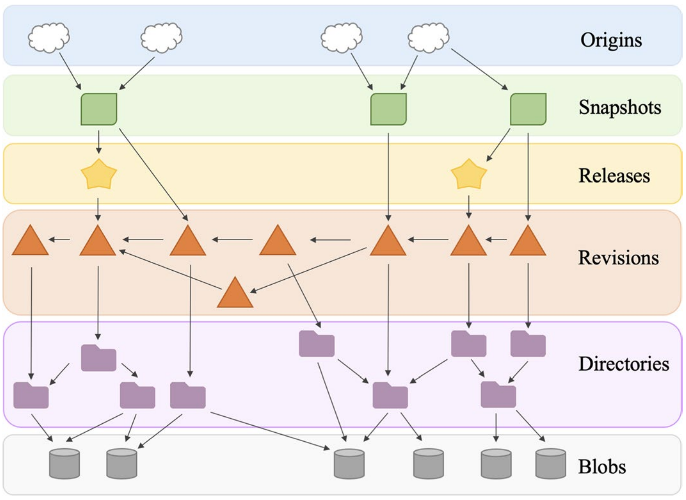


Fig. 1 The SWH data model, stored as a Merkle DAG

· *Revisions* or *commits* which are point-in-time captures of the entire source tree of a development project;
· *Releases* or *tags* which are project-related revisions that have been marked;
· *Snapshots* which are point-in-time captures of the full state of a project development repository. In other words, when the SWH framework crawls projects from a software development repository, it makes a snapshot of its current state;
· *Origin* nodes which represent software distribution repositories, such as public Git repositories identified by URLs. These nodes represent the graph roots pointing into the Merkle DAG.

The Merkle DAG is encoded in the dataset as a set of relational tables. Further, the dataset also includes crawling data in the form of triples, capturing details about where a specific snapshot was encountered (origin URL) and when it was encountered (timestamp). We refer the reader to the SWH's official documentation for more details on its data model [20].

2.2 The SWH-Analytics infrastructure

The Software Heritage Analytics (SWHA) framework has been designed and developed in the context of the ADMIRE European project, whose main objective was to produce software solutions to enhance the throughput of HPC systems and the performance of individual applications. In addition, the project aimed to decrease energy consumption while offering quality of service and resilience. In particular, SWHA was built as a development and runtime environment tailored for applications created to analyze the software preserved over time by SWH. In other words, SWHA provides an environment that empowers users to perform any query allowed by SWH on its dataset. This encompasses not only SWH software

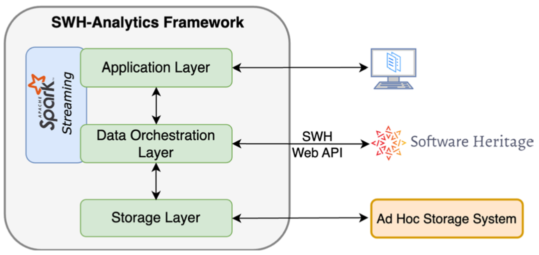


Fig. 2 The SWHA architecture

projects but also extends to projects' subdirectories, files (such as retrieving all README files in the entire dataset), or even all versions of a project within a specific timeframe. SWHA was created to address the challenges posed by querying and analyzing the vast scale of the SWH dataset by

1. *Providing a structured platform for querying and storage management.* SWHA offers a structured platform that alleviates users from explicitly managing queries to the SWH repository and handling storage resources. This functionality simplifies accessing and extracting information from the SWH dataset, allowing users to concentrate on their analyses without the need to handle the technical aspects of querying and storage.
2. *Offering an efficient data update.* Another key aspect of SWHA is its ability to facilitate the updating of existing data and analyses. This capability is particularly beneficial when users need to track changes or updates in the SWH dataset. Rather than reprocessing the entire dataset, SWHA enables users to download and process only the new data. This functionality enhances efficiency by allowing users to incorporate the latest information, saving time and resources in situations where only incremental updates are required.

The SWHA architecture is made up of three main software layers, namely storage, data orchestration, and application layers. Figure 2 schematically represents how layers communicate, while Fig. 3 details the execution and data flow within the framework. A description of each layer follows.

Storage layer. This layer primarily comprises a data cache named *Cachemire*, which speeds up the data retrieval and computation process. Cachemire implements a distributed key value storage system, where the key corresponds to a project's unique identifier assigned by SWH, while the associated value encapsulates the project package in a compressed tgz format, along with any potential analysis outcomes related to the project. Precisely, depending on the specific application, the project identifier corresponds to the identifier of any of the project's main directories in the SWH Merkle tree (see Sect. 2.1). The Cachemire interface provides a simple API that exposes PUT and GET functions, and their implementation relies on locking mechanisms offered by Posix-compliant file system primitives. Cachemire adopts the LRU algorithm (least recently used) as the cache replacement policy. To manage cache size effectively, an external script runs at

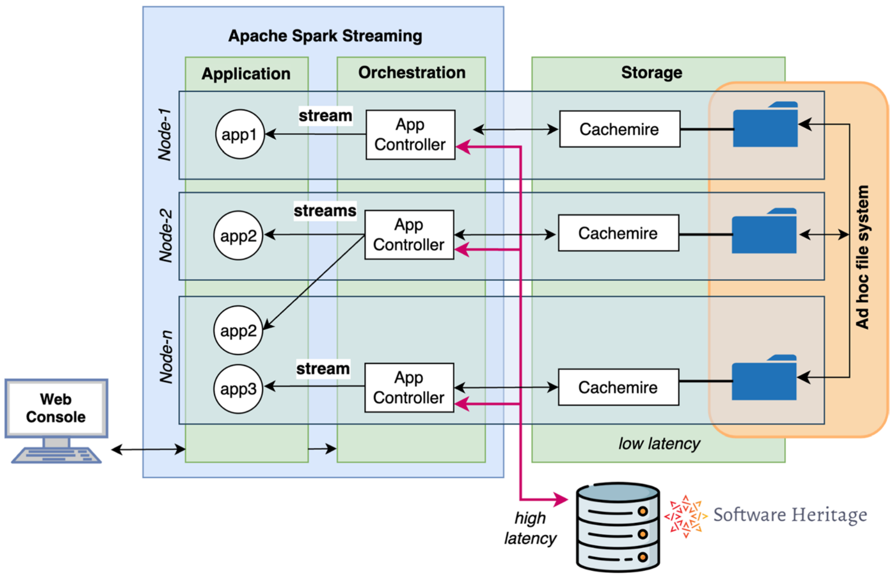


Fig. 3 The SWHA data flow

regular intervals, actively monitoring and maintaining the size within the predefined threshold.

A pivotal feature facilitating workload balancing is the ability to launch additional Cachemire instances across multiple nodes dynamically, provided each node offers a mounting point to a distributed file system. This storage infrastructure is seamlessly delivered by the ADMIRE framework through specialized storage systems like GekkoFS or Hercules. The synergy between a cache component optimized for use with such ad hoc storage systems significantly augments the efficiency and reduces the completion time of applications developed within the SWHA framework.

Data orchestration layer. This layer incorporates a pool of data stream generators (*app controllers*), which cooperate with Cachemire and the user-defined applications in a parallel computing environment. In particular, the orchestration and application layers communicate via the Apache Spark Streaming Framework[5] (see Fig. 2). In more detail, each app controller within a computing node is responsible for managing a user application that may be distributed across multiple computing nodes. Upon receiving a query for the SWH repository to execute, the controller initially checks whether the requested data or computation is already present in Cachemire. If not, the controller then queries the SWH dataset and awaits the arrival of the required data. Once SWH begins transmitting the data, the application controller directs this data flow to the Apache Spark Streaming framework, as illustrated in Fig. 3.

Apache Spark is an open-source, distributed computing framework designed for big data processing and analytics, with widespread use and a highly active developer

5 https://spark.apache.org/streaming

community. It was developed in response to the limitations of the Hadoop MapReduce model, offering significant performance improvements and enhanced capabilities for various data processing tasks. In this project, we leveraged the Spark Streaming version to build low-latency applications and, thereby, enhance the efficiency of the data retrieval to the computation cycle. In particular, stream-based analytics enabled the real-time processing of data, allowing on the fly analysis without the need to store the complete dataset locally. This approach allowed us to tackle the limitations posed by the vastness of SWH while offering a scalable and effective method for extracting valuable insights from the repository without overwhelming local storage resources and computation capabilities.

Application layer. SHWA offers the capability to run custom analytics applications written in Scala. In addition to Java and Python, Scala is among the programming languages compatible with the Apache Spark framework. Employing custom applications written in Scala ensures seamless compatibility with the underlying streaming mechanism, as we chose the Scala version of Apache Spark. This design choice was driven by the fact that, during the implementation of SWHA, the Python version did not support multiple parallel input streams, limiting the computational capabilities of the framework.

Each application can analyze a set of SWH data specified via a recipe defined by the user. Essentially, these *recipes* serve as queries formatted according to the SWH specifications and can request SWH projects, specific subdirectories, or groups of files. Recipes include essential information, such as the SWH identifier of the projects to analyze and additional metadata when needed, like the projects' programming language. The term "recipe" in SWHA is inherited from SWH due to its association with the process of preparing a set of data to download, which is colloquially referred to as "cooking". The cooking process involves the preparation of a tar.gz archive containing all the files associated with the requested data. This naming convention helps maintain consistency and aligns with the terminology used in SWH. Simply put, SWHA executes the user-defined query, manages the download process by caching the necessary data, and runs the user-defined application.

