

Università degli Studi di Salerno

Dipartimento di Informatica

Dottorato di Ricerca in Informatica
Curriculum «Internet of Things and Smart Technologies»
XXXVIII Ciclo

Tesi di Dottorato / Ph.D. Thesis

Bill of Materials: Understanding and Enhancing SBOM and AIBOM to Support the Management of the Supply Chain of Software and Artificial Intelligence Systems

Sabato NOCERA

Supervisors: **Prof. Rita FRANCESE**
Prof. Simone ROMANO

PhD Program Director: **Prof. Andrea DE LUCIA**

A.A 2024/2025

ABSTRACT

Software Bills of Materials (SBOMs) and Artificial Intelligence Bills of Materials (AIBOMs) are emerging as critical support for managing the supply chain of software and Artificial Intelligence (AI) systems by improving transparency, traceability, compliance, and more. While SBOMs offer structured, machine-readable inventories of software components, AIBOMs extend these principles to AI and AI-enabled systems by documenting models and datasets. However, we still know little about the adoption and practical usage of both SBOM and AIBOM.

This thesis investigates the current state of practice of SBOM and AIBOM to support the management of the supply chain of software and AI systems, and contributes to improving the tooling of SBOM and AIBOM. Through a combination of qualitative and quantitative methods—including semi-structured interviews with practitioners, mining of open-source repositories, and a multivocal literature review—this research identifies low awareness and limited adoption of SBOM and AIBOM, and significant technical limitations in existing SBOM and AIBOM generators.

To improve support for the generation of SBOM and AIBOM, two tools were developed and empirically evaluated. AIRBORNE enhances SBOMs with fine-grained provenance data by linking software artifacts to reused content from Stack Overflow. ALOHA generates AIBOMs by extracting and structuring information from Hugging Face, a major platform for sharing AI models and datasets. Both tools demonstrated the feasibility of enriching SBOMs and AIBOMs through deeper software composition analysis.

The findings of this thesis offer practical and research-oriented implications. For practitioners—especially software suppliers, choosers, and operators—adopting SBOMs and AIBOMs is essential for managing the software supply chain. Suppliers should integrate automated SBOM/AIBOM generation into their pipelines, ensuring compliance with guidelines, standards, and regulations. Choosers should demand complete and accurate SBOMs/AI-

BOMs during procurement, while operators should incorporate them into CI/CD pipelines for continuous risk monitoring. For researchers, future work should focus on improving the support for the generation of SBOM and AIBOM, enhancing their accuracy and coverage, improving the quality of supporting data sources, and tracking software provenance more effectively.

CONTENTS

I Problem Statement

1 Introduction 3

2 Background and Related Work 11
- 2.1 Software Supply Chains 11
  - 2.1.1 Empirical Studies on Software Supply Chains 12
- 2.2 Software Bills of Materials 15
  - 2.2.1 Empirical Studies on Software Bills of Material 22
- 2.3 Artificial Intelligence Bill of Materials 24
  - 2.3.1 Empirical Studies on Artificial Intelligence Bills of Material 25
- 2.4 Software Composition Analysis 26
  - 2.4.1 Empirical Studies on Software Composition Analysis 28

II Studies on Software Bill of Materials

3 Interview Study on the Adoption of SBOMs among Italian Software Practitioners 31
- 3.1 Study Design 31
  - 3.1.1 Study Protocol 31
  - 3.1.2 Study Participants 35
  - 3.1.3 Data Analysis 36
- 3.2 Results 38
  - 3.2.1 Regulations 38
  - 3.2.2 Component Inventory 40
  - 3.2.3 Component Updates 43
  - 3.2.4 Licenses 45
  - 3.2.5 Provenance 46
- 3.3 Discussion 49
  - 3.3.1 Threats to Validity 50

4 Mining Software Repositories Study on the Adoption of SBOMs in Open-source Software Systems hosted on GitHub 53
- 4.1 Study Design 53