The application layer serves as the bridge for communication between an authenticated user and the SWHA system via a web-based console accessible through a web browser application. In this interface, users can perform various actions, including:

· *Project search*. Users can search for one or more projects within the SWH archive by name. Additionally, they have the option to add these projects to a recipe file for further processing.
· *Upload of custom analytic application*. Users can upload custom analytic applications in JAR format. These applications are designed to perform specific analyses on selected projects.
· *Specify the association between applications and recipes*. Users can associate an uploaded application with a recipe, specifying which projects the application should analyze. Both recipe files and application JAR files are stored in a local repository for easy management and accessibility. The system is designed to offer flexibility, allowing users to utilize the same application with multiple recipes or

the same recipe with different applications. This versatility enables users to tailor their analyses by mixing and matching recipes and applications as needed, providing a powerful and adaptable framework for data processing and analytics.
· *Application execution*. Users can initiate the execution of their application, triggering the analysis process.

This web console simplifies user interaction with the SWHA system, enabling efficient project searching, application management, and execution within a user-friendly browser environment.

3 Related work

Open-source licensing and the detection of potential license and copyright violations have been subjects of research in both industry and academia for many years [21]. In 2011, Hemel et al. introduced a system for identifying code clones in binary files to gauge the extent of this issue [22]. A decade later, the same researchers reevaluated their work and its influence [23]. They discovered that the industry and academia have progressed in the realm of license compliance and detection, largely due to recent tools like FOSSology [24] and findOSSLicense [25], as well as open-source compliance initiatives, such as OpenChain [26, 27] and SPDX [28, 29]. These resources have made it more convenient to recognize and adhere to open-source licenses, significantly reducing the likelihood of inadvertent license violations [30]. However, recent studies indicate that violations of open-source licenses continue to be a widespread problem, affecting a majority of software and hardware products containing open-source components [21, 30, 31].

The roots of such a problem may lie in two primary causes: a lack of understanding of license usage and code duplication. In the first case, Almeida et al. [32, 33] found empirical evidence that software developers generally grasp how to apply individual open-source licenses in straightforward and complex development scenarios. However, they tend to face difficulties when dealing with situations involving combinations of open-source licenses. Additionally, Kapitsaki et al. [21] categorized problems related to open-source software based on user queries on the Open Source Stack Exchange platform, showing that posts related to license texts/conditions and license/copyright notices were more prevalent, while posts discussing differences between licenses were the most widely viewed by other users. The second factor contributing to this issue can be attributed to the substantial amount of duplicated code on platforms like GitHub. In particular, Lopes et al. [34] demonstrated that out of 428 million files hosted on GitHub, only 85 million were distinct, while the remaining files were either copied from larger repositories or represented new forks of abandoned repositories. Moreover, Rousseau et al. [35] observed a high replication factor in the raw byte sequence of files. These observations indicate that license inconsistencies can likewise be replicated and propagated on a larger scale.

Over the past few years, researchers focused on analyzing license inconsistencies patterns within specific programming languages. For instance, Moraes et al. [36]

conducted a study on (multi-)license usage in JavaScript repositories, analyzing a sample of 1,552 projects. Their findings revealed that, on average, these projects had 4.7 licenses, with 61% of them employing more than one license. Moreover, nearly 40% of the multi-licensed projects experienced compatibility issues (i.e., licenses contained incompatible legal clauses; for instance, the Apache 2.0's license partner rights make it incompatible with GPL v2). Upon further investigation involving communication with project maintainers, it became evident that developers often lack an understanding of the interplay between licenses and the implications of utilizing multiple licenses. Similarly, Makari et al. [37] empirically studied the evolution, popularity, and compliance with dependency licenses in the npm and RubyGems software package ecosystems. Their research revealed that 7.3% of npm and 13.9% of RubyGems packages had direct or indirect dependencies with incompatible licenses. Notably, GPL dependencies emerged as the primary source of these incompatibilities. These findings further suggested the substantial variations that can exist between different software ecosystems concerning this issue.

Golubev et al. [38] focused on Java projects, considering the language's widespread use in the industry, where plagiarism issues hold particular significance due to potential legal consequences. Their study involved the analysis of 23,378 Java repositories sourced from GitHub, encompassing a dataset of 94 licenses. They examined the distribution of these licenses among files and estimated the likelihood of code borrowing and license violations between them. In their investigation of potential license violations within specific code segments, the authors found that approximately 29.6% of these segments might be linked to potential code borrowing, with 9.4% possibly violating the original licenses. The most recent work about open-source license inconsistencies on GitHub comes from Wolter et al. [30]. Their investigation involved the analysis of a sample comprising 1000 open-source GitHub repositories. Their findings indicated that nearly half of these repositories did not comprehensively declare all the licenses present in the code. Furthermore, among these cases, approximately 10% exhibited a mismatch between permissive and copyleft licenses. Based on this outcome, the authors advise that users of open-source code should not rely solely on the declared licenses but should also inspect the software to gain a clear understanding of its actual licensing.

The academic community has recently begun investigating the use of software bills of materials (SBOM) files. These documents comprise a structured inventory listing all open-source and proprietary software components, such as libraries and frameworks, contained within a software product, along with their licenses, versions, and vendors [39]. SBOMs are critical in ensuring transparency, thereby enhancing software supply chain security [40]. In particular, these files enable developers to identify, track, and mitigate not only reliability and security risks but also legal concerns arising from integrating software with incompatible licenses [41]. In this regard, the work from Zahan et al. [42] marks an initial effort to systematically classify the advantages and obstacles associated with implementing SBOMs. Specifically, their approach involved analyzing 200 online resources to delineate these categories, including articles, blogs, videos, and webpages. Meanwhile, Xia et al. [40] focused on examining the adoption of SBOMs among industry practitioners. In particular, they gathered data from 17 interviewees and 65 survey respondents from

15 countries across five continents to gauge practitioners' perspectives on SBOMs. Their findings highlight the importance of addressing several key factors to accelerate SBOM adoption, including enhancing the quality of SBOM automatic (AI-supported) generation, elucidating the benefits and practical applications of SBOMs, and reducing barriers to SBOM sharing. Nocera et al. [41] offer another perspective on the topic by exploring the utilization of SBOM generation tools and the publication of SBOMs within open-source projects. Their study involved an analysis of 186 public repositories on GitHub, revealing a gradual but notable increase in SBOM adoption among software creators and consumers, likely influenced by heightened attention and demand from prominent entities like the United States Government. However, their findings indicate that SBOM files are present in only 46% of the software projects examined, pointing out the current low prevalence of SBOM integration into repository or release versions.

In our work, we focus on unraveling the usage patterns of FOSS licenses across a collection of GitHub projects indexed by SWH. We specifically investigate potential correlations between license usage and various project attributes, such as project size, the main programming language employed, and the application domain. Our analysis extends to both explicitly declared licenses and those identified within the project's source code, as well as the combination of these two types. In essence, our contribution encompasses the development of a dedicated tool, the license analytics application built on top of SWHA, and a comprehensive methodology for identifying discrepancies and conflicts in project licenses.

4 The license checker analytic app

Today, open-source software has been widely adopted, and its licensing terms and conditions can significantly influence community involvement and contributions [43]. In particular, project licensing plays a critical role in companies as any violations of licenses can lead to substantial legal risks [30]. Given the extensive use of open-source software from public repositories in products, gaining a strategic understanding of multi-licensing becomes essential.

Although not directly related to the analyses performed in this work, the following two examples motivate the importance of license literacy. The case of BusyBox represents an illustrative incident highlighting the complexities arising from license inconsistencies [44]. In 2007, the Software Freedom Law Center (SFLC) initiated the first-ever US copyright infringement lawsuit rooted in violating the GNU General Public License (GPL). This legal action was on behalf of two principal developers of BusyBox, an open-source software comprising standard Unix utilities commonly used in embedded systems and licensed under the GPL version. The lawsuit targeted Monsoon Multimedia, Inc., which admitted using BusyBox in its products and firmware on its website but did not comply with GPL's requirement to provide recipients access to the source code. Although the BusyBox vs. Monsoon case concluded within a month, Monsoon had to invest significant organizational and financial resources to resolve the conflicts [45]. Another recent example dates back to March 2021, when the mimemagic software library, initially distributed under a declared MIT license, was integrated within the shared-mime-info library, which carries the more restrictive GPL license. However, the copyright notice from the shared-mime-info library was inadvertently removed during a merge operation. As a result, users of the mimemagic library had to examine the library in the repository to find the (in-code) license. This license mismatch had far-reaching consequences: the mimemagic library was an essential component of the Ruby on Rails web framework and impacted 172 other software packages, affecting approximately 577,000 software repositories. This situation led to an urgent collaborative effort to rectify the issue. These instances are merely a couple of illustrations highlighting the challenges associated with violations of FOSS licenses. However, the proliferation of generative deep learning models, which have been trained using openly accessible resources like ChatGPT and Copilot, is expected to exacerbate these issues [46, 47].

In general, with the desirable advance of the principles of open science and open-source software, it becomes of utmost importance to improve license literacy among developers for several reasons, including legal, financial, and practical considerations. Software licensing is a critical component for both software developers and users, providing a legal and financial framework that ensures fair use, protection of intellectual property, and sustained development and support for the software. Hence, considering the crucial role of proper licensing in software projects, our goal is to unveil recurring license patterns by answering the following research questions.

- RQ_1: Are there clearly identifiable patterns in the use of multi-class licenses or the appearance of conflicts within publicly available software projects?

  When we mention a multi-class licensed project, we refer to situations where a single software project incorporates (at least) two licenses with differing levels of restrictiveness. Such situations do not automatically imply the presence of a conflict, which occurs when two licenses contain contradictory rights or incompatible obligations.

  This question intends to measure the prevalence of multi-class licenses and conflicts within publicly available projects indexed by SWH. To address this, we examine how frequently such potential issues arise and whether their incidence correlates (in particular, positively grows) with the size of the projects. Additionally, we conduct a qualitative exploration of conflict instances to identify the types of licenses most likely to give rise to conflicts.

  Understanding the distribution of multi-class licenses and the most common conflicts is crucial for comprehending the common errors that may occur and for providing warnings when incorporating or reusing existing software. By answering such a question, we aim to offer users valuable insights and knowledge about common license discrepancies and conflicts that frequently arise within open-source projects, ultimately facilitating informed decision-making in software inclusion or reuse scenarios.
- RQ_2: Is the emergence of the number of (multi-class) licenses and conflicts correlated with the use of a given programming language or a specific application domain?

  This question investigates whether there is a recognizable relationship between the frequency of FOSS license usage and the occurrence of multi-class licenses and conflicts with the choice of programming language or the targeted application domain. To answer this question, we analyze a collection of publicly available projects indexed by SWH and examine whether the prevalence of license, multi-class licenses, and conflicts varies depending on the programming language employed or the specific domain for which the software is developed.

The objective is to explore whether certain languages or application domains exhibit a statistically higher likelihood of encountering problems related to multi-class licenses or conflicts. For instance, our objective is to evaluate whether projects using the C language demonstrate a statistically distinct distribution of the considered features compared to other languages. Similarly, we seek to determine if a particular prolific application domain (e.g., software related to operating systems) shows a statistically different pattern in the distribution of these features compared to other domains.

Understanding potential correlations between the choice of programming language or application domain and the likelihood of license-related problems could provide valuable insights for developers, project managers, and policymakers. This knowledge may inform best practices and guide considerations when developing software in a specific programming language (e.g., when including external code) or for particular application domains to minimize the risk of license conflicts, thereby fostering a more efficient and compliant open-source ecosystem.

4.1 Application pipeline

An asset of SWHA is that it allows the execution of custom analytic applications. Specifically, in this work, we demonstrate the effective utilization of SWHA to investigate license inconsistencies. This analysis can extend to cover all revisions of every source code ever created or focus on specific partitions, showcasing the platform's versatility and analytical potential.

The application pipeline comprises three main steps: (i) dataset creation, (ii) license identification, and (iii) license compliance verification, including the detection of any multi-class licenses and conflicts. While the data orchestration layer transparently handles the data retrieval phase, the core logic of the application—which includes the tasks of license identification and verification of their compliance—is managed by the analytical application. The application output is a JSON file that provides information for each project, including the number and types of licenses detected, their categories, and the presence of any conflicts. Additionally, the application can generate a summary detailing the quantity and types of the identified multi-class licenses, as well as pairs of licenses causing conflicts. A more detailed description of each phase of the application workflow follows.

4.1.1 Dataset creation

The initial stage involves providing the application with the designated set of SWH projects a user wishes to examine. Specifically, in this use case, for each project, we considered only the directory associated with the last revision (i.e., commit) of the last snapshot (e.g., SWH capture of the full state of a project development repository) in the SWH Merkle tree (see Sect. 2.1). This project set is defined using an arbitrarily complex and customizable 'recipe' to query the SWH archive via web API (see Sect. 2.2 §Application layer). Each *app controller* is responsible for querying the dataset and streaming each project's files (one file per time) to the analytic application (see Sect. 2.2 §Data Orchestration layer).

- In our work, we analyzed 835 unique GitHub repositories indexed by SWH. Specifically, we included the top 100 most starred and the top 100 most-forked projects. To this initial set, we added the top 100 most-starred projects for each of the following programming languages: C, Java, JavaScript, Julia, Kotlin, Python, R, and Rust. We retrieved the list of these projects from the GitHub project https://github.com/EvanLi/Github-Ranking, which consistently updates and maintains a list of the most-starred and most-forked GitHub repositories on a daily basis. Our initial dataset originally consisted of 1000 projects, but we refined it by removing duplicate entries and focusing solely on those that SWH indexes. It is important to emphasize that the projects within our sample are publicly accessible repositories, but not all are necessarily OSS. Some may lack any license, while others might contain a non-free license.

4.1.2 License identification

Building upon prior research [30], we specified two primary methods through which software licenses can be specified: declared and in-code.

· A *declared license* is explicitly designated for the entire project, often located in a license file within the project's root directory.
· An *in-code license* is a license discovered within the project's directory structure, either as stand-alone license files or within source code files, typically located within the header file.

Both declared and in-code licenses are considered explicit licenses since they are visibly provided as part of the repository. However, it is essential to acknowledge that a repository's declared license may differ from one or more in-code licenses. Such discrepancies can arise when a repository incorporates an external software library without appropriately documenting the associated license as a declared license. Consequently, users might not be aware of the license terms and inadvertently overlook the obligations outlined in the in-code licenses.

- In our work, licenses are automatically detected with ScanCode[6], one of the most popular open-source license scanners available today. For the license detection task, ScanCode uses a (large) number of license texts and license detection rules that are compiled in a search index. During the scanning process, the text of the target file is extracted and used to query the license search index and find license matches[7]. Our application looks for one or more declared licenses and in-code licenses attached directly to files by running ScanCode on each streamed file. Specifically, we search for the presence of one or more declared licenses in the root directory of the repository by explicitly checking for the existence of any of the following files: LICENSE, LICENSE.txt, COPYING, COPYING.TXT, NOTICE, README, and README.md. If none of these files are found, we assume the project lacks any declared license.

4.1.3 License compliance verification

The final step in the application workflow involves categorizing license restrictiveness for projects with more than two licenses, either declared or in-code. If a project includes multi-class licenses, the application assesses whether there is a license conflict by querying a license compatibility matrix. A detailed explanation of the evaluation process for inconsistencies and conflicts follows.

Detecting multiple licenses. Real-world software projects usually include more than one license, leading to what we term multi-licensed projects. When these projects encompass licenses with varying restrictiveness, we classify them as multi-class licensed projects. It is crucial to note that the presence of multi(-class) licenses does not automatically imply conflicts, as the involved licenses may still be compatible. However, the inclusion of multi-class licenses does heighten the possibility of conflicts. In general, multi-licensing can potentially escalate into significant license conflicts, especially if undisclosed in-code licenses are more restrictive than declared ones [30].

- We based our analysis on the existing license categories listed in the ScanCode dataset[8] to identify multi-class licenses. In particular, we considered the following (standard) license classes:

  - *Public domain*: These licenses grant unrestricted freedom to use and modify the software.
  - *Permissive*: This category encompasses licenses with minimal restrictions or requirements for distributing or modifying the software. (i.e., code can be modified and can be redistributed under a different license);
  - *Copyleft*: This is a more restrictive class of licenses, further classified into two subcategories:

[6] https://github.com/nexB/scancode-toolkit

[7] https://scancode-toolkit.readthedocs.io/en/stable/reference/overview.html

[8] https://scancode-licensedb.aboutcode.org/index.json

    * *Copyleft Limited* or *Weak Copyleft*: In this category, changes made to existing code must be published under the same license. However, code utilizing the existing code does not necessarily have to follow the same requirement.
    * *Copyleft* or *Strong Copyleft*: This category mandates that changes to existing code and all code using the existing code must be published under the same license.

- *Proprietary* or *Commercial*: These licenses impose the most stringent restrictions, rendering the software ineligible for copying, modification, or distribution. They serve as the most protective type of software license, safeguarding the developer or owner from unauthorized software use.
- *Unstated* and *Unknown*: The first category pertains to licenses that have been indexed by ScanCode, but their specific category or type has not been explicitly identified or specified. In other words, these licenses are recognized by ScanCode, but it is unclear which particular category they belong to. The second category involves licenses that are not indexed by ScanCode. This situation often occurs when a project employs a combination of licenses, making it challenging for ScanCode to accurately classify or categorize them.

Detection of license conflicts. A conflict occurs when two licenses contain contradictory rights or incompatible obligations. Lack of compliance with these terms can lead to a spectrum of issues, ranging from relatively straightforward disputes to protracted legal conflicts [43, 48]. In the most severe scenarios, court-issued injunctions can order the immediate stop of the product sales [30].

- Our analysis focused on the compatibility of FOSS licenses. Hence, we identified possible license conflicts exclusively when both inconsistent licenses fell within this category of licenses. Specifically, we relied on the OSADL Open Source License Checklist project[9] for the conflict detection task. This initiative was launched with the objective of creating comprehensive checklists that delineate the obligations associated with widely adopted open-source software licenses that were accepted and trusted by distributors, copyright holders, and users. The project's website hosts materials, including detailed obligations and use cases for each FOSS license. Additionally, it provides supplementary information such as references to copyleft clauses, patent-related insights, and assessments of compatibility (or incompatibility) with other licenses. In particular, we exploited a compatibility matrix between licenses available on the project's website[10].