4.1.1 Research Questions 54
4.1.2 Study Context and Planning 55
4.1.3 Data Analysis 56
4.2 Results 62
4.2.1 RQ1. Are SBOMs Adopted by Open-source Software Systems? 62
4.2.2 RQ2. What is the Adoption Trend of SBOMs by Open-source Software Systems and their Owners? 65
4.2.3 RQ3. How do Open-source Systems Distribute SBOMs? 67
4.2.4 RQ4. How Frequently SBOMs Get Updated within Open-source Software Systems? 68
4.2.5 RQ5. To What Extent do SBOMs Comply with the Standards? 69
4.2.6 RQ6. What are the Pieces of Information Reported by the Found SBOMs? 72
4.3 Discussion 79
4.3.1 Threats To Validity 89
5 Empirical Evaluation of GitHub's SBOM Generator 93
5.1 Study Design 94
5.1.1 Research Questions 94
5.1.2 Study Context and Planning 95
5.2 Results 100
5.2.1 RQ1. To What Extent Does GitHub's Dependency Graph Provide Accurate Information? 100
5.2.2 RQ2. What Are the Likely Causes for the Error Cases Occurring in GitHub's Dependency Graph? 112
5.3 Discussion 124
5.3.1 Threats To Validity 126
6 AIRBORNE: Augmented fIne-grained softwaRe Bill Of mateRials geNerator for stack ovErflow 129
6.1 AIRBORNE 129
6.1.1 Workflow 130
6.1.2 AIRBORNE in Action 131
6.1.3 AIRBORNE Limitations 134
6.2 Preliminary Empirical Evaluation 134

6.2.1 Study Design 134
6.2.2 Evaluation Results 135
6.2.3 Threats to Validity 137
6.3 Discussion 137

III Studies on Artificial Intelligence Bill of Materials

7 Multivocal Literature Review on AIBOMs 141
7.1 Study Design 142
7.1.1 Planning 142
7.1.2 Conducting the Review 147
7.1.3 Reporting the Review 155
7.2 Results 156
7.2.1 Statistics on the Analyzed Sources 156
7.2.2 RQ1. What are the motivations and benefits of using AIBOMs? 158
7.2.3 RQ2. What are the laws and regulations related to using AIBOMs? 162
7.2.4 RQ3. What elements are included or worth including in AIBOMs? 167
7.2.5 RQ4. What are the use cases where AIBOMs are used? 175
7.2.6 RQ5. What are the approaches, tool categories, and techniques to deal with AIBOMs? 178
7.2.7 RQ6. What are the limitations and challenges of AIBOMs? 183
7.3 Discussion 187
7.3.1 Threats to Validity 191
8 ALOHA: A(IBoM) tooL generatOr for Hugging fAce 195
8.1 ALOHA 195
8.1.1 Workflow 196
8.1.2 ALOHA in Action 199
8.1.3 ALOHA Limitations 201
8.2 Preliminary Empirical Evaluation 202
8.2.1 Study Design 202
8.2.2 Evaluation Results 203
8.2.3 Threats to Validity 206
8.3 Discussion 207

IV Final Remarks
9 Overall Discussion 211
9.1 Linking Results and Objectives 211
9.1.1 O1. Understanding the current state of practice of SBOM and AIBOM to support the management of the supply chain of software and AI systems 211
9.1.2 O2. Improving the tooling of SBOM and AIBOM to support the management of the supply chain of software and AI systems 227
9.2 Implications and Future Directions 228
9.2.1 Software Suppliers, Choosers, and Operators 229
9.2.2 Software Suppliers 229
9.2.3 Software Choosers 232
9.2.4 Software Operators 232
9.2.5 Researchers 232
10 Conclusion 235
A Publications 237

Bibliography 243

Part I

PROBLEM STATEMENT

1 INTRODUCTION

A *supply chain* refers to the linked set of resources and processes involved in the design, development, fabrication, processing, handling, and delivery of a given product [121]. In manufacturing, *Bills of Materials* (*BOMs*) enable suppliers to monitor the resources used to make a product. Whenever an issue is identified in a product, a BOM makes it easy to detect the affected resources and resolve the issue, potentially alerting users. *Software Bills of Materials* (*SBOMs*) apply the same approach to software systems, enabling the identification and management of issues throughout the *software supply chain*. Without SBOMs, the software supply chain lacks transparency, making software systems "black boxes."