Enumerating combinations of multi-class licenses and conflicts. We employed the following methodology to quantify the occurrence of multi-class license pairs and conflicts within our analysis. Consider a project equipped with one declared license of type A and five in-code licenses of type B. In the presence of a conflict between types A and B (indicating that one type imposes more restrictions than the other),

9 https://www.osadl.org/OSADL-Open-Source-License-Checklists.oss-compliance-lists.0.html

10 https://www.osadl.org/fileadmin/checklists/matrixseqexpl.json

Table 1 Distribution of the projects' size (in terms of number of source code files), the number of (unique) declared licenses, and the amount of (unique) in-code licenses

Data	Quantiles				
	0.1	0.25	0.5	0.75	0.9
Project size	20	69.3	235	899.5	4267.7
Project licenses	2	4	13	137.5	1400.6
Unique project licenses	1	2	3	6.75	17
Project declared licenses	1	2	3	9	33.1
Unique project declared licenses	0	1	1	2	3
Project in-code licenses	0	1	8	115	1372
Unique project in-code licenses	0	1	2	5	14

Table 2 Distribution of the number of multi-class license pairs and conflicts among declared licenses, in-code licenses, and between them. The first number in the parentheses represents a lower bound in the calculation since it only accounts for known licenses while excluding the unknown or unstated categories

Type	Data	Quantiles				
		0.1	0.25	0.5	0.75	0.9
Multi-class license occurrences	Between declared licenses	(0, 0)	(0, 0)	(0, 0)	(0, 0)	(0, 2)
	Between in-code licenses	(0, 0)	(0, 0)	(0, 1)	(5, 7.75)	(45.2, 62.2)
	Between declared/in-code licenses	(0, 0)	(0, 0)	(0, 1)	(3, 4)	(19, 27.1)
Conflicts	Between declared licenses	(0, 0)	(0, 0)	(0, 0)	(0, 0)	(2, 3)
	Between in-code licenses	(0, 0)	(0, 0)	(1, 1)	(6, 8.75)	(51.2, 80.1)
	Between declared/in-code licenses	(0, 0)	(0, 0)	(1, 1)	(4, 4.75)	(23, 31)

we compute 5 pairs of multi-class licenses and 5 conflicts (since multi-class licenses can potentially lead to conflicts). The rationale behind adopting this approach is rooted in our intent to consider all potential warnings and issues within our dataset. In other words, we scrutinize all licenses within a project to identify potential problems.

This approach can furnish valuable insights into the license health of a project in a real-world scenario, such as within a company's project dashboard. Specifically, our method provides information about the resolution of license class mismatches that give rise to conflicts. For instance, in the aforementioned example, if the declared license is modified to align with type B, the count of conflicts drops to 0. Similarly, adjusting one of the five in-code licenses to align with the declared license of type A results in a reduction of conflicts to 4. This demonstrates how our approach facilitates a nuanced understanding of the license status and aids in devising strategies to mitigate conflicts within a project.

5 Results and discussion

In this section, we walk through the outcomes of our analyses, exploring the presence of any noticeable patterns in the occurrence of multi(-class) licensing and license conflicts. We also examine whether these patterns exhibit any correlation with the projects' programming language or application domain. The code to reproduce the analyses presented in this article can be found on Zenodo and GitHub [49].

5.1 Patterns of multi(-class) license usage and conflicts

The first aspect we looked into was assessing the proportion between the size of the projects and the total number of licenses, either declared and in-code, available in the projects' repositories while also examining the prevalence of multi(-class) licensing and conflicts. Table 1 summarizes part of these results, showing the distribution of the number of files per project as well as the number of the projects' (unique) declared and in-code licenses. The table also reports the distribution of the overall number of licenses per project, regardless of their type. Table 2 focuses on the distribution of pairs of multi-class licenses and conflicts among declared licenses, in-code licenses, and between them. It is worth noting that the number of multi-class license pairs reported in the table is lower than the number of conflicts. In the former case, we consider the license categories, while in the latter, we identify conflicts by examining the specific licenses involved (see §Enumerating occurrences of multi-class licenses and conflicts).

As highlighted in Table 1, we found a significant presence of huge projects within the dataset, which aligns with our expectations, as we specifically retrieved projects with the highest number of stars and forks, indicating their popularity and potentially larger dimensions. Interestingly, only 25% of the projects possess a unique license, denoting a relatively low incidence of single-licensed projects. It is not surprising that in cases where a single license is present, it is typically a declared license. In contrast, 10% of projects demonstrate a notable complexity with over 17 unique licenses. Within this subset, the complexity is primarily attributed to 14 in-code licenses and three declared licenses, implying a substantial degree of intricacy within these projects. The majority of the dataset, constituting at least 50% of the projects, falls within the range of 2 to almost 7 unique licenses. In this category, projects typically utilize at most two unique declared licenses and five in-code licenses, thus highlighting a prevalent licensing pattern commonly followed in practice. One striking observation concerns the prevalence of in-code licenses, as their number is significantly higher in comparison to declared licenses. This disparity arises because each project may incorporate external code, often accompanied by its distinct license terms. Consequently, the cumulative number of in-code licenses tends to surpass the count of declared licenses, reflecting the complexities of managing diverse code components within a single project. Despite the lower values, this relation holds true when examining the count of unique in-code licenses. Overall, this analysis revealed varying degrees of licensing complexity among projects, with a significant portion adhering to a common practice of incorporating a moderate number of both declared and in-code licenses.

Within our dataset, we identified three projects lacking any FOSS licenses and 119 projects featuring at least one non-FOSS license. Notably, within this latter group, the majority (61 out of 119) employ commercial licenses, while an additional 24% (29 out of 119) specify a 'Source-available' license type. Six out of the 119 projects are patented. The remaining projects exhibit a blend of commercial, patented, and source-available licenses.

Looking at the results in Table 2, we can note how approximately half of the samples in our study are multi-class licensed projects. These multi-class license pairs occur in two primary categories: between in-code licenses and between declared and in-code licenses (cross-combinations). We can observe a significant increase in the use of multi-class licensing in 25% of the projects. In this subset, the number of multi-class license pairs significantly escalates, with 7.75 multi-class license pairs between in-code licenses and 4 cross-combinations. In 10% of the projects, these numbers reach a surprising 62.2 multi-class license pairs between in-code licenses and 27.1 cross-combinations, revealing the existence of a minority of projects struggling with substantial licensing intricacies. As expected, most conflicts within the projects occur among in-code licenses. This trend is unsurprising given the volume of in-code licenses and the inherent difficulty in their verification, often necessitating a thorough examination of the source code. Nevertheless, it is noteworthy that 10% of the projects within our dataset experience a notable level of conflict in the form of at least 3 declared license conflicts. This finding emphasizes that, despite the prevalence of in-code conflicts, declared license conflicts remain relevant and warrant attention.

In particular, we observed that the size of such projects ranged from 155 to over 80,000 files, with 50% having more than 4,600 source code files. To investigate the potential correlation between project size and the number of (unique) licenses, as well as the occurrence of multi-class license pairs and conflicts, we correlated such values via the Pearson and Spearman correlation coefficients. Figure 4 graphically illustrates the relationships between these values. In all plots, a notable upward trend is evident, indicating that as the number of source code files in a project increases, there is a corresponding increase in the occurrences of licenses, multi-class license pairs, and conflicts. Such a monotonic increasing trend is substantiated by the strong Spearman correlation coefficients obtained (0.77, 0.71, 0.69, 0.70, respectively; p values < 0.0001)[11].

Taking a closer look at the types of multi-class license pairs, we identified that a significant proportion of those stem from the discrepancies between FOSS licenses and licenses categorized as "unstated" or "unknown" by ScanCode, the license detection tool we used. The prevalence of such instances stresses the tool's critical role in identifying licenses. Additionally, this situation emphasizes the existence of numerous custom licenses, further complicating the already intricate licensing landscape. Another not negligible portion of license combinations happens with all licenses that fall outside the realm of FOSS, for instance,

[11] This relation does not change if we do not consider unstated and unknown licenses in the calculation.

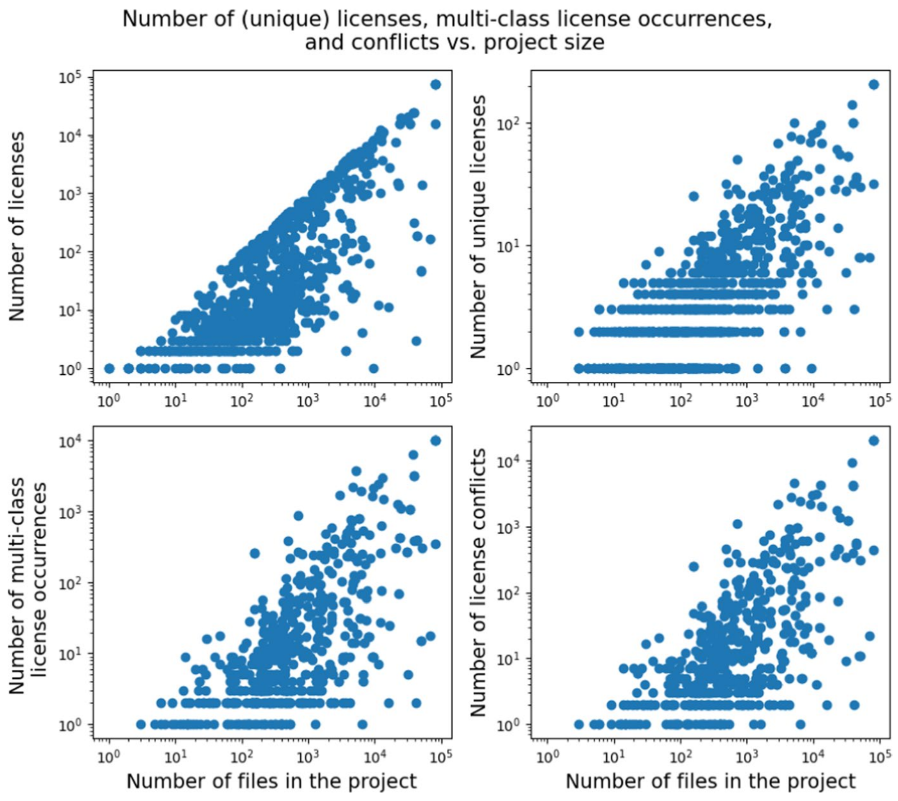


Fig. 4 Correlation between the project size and the following factors: the number of licenses (upper left), unique licenses (upper right), occurrences of multi-class license pairs (bottom left), and occurrences of license conflicts (bottom right)

between a permissive and a commercial license. These combinations are of particular concern due to their potential to trigger legal disputes, disrupt established business models, and introduce operational and technical complexities in the management and integration of software components. One prominent finding of our analysis is the prevalence of multi-licensing between permissive and strong copyleft licenses, aligning with prior research in this domain [30]. Permissive licenses, such as the MIT License and the Apache License, grant considerable freedom for using, modifying, and distributing the licensed software. In contrast, strong copyleft licenses, such as the GNU General Public License (GPL), mandate that any derivative works or software incorporating GPL-licensed code must adhere to the same GPL terms. Combining or linking code under strong copyleft licenses with code governed by permissive licenses typically results in the strong copyleft license dominating the entire work. A substantial percentage of multi-class license pairs also occur between permissive licenses and weak copyleft licenses. Weak copyleft, or copyleft limited licenses, impose fewer constraints compared to strong copyleft licenses. However, these combinations may

Table 3 Percentage of types of multi-class licensing

	Permissive	Copyleft limited (weak Copyleft)	Copyleft (strong Copy-left)	Proprietary/ commercial	Unstated/ unknown
Permissive	–	4.28	10.08	8.31	38.28
	–	6.64	6.68	9.52	22.41
	–	7.82	8.12	12.29	27.76
Copyleft limited	*4.28*	–	2.77	1.26	5.04
	6.64	–	3.80	4.80	9.43
	7.82	–	3.18	2.29	7.76
Copyleft	*10.08*	*2.77*	–	1.51	13.6
	6.68	*3.80*	–	4.41	9.34
	8.12	*3.18*	–	3.47	9.53
Proprietary/commercial	*8.31*	*1.26*	*1.51*	–	14.11
	9.52	*4.80*	*4.41*	–	19.62
	12.29	*2.29*	*3.47*	–	16.24

Values from top to bottom refer to multi-class pairs between declared licenses, in-code licenses, and cross-combinations

still lead to conflicts. Surprisingly, we found that combinations between weak and strong copyleft licenses are less common. Table 3 summarizes the percentage of license combinations for each pair of license categories. The missing percentage refers to combinations between non-FOSS licenses.

Focusing on the license conflicts detected in our dataset, we found out that the most common involve the permissive licenses Apache 2.0 and MIT versus the strong copyleft licenses GPLv2 and GPLv3, as also found out by Makari et al. [37]. As already discussed, the incompatibility between these licenses primarily arises from the different goals and terms of these licenses, specifically regarding the openness and redistribution of derivative works. The Free Software Foundation, which maintains the GPL, introduced provisions in the GPLv3 to address compatibility issues with other licenses. Nevertheless, it is important to note that compatibility is often unidirectional, implying that code governed by the GPLv3 can be incorporated into projects using certain permissive licenses (such as the Apache License 2.0), but the reverse may not hold true. We also observed common conflicts arising from discrepancies between various versions of the GPL license and the blending of different existing licenses into a new one.

5.2 Correlation between license usage and issues with programming languages and application domains

In the previous section, we discussed the existing robust positive correlation between project size and the quantities of (multi-class) licenses, and conflicts. This outcome aligned with our expectations, as larger projects tend to be more intricate, encompassing additional code with its unique licensing terms, which, in turn, heightens the likelihood of causing conflicts. Building upon this finding, we investigated whether specific inherent characteristics of projects might interplay with the emergence of these issues. Specifically, we considered the projects' main programming language and application domain. To rule out the potential influence of project sizes, we normalized the number of licenses, multi-class license instances, and conflicts by the respective project sizes. This normalization process allowed us to account for variations in project scale, ensuring a fair assessment of the associations between the mentioned variables, thus avoiding larger projects driving the results and hindering actual patterns.

To verify the existence of any potential correlation, we employed the Kruskal–Wallis H test [50], which assesses the null hypothesis that the population medians of all groups are equal. This test is the nonparametric alternative to the ANOVA test, used when the data to examine do not follow a normal distribution. In particular, if the null hypothesis is rejected, it indicates a statistically significant difference between groups without specifying which groups differ. To discern which groups exhibited statistically significant differences from one another, we utilized the Mann–Whitney U test, a nonparametric test that evaluates the null hypothesis that the distribution underlying sample x is identical to the distribution underlying sample y.

Correlation between programming languages and license usage and issues. Within our dataset, we initially had 29 distinct programming languages[12]. However, to fulfill the prerequisites of the Kruskal-Wallis-H test[13], we focused our analysis solely on languages associated with more than 5 projects. As a result, we conducted our analysis on the following languages: C, C++, Go, HTML, Java, JavaScript, Julia, Kotlin, Python, R, Rust, Shell, and TypeScript. In the following, we only report the results associated with the programming languages having more than 80 projects each in our dataset (corresponding to the 75 percentile).

Figure 5 illustrates the distribution of the average number of licenses, multi-class license instances, and conflicts per project across programming languages. From the top plot, it is evident that C, Java, and Kotlin projects exhibit the highest median in the average number of licenses per project. This implies that in over half of the projects for each of these languages, approximately 40% of the files are associated with a license (a value of 1 means that every file in the project has a license attached). Further, these languages also present the highest variability in the distribution. Interestingly, the higher average number of licenses in these projects does not necessarily correspond to a higher average number of multi-class license instances and conflicts. A closer examination of the middle and bottom plots reveals that C projects tend to have a greater distribution of the average number of multi-class license instances and conflicts, while Java and Kotlin projects exhibit a median value trending toward a smaller quantity. This observation suggests that C projects experience a relatively higher prevalence of multi-class

[12] We indistinctly refer to programming languages, although some of them are markup or scripting languages.

[13] https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.kruskal.html

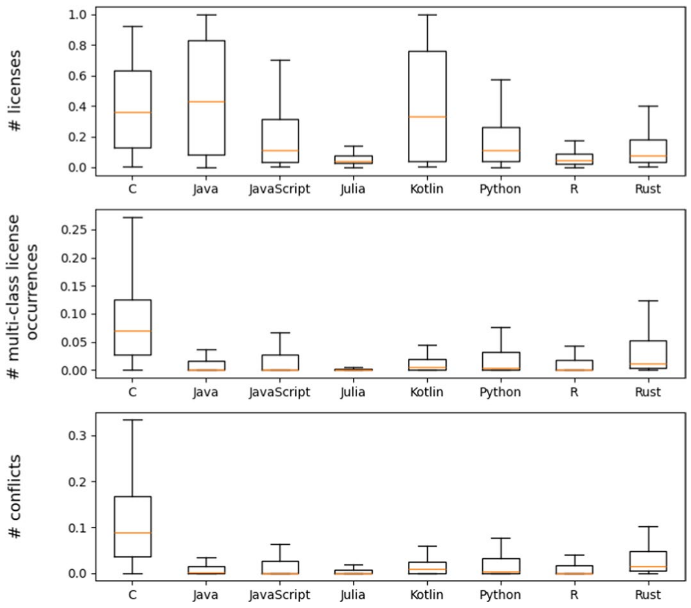


Fig. 5 Distribution of the average number of licenses (top), multi-class license instances (center), and conflicts (bottom) normalized by the projects' size for each programming language

license instances and conflicts compared to Java and Kotlin projects. The rationale behind this outcome necessitates further investigation, as it may be influenced by several factors other than the project size and complexity, which should be identified through surveys administered to developers and ad hoc experiments. Possible causes include:

· *Library and dependency ecosystem.* Different programming languages have varying ecosystems of libraries, frameworks, and dependencies, each with its own set of licensing terms. When projects integrate various libraries with varying licenses, conflicts can arise.

· *Licensing awareness and culture.* The level of awareness and understanding of open-source licensing can vary among developers and project maintainers. Some languages and communities may prioritize license compliance and education. Further, communities (from companies) with a strong focus on licensing are more likely to identify and resolve conflicts promptly.

· *Legal and compliance resources.* Organizations with dedicated legal and compliance resources may be better equipped to identify and address licensing issues.

· *Tools and automation.* The availability of tools and automation for license compliance checking can also affect the likelihood of conflicts.

· *Historical precedents.* Some programming languages may have had more license-related issues in the past, leading to increased awareness and efforts to avoid conflicts in newer projects.

Table 4 Summary of statistical difference in the distribution of conflicts across programming languages

Language	C	Java	JavaScript	Julia	Kotlin	Python	R	Rust
C	-	●	●	●	●	●	●	●
Java	○	-		●	●			●
JavaScript	○		-	●				●
Julia	○	○	○	-	●	●		●
Kotlin	○	○		○	-		●	●
Python	○			○		-		●
R	○				○		-	●
Rust	○	○	○	○	○	○	○	-

The symbol ● indicates that the two languages statistically differ regardless of the number of projects considered and the projects' size

The statistical tests we ran provided numerical confirmation of the observed visual distinction in the distribution depicted by the box plots. Specifically, our analysis revealed statistically significant differences among these groups in terms of licenses, multi-class license instances, and conflicts (all with p-values < 0.001). Table 4 provides a summary of the pairwise differences in programming languages regarding license conflicts (with p-values < 0.001). Similar results were obtained for the number of licenses and multi-class license instances. In conjunction, these outcomes suggest an inherent connection between the average number of licenses in a project and the programming language employed for its development. Moreover, our results indicate an increased likelihood of encountering statistically significant higher numbers of multi-class license occurrences and conflicts when examining C projects within the most highly starred projects.

Correlation between application domains and license usage and issues

To identify the application domain of each project, we relied on the set of keywords provided by the project's authors on its GitHub repository. Specifically, we assigned each project to an application domain according to the following protocol:

- Initially, three researchers individually examined the entire pool of keywords in the dataset, each creating their own lists of potential application domains based on the identified keywords. For instance, terms such as *Android* and *iOS* were linked to the *Mobile* domain. Then, the researchers collaborated to merge their individual lists, resulting in a refined final list of application domains (detailed in Table 5).
- Then, each reviewer independently assigned each project to one of the application domains identified in the previous step based on the keywords present in the project's GitHub repository;

Table 5 Application domains and their description

Application Domain	Brief Explanation
Blockchain	Applications related to blockchain technologies.
Db	Database-related applications or database implementations.
Dev	Development tools and environments. This category generally includes all libraries and applications for data analysis and artificial intelligence.
DevOps	Development and operations (DevOps) tools.
Education	Educational applications and platforms, tutorials, and educational material.
Events	Event-related repositories, such as Hackathons.
Framework	Frameworks for building applications, such as Visual Studio Code.
IoT	Internet of Things (IoT) applications.
Mobile	Mobile applications for smartphones and tablets.
Networking	Networking-related applications and tools.
OS	Operating system applications, such as command-line tools.
Other	Applications falling into other categories.
Search-engine	Implementation of search engines or plugins.
Visualization	Libraries and applications for data visualization.
Web	Web applications and development.