An SBOM is a machine-readable record describing the inventory of open-source and proprietary components structuring a software system [185]. These components are identified and profiled through a process named *Software Composition Analysis* (*SCA*) [7]. For instance, an SBOM can detail all information on the libraries on which a software system depends, including their licenses, versions, suppliers, vulnerabilities, and supply chain relationships.

Stakeholders at different stages of software development and usage can benefit from SBOMs [115]. For example, SBOMs assist *software suppliers* in producing their software by reducing code bloat, ensuring compliance with licensing obligations, monitoring vulnerabilities, facilitating code reviews, and preventing the inclusion of banned components. SBOMs are also valuable to *software choosers* to assess security, verify sourcing, ensure policy compliance, and evaluate claims made about the software during development, acquisition, or procurement. Furthermore, *software operators* can leverage SBOMs for the configuration, maintenance, and administration of software systems, enabling more informed, risk-based decisions.

SBOMs are expected to be widely adopted by any organization developing or maintaining software systems, both in industrial

and open-source contexts. In 2020, the European Union Agency for Cybersecurity (ENISA) delineated a series of guidelines for securing the Internet of Things (IoT), which include the release of SBOMs for IoT devices [57]. In 2021, the United States Federal Government, per President Biden's Executive Order 14028 on Improving the Nation's Cybersecurity, laid down that any organization releasing software systems to federal agencies must provide SBOMs for the released software systems [186]. The following year, in 2022, the European Commission proposed the Cyber Resilience Act [56], a cybersecurity regulation outlining that software producers must document the vulnerabilities and components of their software systems with SBOMs. The Cyber Resilience Act was then adopted by the Council of the European Union in 2024. Starting from 2023, GitHub—the leading version control platform for personal and professional development [168]—introduced a new feature to export SBOMs from the dependency graph of the software systems hosted on the platform [75].

As Artificial Intelligence (AI) and Machine Learning (ML) become widely integrated into software systems across different domains [8, 55, 147, 192], new challenges emerge in managing AI components involved in the software supply chain. For example, the EU AI Act [58] mandates systems with embedded AI to be assessed based on risk levels throughout their lifecycle, with stringent requirements for high-risk applications to ensure transparency, accountability, and compliance with laws. As a result, conventional notions of the software supply chain require re-examination and adaptation to account for the unique characteristics of AI components from the perspective of the AI supply chain [29]. Building on the principles of SBOM, the AI Bill of Materials (AIBOM) [195, 196] has been proposed to provide a structured inventory of AI components, including AI models and datasets.

While the adoption of SBOMs to manage the software supply chain has primarily been driven by public administrations, their uptake remains limited [171, 198]. In this respect, there is a **lack of empirical research on how software stakeholders—across both industry and open-source communities—use SBOMs and AIBOMs**, as well as on **the challenges they face and the sup-**

Table 1.1: Main contributions of the thesis.

	SBOM	AIBOM
O1	Interview study on the adoption of SBOMs among Italian software practitioners [156] Mining software repositories study on the adoption of SBOMs in open-source software systems hosted on GitHub [157, 158] Empirical evaluation of GitHub's SBOM generator [44, 45]	Multivocal literature review on AIBOMs [155]
O2	Development and evaluation of a fine-grained SBOM generator [127]	Development and evaluation of an AIBOM generator [51]

port they require. Based on the aforementioned gaps, this thesis pursues the following overarching goal: ***understanding and enhancing SBOM and AIBOM to support the management of the supply chain of software and AI systems***. Based on this overarching goal, two objectives were formulated to guide the study.

O1. *Understanding the current state of practice of SBOM and AIBOM to support the management of the supply chain of software and AI systems.*

This objective aims to better understand the extent to which SBOMs and AIBOMs are used by stakeholders in the management of the software supply chain, along with the current challenges, unmet needs, and areas where further support is required.

O2. *Improving the tooling of SBOM and AIBOM to support the management of the supply chain of software and AI systems.*

This objective concerns the development and empirical evaluation of tools for generating SBOMs and AIBOMs. Currently, SBOM generators are largely limited to representing coarse-grained components, such as software libraries and packages, while tools for generating AIBOMs are virtually nonexistent. The aim is to enhance existing SCA techniques to generate SBOMs/AIBOMs describing a broader range of component types, thus improving the support for managing the software supply chain.

Table 1.1 summarizes the main contributions of the thesis. The studies that were conducted to address O1 are presented below, distinguishing between SBOM and AIBOM.

SBOM. The current state of practice regarding the management of the software supply chain using SBOMs was explored from two perspectives: industry and open-source. In the industry context, an **interview study** [156] was conducted to assess how Italian software practitioners are adopting SBOMs to manage their software supply chains, focusing particularly on regulatory compliance, dependency management, provenance, and licensing practices. The study found that, despite considerable attention to software supply chain challenges, Italian software practitioners have limited knowledge of SBOMs and related regulations. As for the open-source context, a **mining software repositories study** [157, 158] was carried out to investigate the adoption of SBOM generators by open-source software systems hosted on GitHub. This investigation included adoption trends, distribution patterns, update frequency, standard compliance, and the reporting of data fields in SBOMs. The findings revealed that while SBOM adoption is still low, it is on the rise. Only a small fraction of SBOMs contain the minimum or recommended information, with some even being non-compliant with established standards, and none reporting vulnerabilities. To better understand the limitations of current SBOM generators, an **empirical evaluation of GitHub's SBOM generator** [44, 45] was conducted. This generator leverages GitHub's dependency graph, a built-in SCA tool that maps the dependencies of the systems hosted on GitHub. The evaluation provided insights into the accuracy of the information generated by GitHub's dependency graph and identified the likely causes of errors. The results of this evaluation show that GitHub's SBOM generator yields inaccuracies due to the oversimplified processing of dependency data during SCA.

AIBOM. The current state of practice regarding the management of the software supply chain using AIBOMs was investigated through a **Multivocal Literature Review** (MLR) [155]. It focused on the motivations, benefits, laws, regulations, elements, use cases, approaches, tools, techniques, limitations, and challenges of using AIBOMs. The results of this MLR indicate that AIBOMs improve transparency and traceability—

as well as security, safety, and regulatory compliance—by documenting AI models, datasets, and their relationships. However, several concerns remain unaddressed, such as immature generation and consumption tools, data source availability, poor interoperability with existing infrastructures, and limited stakeholder awareness.

The studies conducted to address O2 build upon the insights gained from the studies mentioned above. These are described below, distinguishing between SBOM and AIBOM.

SBOM. To better support SBOMs in the management of the software supply chain, a **fine-grained SBOM generator** [127] was developed and evaluated, namely AIRBORNE (Augmented fIne-grained softwaRe Bill Of mateRials geNerator for stack ovErflow). It augments traditional SBOMs with information on source code provenance from Stack Overflow, one of the most popular Question-and-Answer (Q&A) platforms about software development. The tool detects self-admitted Stack Overflow references within the files of a given software system, potentially linking these references to code snippets reused from Stack Overflow posts. The evaluation revealed that AIRBORNE can successfully generate fine-grained SBOMs by incorporating explicit reuse declarations of Stack Overflow.

AIBOM. To better support AIBOMs in the management of the software supply chain, an **AIBOM generator** [51] was developed and evaluated, namely ALOHA (AIBOM tooL generatOr for Hugging fAce). It generates AIBOMs by mining information from AI models and datasets hosted on Hugging Face, the central platform for collaboration among the open-source AI community. The evaluation revealed that ALOHA can successfully generate AIBOMs by retrieving and structuring data from Hugging Face.