- Finally, the definitive application domain for each project was determined through a majority rule. In cases of ties, decisions were reached through discussions among the researchers.

As for the programming languages, we filtered out application domains with fewer than five associated projects. Table 5 lists the domains we analyzed with a brief description. In the following, we only report the results associated with the application domains having more than 30 projects each in our dataset (corresponding to the 75 percentile).

Figure 6 illustrates the distribution of average number of licenses, multi-class license instances, and conflicts per project across application domains. The first point to note is that projects in the Mobile application domain exhibit the highest average number of licenses per project (with half of the projects having around 40% of files associated with a license). We can generally observe a high variability in this dimension across all domains shown in the plot (except the Dev and Education domains). Once again, a higher number of licenses does not necessarily imply an increased incidence of multi-class license instances and conflicts. This is exemplified by the Mobile domain, which, despite having the highest median value in terms of licenses, does not exhibit a proportionately higher number of multi-class licenses and conflicts. Conversely, the OS domain shows the highest incidence of multi-class license pairs and conflicts, even though it has a relatively low median value in terms of licenses. The statistical tests numerically confirmed the visual differences in the distribution represented by the box plots. Specifically, these tests indicate that the distribution of multi-class license occurrences and conflicts in the Events and OS

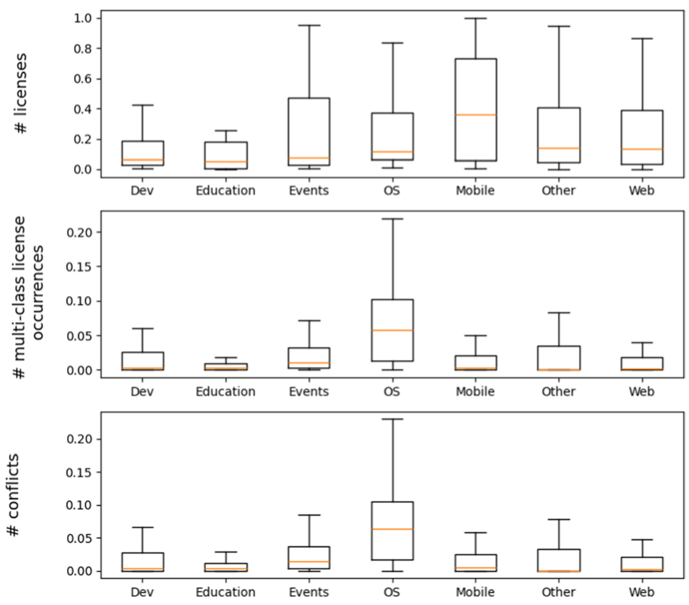
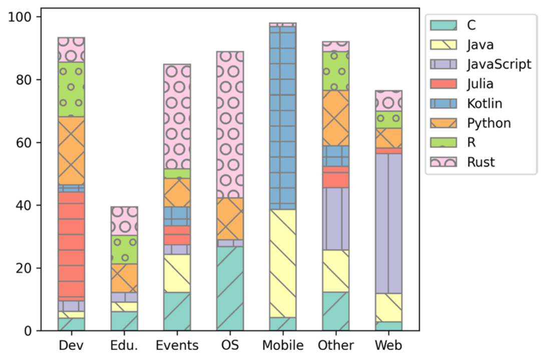


Fig. 6 Distribution of the average number of licenses (top), multi-class license instances (center), and conflicts (bottom) normalized by the projects' size for each application domain

Fig. 7 Distribution of programming languages across application domains. The report includes only programming languages associated with a minimum of 80 projects in our dataset. The unaccounted percentage refers to the other languages in our sample



domains exhibits a statistically significant difference from all other domains presented. Surprisingly, within the Education domain, which is primarily comprised of instructional materials such as guides, tutorials, and annotated API collections, there is a notable absence of significant multi-class license instances and conflicts despite the possibility that the creators may have deliberately integrated other software projects. This observation implies that repositories focused on training or education tend to have straightforward licensing terms. This simplicity may arise either from authors intentionally omitting license information or from the absence of any encountered issues related to licensing.

As previously mentioned, further investigation is required to fully understand the reasons behind these outcomes. These reasons align with those discussed for specific programming languages, as the choice of a programming language is closely tied to the given application domain. Figure 7 shows the distribution of the programming languages with more than 80 projects in our dataset across the application domains considered. Notably, C projects contribute to 26.7% of the composition in the OS domain, marking the highest percentage among all domains. Based on this result, it is plausible to attribute a significant portion of the multi-class license instances and conflicts in this domain to these C projects. We can further speculate on additional motivations why a given application domain has a higher prevalence of (possible) license issues (e.g., Events domain):

- *Software component diversity.* Different application domains may necessitate varying levels of software component diversity. Projects that use a wide variety of third-party libraries and components are more likely to encounter license conflicts. This complexity is particularly evident in Mobile projects, which often necessitate the integration of numerous diverse libraries.
- *Presence of copyleft licenses.* Some licenses, like the GNU General Public License (GPL), contain copyleft provisions that require any derivative work to also be licensed under the same terms. Projects using components with copyleft licenses can face compatibility issues if they intend to avoid releasing their entire codebase under those same terms.
- *License updates.* Over time, the licenses of software components can change. If a project fails to keep pace with these alterations or neglects to update its dependencies accordingly, it might encounter license conflicts as new component versions introduce different licensing terms. This dynamic may be particularly true for rapidly evolving technologies, such as web technologies.
- *Incompatible goals.* Projects developed for specific applications may have distinct goals and constraints. If the goals of different components are not aligned, their licenses might not be compatible. For instance, a project-oriented toward commercial usage may clash with a component employing a more restrictive open-source license.

5.3 Performance evaluation

In this section, we detail the performance of our application by first focusing on the application's running time and then on the scalability performance of SWHA. All tests have been run on 16 Broadwell nodes (hosted by the HPC4AI infrastructure), each equipped with two Intel(R) Xeon(R) CPU E5-2697 v4 processors, coupled with the Lustre parallel file system and interconnected via the OmniPath network.

5.3.1 Application's running times

The total running time comprises two primary tasks: querying and downloading projects from SWH and executing the application's core logic. Table 6 presents the

Table 6 Average time required to complete the overall computational pipeline, comprising the projects' download phase and their analysis

Recipe cooked (projects cached by SWH)	SWHA Cache (Cachemire)		Time (approx. minutes)
	SWH projects cached	Application output	
–	-	–	297.6
✓	-	–	244.0
✓	✓	–	236.3
✓	-	✓	2.9
✓	✓	✓	2.6

application's average running time to handle 100 projects. These times represent the average across ten runs, each with a distinct set of projects. The measured variance was negligible. In particular, the table quantifies the positive impact of caching on the overall performance of the system. It provides information on whether the requested projects have been previously cached, indicating if they were stored in the SWH cache due to a previous query or, in Cachemire, the caching system embedded within SWHA. Additionally, the table indicates whether the application's output, precisely the results generated by ScanCode in our scenario, were also been stored in Cachemire.

The running times reported in the last column reveal that the bottleneck in the application lies in the execution of the ScanCode tool, primarily due to its need to examine all source code files within a project (as highlighted by the significant disparity between the first three rows and the others). In particular, ScanCode required approximately 3.5 h to scan each set of 100 projects, equivalent to around 200 min. This process involved an average scanning time of 6 s per file, considering an average file size of 26.6 KB and a standard deviation of 87 KB (averaged over all ten data batches).

The advantage of pre-requesting a recipe from SWH becomes evident as well since it resulted in a time savings of 18% (comparing the first and second rows) due to SWH's cache for recently requested projects. Furthermore, storing projects in Cachemire resulted in a time savings of 21% (comparing the first and third rows). To better quantify the benefits of locally caching SWH projects, we systematically investigated the time required to download projects from the SWH dataset. Specifically, we tracked the retrieval time for projects within our dataset by querying SWH at various times and days over a week. Each query involved requesting data for 100 projects. The download process took approximately 160 s on average, with a standard deviation of around 250 s for larger projects. Downloading times were comparable across queries. However, we also experienced service disruption due to a high workload of requests to the SWH dataset, leading to processing delays of several days for our queries. For each query, the project sizes ranged from a few kilobytes (around 20kb) to as much as 4 gigabytes, with a median size of around 70 megabytes and a 95th percentile size of approximately 500 megabytes.