Summing up, the results of this thesis indicate that, although the adoption of SBOMs is increasing, it remains limited. This slow uptake can be primarily attributed to a lack of awareness regarding SBOM principles and software supply chain regulations, as well as the immaturity of current tools for SBOM generation and

consumption. Similarly, AIBOM has gained traction as a means to support supply chains with AI components, but it still lacks mature tooling. The results of this thesis also indicate that SBOMs and AIBOMs can be improved to better support the management of software supply chains by enhancing the underlying SCA to accurately identify and profile a broader range of component types.

THESIS STRUCTURE. The thesis is structured into four parts:

1. *Part i* defines the research problem addressed in the thesis and includes *Chapter 2*, which focuses on relevant background information and related work.

2. *Part ii* presents studies on SBOMs, and includes four chapters:

   Chapter 3 reports the interview study on the adoption of SBOMs among Italian software practitioners. This chapter is based on the paper entitled "*If it's not SBOM, then what? How Italian Practitioners Manage the Software Supply Chain*" [156], published in the proceedings of the 40th IEEE International Conference on Software Maintenance and Evolution.

   Chapter 4 presents the mining software repositories study on the adoption of SBOM in open-source software systems hosted on GitHub. This chapter is based on the paper entitled "*On the adoption of software bill of materials in open-source software projects*" [158], published in the Journal of Systems and Software, which extends the work presented in "*Software Bill of Materials Adoption: A Mining Study from GitHub*" [156], published in the proceedings of the 39th IEEE International Conference on Software Maintenance and Evolution.

   Chapter 5 offers the empirical evaluation of GitHub's SBOM generator. This chapter is based on the paper entitled "*An empirical study on the accuracy of GitHub's dependency graph and the nature of its inaccuracy*" [45], published in the Information and Software Technology, which extends the work presented in "*On the Accuracy*

of GitHub's Dependency Graph" [44], published in the proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering.

Chapter 6 describes the development and evaluation of the fine-grained SBOM generator. This chapter is based on the paper entitled "*AIRBORNE: Augmented fIne-grained softwaRe Bill Of mateRials geNerator for stack ovErflow*" [127], submitted to the 48th IEEE/ACM International Conference on Software Engineering.

3. *Part iii* presents studies on AIBOMs, and includes two chapters:

   Chapter 7 provides the MLR on AIBOM. This chapter is based on the paper entitled "*What We Know about AIBOMs: Results from a Multivocal Literature Review on Artificial Intelligence Bill of Materials*" [155], submitted to the ACM Transactions on Software Engineering and Methodology (currently at the second round of revision).

   Chapter 8 covers the development and evaluation of the AIBOM generator. This chapter is based on the paper entitled "*ALOHA: A(IBoM) tooL generatOr for Hugging fAce*" [51], published in the proceedings of the 29th International Conference on Evaluation and Assessment in Software Engineering.

4. *Part iv* provides an overall discussion of the results from the individual studies in *Chapter 9* and concludes the thesis in *Chapter 10*.

2

BACKGROUND AND RELATED WORK

This chapter provides background information and discusses related work on software supply chain, SBOM, AIBOM, and SCA. For each of these topics, the chapter first introduces the necessary background to establish a foundational understanding, followed by a review of related work to contextualize current research and practices.

2.1 SOFTWARE SUPPLY CHAINS

A software supply chain consists of software that is increasingly built using layers of reusable abstractions, such as libraries, frameworks, cloud infrastructure, and AI components [191]. In this scenario, software systems depend on and build upon other existing software systems [191]. According to the Census II investigation by the Harvard Laboratory for Innovation Science and the open-source Security Foundation, 70-90% of modern software products depend on open-source software [178].

A primary concern for software supply chains is security. The OWASP Top 10—a document representing a broad consensus about the most critical security risks, regularly updated by the OWASP Foundation—reports the usage of outdated and vulnerable components as one of the most critical security risks [128]. For example, Heartbleed, a vulnerability disclosed in 2014 in the popular OpenSSL cryptography library, costed businesses at least 500 million USD [207]. Another example is Log4Shell, a vulnerability disclosed in 2021 in the popular Apache Log4j logging framework, which affected more than 35,000 Java packages (amounting to over 8% of the Maven ecosystem), and fixing it in all the affected software products will take years [79].

The growing pervasiveness of AI is accelerating the proliferation of AI components into software supply chains. In this respect, two specific categories of software systems can be distinguished: AI systems and AI-enabled systems. An *AI system* is one in which

AI constitutes the core functionality, performing tasks such as perception, reasoning, decision-making, or learning (*e.g.,* computer vision systems) [149]. In contrast, an *AI-enabled system* incorporates AI as a supporting or augmenting component, rather than as its primary function (*e.g.,* a customer relationship management system enhanced with AI-based predictive analytics) [133]. Both AI and AI-enabled systems differ fundamentally from traditional software systems, particularly in how their behavior is shaped by AI components [133]. For example, differently from traditional software that once deployed remains static in its behavior, AI models used by such systems may evolve as they process new data [196]. As a result, conventional notions of the software supply chain require re-examination and adaptation to account for the unique characteristics of AI components from the perspective of the AI supply chain [29].

2.1.1 *Empirical Studies on Software Supply Chains*

Several empirical studies investigated the security of software supply chains. Derr *et al.* [47] analyzed Android applications and found that almost all vulnerable components could be updated without changing the source code. They also found that vulnerable components are not updated to avoid concerns that could compromise the functioning software (*e.g.,* incompatibilities, high integration effort), but also due to a lack of awareness regarding available updates. Decan *et al.* [46] studied the impact of vulnerabilities in npm components and found that the number of new vulnerabilities and affected components grows over time. The authors also observed that it takes a long time to discover and fix vulnerabilities, and a large fraction of components dependent on vulnerable ones are not updated even when a fix is available. Zapata *et al.* [200] manually inspected 60 projects adopting a vulnerable version of the `ws`, `angular`, and `marked` components and found that most of them were safe as they did not use the vulnerable code. Zimmerman *et al.* [203] analyzed security concerns in the npm ecosystem by focusing on the impact that vulnerable components can have. The results indicate that vulnerable components could impact large parts of the entire ecosystem and that a lack of security maintenance leads to persistent vulnerabilities.

Prana *et al.* [143] analyzed vulnerabilities in open-source components used by 450 software projects written in Java, Python, and Ruby. The authors found that the most common vulnerabilities related to "Denial of Service" and "Information Disclosure" were of a medium severity level; moreover, components were rarely updated or changed by project owners, despite the availability of updates. Alfadel *et al.* [6] investigated the propagation and life span of vulnerabilities in the Python ecosystem. Their results seem to suggest that vulnerabilities in the packages increase over time and can take more than three years to be found and seven months to be fixed by the dependent.

Levy [104] asserted that open-source software can be more challenging to secure than proprietary software because of its development environment, which is more complex to a large extent. For example, organizations like Linux package the software by commonly accepting code from others and bundling it with other packages. The resulting software is then distributed through websites. This complex web of interrelated parties creates several attack points in the software supply chain. The author highlighted the need for technologies to secure the supply chains of open-source software. To enable sharing among industry practitioners having practical experiences and challenges with software supply chain security, Enck and Williams [53] conducted two industry and one government software summits (for a total of 30 organizations across the three summits). The result of these summits is a list of five software supply chain security challenges. Among these five top challenges, there is "leveraging SBOM for security." While participants had widely divergent opinions, they agreed that efforts at establishing standards and requirements for SBOMs have the potential to lay the groundwork for innovative security enhancements.