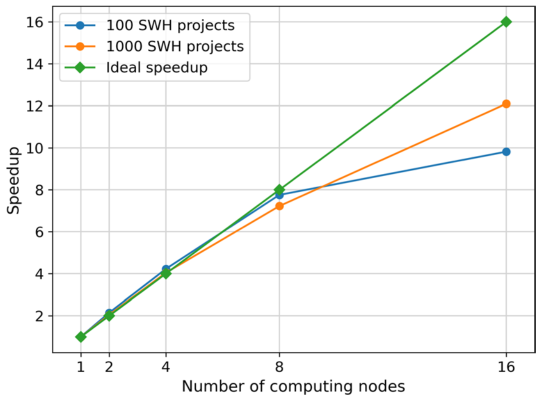


Fig. 8 Evaluation of SWHA scalability

5.3.2 Framework scalability

The scalability performance of the SWHA framework is depicted in Fig. 8. This figure illustrates the speedup achieved by executing the license application on 2, 4, 8, and 16 computing nodes. The experiment was conducted on two distinct sets of SWH projects, comprising 100 and 1000 SWH projects, respectively. In this experimental setting, both the projects and the application's results were stored in Cachemire. Interestingly, a speedup closely aligning with the ideal scenario is evident in both project sets up to 8 computing nodes. Despite a degradation in scalability performance with more nodes, we can still note a significant speedup with a higher workload. This result aligns with expectations considering the distributed nature of the SWHA environment. In such situations, a larger workload tends to enhance the computation-to-communication ratio, leading to an overall performance improvement. Conversely, with a smaller workload, the communication and synchronization overhead of the framework, coupled with reduced data locality, constitutes a more significant proportion of the overall computing time, thus reducing the system's performance.

5.4 Threats to validity

This section identifies key limitations of our case study.

Construct validity. The biggest challenge we faced in our work was defining the amount of repositories to analyze to obtain representative results. To ensure the construct validity of our findings, we crawled the most popular projects on GitHub, specifically the top 100 most starred and the top 100 most-forked projects overall. Additionally, we included the top 100 most-starred projects for each of the following programming languages: C, Java, JavaScript, Julia, Kotlin, Python, R, and Rust. The rationale behind choosing these projects lies in their high usage, making them more likely to (i) have a reasonable size, (ii) state one or more licenses, and (iii) exhibit license compatibility issues. The decision to initially select 1000 projects, later reduced to 835 after eliminating duplicates and the projects not indexed by

SWH, was driven by the limited computing hours available on the cluster used for our experiments.

A major limitation of this use case comes from the license detection tool we used. Although ScanCode is considered a state-of-the-art tool for this task, it is not always able to recognize the exact license attached to a file because of missing version numbers, spelling errors, or altered licensing text. To mitigate such an issue, we opted to incorporate only those licenses for which ScanCode provided a confidence score exceeding 95%. We still considered the categories Unstated/Unknown in our statistics to give the reader the full picture of the licensing landscape and the limitations of ScanCode.

Internal validity. To ensure the reliability and accuracy of the findings about the connection between license utilization and the emergence of potential issues in programming languages and application domains, we tried to mitigate the impact of confounding factors. Specifically, we normalized the number of licenses, multi-class license instances, and conflicts by the project size to avoid larger projects influencing the results and hindering actual patterns. This normalization process enabled us to account for variations in project scale, ensuring a fair assessment of the associations between the mentioned variables. Nevertheless, other hidden factors may impact the relations observed.

The statistical tests used in our analysis only allowed us to verify the presence of a correlation between two observed variables, such as the project size and the number of licenses. These tests also enabled us to determine whether distinct groups of projects, categorized by programming languages or application domains, exhibited statistically significant differences, meaning that the observed differences between the groups were unlikely to have occurred by random chance alone. Consequently, our conclusions were limited to discussing potential explanations for the observed patterns, and we refrained from making any inferences about causal relationships other than arguing about possible causes for the findings we had. Establishing causality should be explicitly addressed through surveys administered to developers and ad hoc experiments.

External validity. As previously mentioned, our dataset is limited to the most popular projects on GitHub indexed by SWH. While we crawled the top projects associated with the most *admired* languages based on the 2023 StackOverflow developer survey [51], broadening the dataset has the potential to provide a more comprehensive perspective on the relation between license usage and a language's developer community. The same consideration holds for the application domains.

One last reflection relates to the inherently highly dynamic nature of OS projects and licensing patterns. Our work analyzed a partial snapshot of the current OS licensing landscape, but it will surely evolve in the next few years, as also confirmed by the study run by Hemel et al. [23]. An interesting future work is running a large-scale follow-up study to delve into the temporal dimension of open-source projects and uncover trends, and implications that may not be apparent in a static snapshot. This temporal setting would enable an in-depth analysis of how the most problematic projects identified in our current study have evolved over time.

6 Conclusion

Over the past twenty years, open-source software has experienced a remarkable evolution, now enjoying extensive adoption. In this context, the SWH initiative represents a valuable source as it aims to archive, preserve, and make all software publicly available in source code form ever produced by humankind accessible. The SWH dataset experiences rapid growth, accumulating several terabytes of data each month. Although designed to archive deduplicated small files thanks to the use of a Merkle tree as the underlying data structure, querying the SWH dataset presents challenges due to the nature of these structures, which organize content based on hash values rather than any locality principle. The magnitude of the repository, coupled with the resource-intensive nature of the download process, highlights the need for specialized infrastructure and computational resources to effectively handle and study the extensive dataset housed within SWH. Currently, there is a lack of infrastructures specifically tailored for running analytics on the SWH dataset, leaving users to handle these issues manually. To address these challenges, we presented the SWHA framework, a development environment that transparently runs custom analytic applications on publicly available software data preserved over time by SWH. Specifically, this work showed how SWHA can be effectively exploited to study usage patterns of open-source licenses, highlighting the need to improve license literacy among developers.

Our analysis revealed a positive correlation between project complexity, indicated by its size, and the number of (multi-class) licenses and conflicts (RQ_1). In line with the previous literature, we identified that a substantial portion of (multi-class) licenses belong to copyleft and strong copyleft licenses [30], with GPL dependencies emerging as a primary source of conflicts [37] (RQ_1). Furthermore, a more in-depth examination of the relationships between the programming language employed and the occurrences of (multi-class) licenses and conflicts suggested the existence of a correlation between the use of programming languages and these observed features (RQ_2). These patterns persisted when considering the application domain (RQ_2).

In future work, we aim to provide additional analytical tools and applications to enhance the capabilities of SWHA users when it comes to examining the extensive dataset hosted within SWH (for instance, by offering the capabilities to analyze SBOM files in OSS [41] and analyzing the relationship between FOSS licenses and software projects under a probabilistic framework). These forthcoming additions will expand the range of analytical options available to users, allowing them to gain deeper insights and extract more valuable information from the vast dataset at their disposal.

7 Software availability

To reproduce on a local machine the results of this article, please refer to the GitHub or Zenodo repositories available in [49]. Details about the SWHA framework follow.

Software name	Software Heritage Analytics
Year of first official release	2022
Programming language	C, Python, Scala
System requirements	Linux-based system
Availability	https://github.com/alpha-unito/Software-Heritage-Analytics
Website	https://admire-eurohpc.eu/33-2/usecases/
License	MIT License

Acknowledgements This work has been partially supported by the EuroHPC JU ADMIRE project (G.A. n. 956748) and the spoke "FutureHPC and BigData" of the ICSC—Centro Nazionale di Ricerca in High-Performance Computing, Big Data and Quantum Computing funded by European Union—NextGenerationEU.

Author contributions AA designed the analytic application, performed the experiments, and wrote the manuscript. AA, MT, and MA supervised the research activity. MT and MA reviewed the manuscript. DG, GC, GS, and FP implemented the SWHA framework and the analytical application.

Funding Open access funding provided by Università degli Studi di Torino within the CRUI-CARE Agreement.

Declarations

Conflict of interest The authors declare no Conflict of interest.

References

1. Wu M-W, Lin Y-D (2001) Open source software development: an overview. Computer 34(6):33–38. https://doi.org/10.1109/2.928619
2. Bordeleau F, Meirelles P, Sillitti A (2019) Fifteen years of open source software evolution. In: Bordeleau F, Sillitti A, Meirelles P, Lenarduzzi V (eds) Open source systems. Springer, Cham, pp 61–67. https://doi.org/10.1007/978-3-030-20883-7_6
3. Kritikos A, Stamelos I (2023) A resilience-based framework for assessing the evolution of open source software projects. J Softw Evolut Process. https://doi.org/10.1002/smr.2597
4. GitHub: Octoverse 2022: 10 years of tracking open source. https://github.blog/2022-11-17-octoverse-2022-10-years-of-tracking-open-source/. Accessed on 28 09 2023 (2022)
5. European Commission: the economic and social impact of software and services on competitiveness and innovation. https://digital-strategy.ec.europa.eu/en/library/economic-and-social-impact-software-and-services-competitiveness-and-innovation. Accessed on 28 09 2023 (2017)
6. Hat R (2022) What is open source software? https://www.redhat.com/en/topics/open-source/what-is-open-source-software. Accessed on 28 09 2023

7. Di Cosmo R, Zacchiroli S (2017) Software Heritage: Why and how to preserve software source code. In: iPRES 2017: 14th International Conference on Digital Preservation, Kyoto, Japan. https://www.softwareheritage.org/wp-content/uploads/2020/01/ipres-2017-swh.pdf
8. Antelmi A, Torquati M, Corridori G, Gregori D, Polzella F, Spinatelli G, Aldinucci M (2023) The SWH-Analytics Framework. In: Proceedings of the 2nd Italian Conference on Big Data and Data Science (ITADATA2023), 2023. CEUR Workshop Proceedings, vol. 3606, pp. 1–6. CEUR-WS
9. Di Cosmo R, Zacchiroli S (2016) Software Heritage. https://www.softwareheritage.org. Accessed on 28 09 2023
10. Abramatic J-F, Di Cosmo R, Zacchiroli S (2018) Building the Universal Archive of Source Code. Commun ACM 61(10):29–31. https://doi.org/10.1145/3183558
11. Rossi D, Zacchiroli S (2022) Geographic diversity in public code contributions: an exploratory large-scale study over 50 years. In: Proceedings of the 19th International Conference on Mining Software Repositories. MSR'22, pp. 80–85. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3524842.3528471
12. Zacchiroli S (2021) Gender differences in public code contributions: a 50-year perspective. IEEE Softw 38(2):45–50. https://doi.org/10.1109/MS.2020.3038765
13. Rossi D, Zacchiroli S (2022) Worldwide gender differences in public code contributions (and how they have been affected by the COVID-19 pandemic). In: 2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Society (ICSE-SEIS), pp. 172–183. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3510458.3513011
14. Zacchiroli S (2022) A large-scale dataset of (open source) license text variants. In: Proceedings of the 19th International Conference on Mining Software Repositories. MSR'22, pp. 757–761. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3524842.3528491
15. Pietri A, Rousseau G, Zacchiroli S (2020) Forking without clicking: on how to identify software repository forks. In: Proceedings of the 17th International Conference on Mining Software Repositories. MSR'20, pp. 277–287. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3379597.3387450
16. Lorentz V, Di Cosmo R, Zacchiroli S (2023) The popular content filenames dataset: deriving most likely filenames from the software heritage archive. https://inria.hal.science/hal-04171177
17. Pietri A (2021) Organizing the graph of public software development for large-scale mining. PhD thesis, Université Paris Cité
18. Merkle RC (1988) A digital signature based on a conventional encryption function. In: Pomerance, C. (ed.) Advances in Cryptology—CRYPTO '87, pp. 369–378. Springer, Berlin. https://doi.org/10.1007/3-540-48184-2_32
19. Pietri A, Spinellis D, Zacchiroli S (2019) The software heritage graph dataset: public software development under one roof. In: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories. MSR'19, pp. 138–142. IEEE Press, Piscataway, NJ, USA. https://doi.org/10.1109/MSR.2019.00030
20. Software Heritage: Software Heritage documentation—data model. https://docs.softwareheritage.org/devel/swh-model/data-model.html. Accessed on 06/03/2024 (2018)
21. Kapitsaki GM, Tselikas ND, Kyriakou K-ID, Papoutsoglou M (2022) Help me with this: a categorization of open source software problems. Inf Softw Technol 152:107034. https://doi.org/10.1016/j.infsof.2022.107034
22. Hemel A, Kalleberg KT, Vermaas R, Dolstra E (2011) Finding software license violations through binary code clone detection. In: Proceedings of the 8th Working Conference on Mining Software Repositories. MSR '11, pp. 63–72. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/1985441.1985453
23. Hemel A, Kalleberg KT, Vermaas R, Dolstra E (2021) Finding software license violations through binary code clone detection: a retrospective. SIGSOFT Softw Eng Notes 46(3):24–25. https://doi.org/10.1145/3468744.3468752
24. Gobeille R (2008) The FOSSology Project. In: Proceedings of the 2008 International Working Conference on Mining Software Repositories. MSR'08, pp. 47–50. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/1370750.1370763
25. Kapitsaki GM, Charalambous G (2021) Modeling and recommending open source licenses with findosslicense. IEEE Trans Software Eng 47(5):919–935. https://doi.org/10.1109/TSE.2019.2909021

26. Azhakesan A, Paulisch F (2020) Sharing at scale: an open-source-software-based license compliance ecosystem. In: 2020 IEEE/ACM 42nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), pp 130–131
27. Coughlan S (2020) Standardizing open source license compliance with OpenChain. Computer 53(11):70–74. https://doi.org/10.1109/MC.2020.3016107
28. Kapitsaki GM, Kramer F, Tselikas ND (2017) Automating the license compatibility process in open source software with SPDX. J Syst Softw 131:386–401. https://doi.org/10.1016/j.jss.2016.06.064
29. Harutyunyan N (2020) Managing Your Open Source Supply Chain-Why and How? Computer 53(6):77–81. https://doi.org/10.1109/MC.2020.2983530
30. Wolter T, Barcomb A, Riehle D, Harutyunyan N (2023) Open source license inconsistencies on GitHub. ACM Trans Softw Eng Methodol. https://doi.org/10.1145/3571852
31. Feng M, Mao W, Yuan Z, Xiao Y, Ban G, Wang W, Wang S, Tang Q, Xu J, Su H, Liu B, Huo W (2019) Open-source license violations of binary software at large scale. In: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp 564–568. https://doi.org/10.1109/SANER.2019.8667977
32. Almeida DA, Murphy GC, Wilson G, Hoye M (2017) Do Software Developers understand open source licenses? In: 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC), pp 1–11. https://doi.org/10.1109/ICPC.2017.7
33. Almeida DA, Murphy GC, Wilson G, Hoye M (2019) Investigating whether and how software developers understand open source software licensing. Empir Softw Eng 24(1):211–239. https://doi.org/10.1007/s10664-018-9614-9
34. Lopes CV, Maj P, Martins P, Saini V, Yang D, Zitny J, Sajnani H, Vitek J (2017) DéjàVu: a map of code duplicates on GitHub. Proc ACM Program Lang. https://doi.org/10.1145/3133908
35. Rousseau G, Di Cosmo R, Zacchiroli S (2020) Software provenance tracking at the scale of public source code. Empir Softw Eng 25(4):2930–2959. https://doi.org/10.1007/s10664-020-09828-5
36. Moraes JP, Polato I, Wiese I, Saraiva F, Pinto G (2021) From one to hundreds: multi-licensing in the JavaScript ecosystem. Empir Softw Eng 26(3):39. https://doi.org/10.1007/s10664-020-09936-2
37. Makari IS, Zerouali A, De Roover C (2022) Prevalence and evolution of license violations in npm and RubyGems dependency networks. In: Reuse and Software Quality, pp. 85–100. Springer, Cham. https://doi.org/10.1007/978-3-031-08129-3_6
38. Golubev Y, Eliseeva M, Povarov N, Bryksin T (2020) A study of potential code borrowing and license violations in Java projects on GitHub. In: Proceedings of the 17th International Conference on Mining Software Repositories. MSR'20, pp. 54–64. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3379597.3387455
39. Linux FOundation: What is an SBOM? https://www.linuxfoundation.org/blog/blog/what-is-an-sbom. Accessed on 04 03 2024 (2021)
40. Xia B, Bi T, Xing Z, Lu Q, Zhu L (2023) An empirical study on software bill of materials: where we stand and the road ahead. In: Proceedings of the 45th International Conference on Software Engineering. ICSE '23, pp 2630–2642. IEEE Press. https://doi.org/10.1109/ICSE48619.2023.00219
41. Nocera S, Romano S, Di Penta M, Francese R, Scanniello G (2023) Software bill of materials adoption: a mining study from GitHub. In: 2023 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp 39–49. https://doi.org/10.1109/ICSME58846.2023.00016
42. Zahan N, Lin E, Tamanna M, Enck W, Williams L (2023) Software bills of materials are required. are we there yet?. IEEE Secur Privacy 21(2):82–88. https://doi.org/10.1109/MSEC.2023.3237100
43. Gamalielsson J, Lundell B (2017) On licensing and other conditions for contributing to widely used open source projects: an exploratory analysis. In: Proc. of the 13th Int. Symp. on Open Collaboration. OpenSym'17. ACM, NY, USA. https://doi.org/10.1145/3125433.3125456
44. Software Freedom Law Center: On Behalf of BusyBox Developers, SFLC Files First Ever U.S. GPL Violation Lawsuit. https://softwarefreedom.org/news/2007/sep/20/busybox/. Accessed on 03 10 2023 (2007)
45. Software Freedom Law Center: BusyBox Developers and Monsoon Multimedia Agree to Dismiss GPL Lawsuit. https://softwarefreedom.org/news/2007/oct/30/busybox-monsoon-settlement/. Accessed on 19 Jan 2024 (2007)
46. New York Times: OpenAI Says New York Times Lawsuit Against It Is 'Without Merit'. https://www.nytimes.com/2024/01/08/technology/openai-new-york-times-lawsuit.html. Accessed on 19 Jan 2024 (2024)

47. Joseph Saveri Law Firm: GitHub and Copilot Intellectual Property Litigation. https://www.saverilawfirm.com/our-cases/github-copilot-intellectual-property-litigation. Accessed on 19 Jan 2024 (2023)
48. Mathur A, Choudhary H, Vashist P, Thies W, Thilagam S (2012) An Empirical Study of License Violations in Open Source Projects. In: 2012 35th annual IEEE software engineering workshop, pp. 168–176. https://doi.org/10.1109/SEW.2012.24
49. Antelmi A (2024) SWHA-license-checker-app. https://zenodo.org/records/10801517. Accessed on 10 03 2024. https://doi.org/10.5281/zenodo.10801517
50. Kruskal WH, Wallis WA (1952) Use of ranks in one-criterion variance analysis. J Am Stat Assoc 47(260):583–621
51. Stack Overflow: 2023 Developer Survey. https://survey.stackoverflow.co/2023/. Accessed on 28 Jan 2024 (2023)

Authors and Affiliations

Alessia Antelmi[1,3] · Massimo Torquati[2,3] · Giacomo Corridori[5] · Daniele Gregori[4] · Francesco Polzella[5] · Gianmarco Spinatelli[5] · Marco Aldinucci[1,3]

✉ Alessia Antelmi
alessia.antelmi@unito.it

Massimo Torquati
massimo.torquati@unipi.it

Giacomo Corridori
g.corridori@zerodivision.it

Daniele Gregori
daniele.gregori@e4company.com

Francesco Polzella
f.polzella@zerodivision.it

Gianmarco Spinatelli
g.spinatelli@zerodivision.it

Marco Aldinucci
marco.aldinucci@unito.it

1 Department of Computer Science, University of Turin, Turin, Italy

2 Department of Computer Science, University of Pisa, Pisa, Italy

3 HPC-KTT National Lab, CINI, Rome, Italy

4 E4 Computer Engineering SpA, Scandiano, Reggio Emilia, Italy

5 Zerodivision Systems Srl, Pisa, Italy