Hartmann and Trew [84] proposed the concept of *Context Variability* to model multiple product lines and to support software supply chains. In detail, this model facilitates the merging of feature models during staged configuration, where variability originates from different product types, geographic regions, and customers. Preliminary evaluations of the model suggest that Context Variability is both intuitive and easy to apply, even with existing, commercially available variability management

tools. Lamb and Zacchiroli [100] examined *Reproducible-Builds* (*R-B*) from the perspective of software professionals. R-B is an approach that can determine whether generated binaries correspond to a given source code. The main idea is that when a given source tree reliably produces bit-for-bit identical results, the resulting artifacts can be considered trustworthy. The results of their study indicate that uncontrolled build inputs occur when tool chains allow the build process to be affected by the surrounding environment. Similar to Lamb and Zacchiroli [100], Vu *et al.* [188] proposed an approach, motivated by the intuition behind R-B, to detect injected code in typosquatting and hijacked packages. In particular, their approach detects code injected into software packages by comparing their distributed artifacts (PyPI) with the source code repository (GitHub). The approach has been preliminarily evaluated on two datasets of known malicious packages and the top ten most downloaded packages. Such an approach captures the known malicious packages and reduces the review effort for distributed software releases by 97%.

There are also studies related to analyzing dependencies in the software ecosystem and their evolution. For example, Bavota *et al.* [18] observed that the developers of the Apache Java ecosystem try to lower the impact that dependency upgrades would have when triggered by major releases and critical bug fixes. As for dependencies in R and npm ecosystems, Bogart *et al.* [23] revealed that developers do not use systematic approaches to cope with changes. Later, Zahan *et al.* [199] analyzed the metadata of 1.63 million JavaScript npm packages and proposed six alerts of security weaknesses (*e.g.*, install scripts and maintainer accounts associated with an expired email domain) in a software supply chain. Zahan *et al.* [199] identified 11 malicious packages from the install scripts signal and found 2,818 maintainer email addresses associated with expired domains (allowing the hijacking of 8,494 packages). Zahan *et al.* [199] also obtained qualitative feedback on link signals through a survey responded by 470 npm package developers. One of the most important results of the survey is that the developers would want to be notified about weak link signals before using third-party packages.

Regarding software supply chains involving AI components, Widder and Nafus [190] explored the ethical challenges faced by

27 AI professionals when applying AI ethics guidelines. They observed a phenomenon they describe as "dislocated accountability," which emerges from the fragmentation of the development process into smaller tasks. This fragmentation often obscures the broader context, diffusing ethical responsibility across various actors in the supply chain. The authors argue that conventional tools—such as ethical checklists and static guidelines—could be enhanced by adopting an approach of "situated accountability," which emphasizes the specific relationships and responsibilities embedded within and extending beyond the AI supply chain.

2.2 SOFTWARE BILLS OF MATERIALS

An SBOM describes in a structured and machine-readable format (*e.g.,* XML, JSON) the inventory of components used in a software product, including third-party open-source and proprietary components [185]. The need to develop a mechanism such as SBOM is fostered by the peculiar nature of modern software products, which no longer consist of only proprietary code.

Despite having received notable attention from policymakers [56, 57, 186], the software industry has yet to implement SBOMs fully [198]. In this respect, the United States Government was the first authority to show interest in establishing guidelines to facilitate the adoption of SBOM. In particular, President Biden's Executive Order 14028 on Improving the Nation's Cybersecurity [186] mandated the United States Department of Commerce and the National Telecommunications and Information Administration (NTIA) to publish the minimum elements for SBOMs [185], which comprise the following interrelated areas:

- *Data Fields*, indicating the baseline information about each component to enable its tracking across the software supply chain and its mapping to other sources of data (*e.g.,* vulnerability or license databases). In detail, NTIA's minimum data fields for SBOMs are:
  - Supplier name, which represents the entity responsible for creating, defining, and identifying components.
  - Component name, which describes the name given to a component by the original supplier.

  - Component version, which details the identifier the supplier uses to specify different versions of a component.
  - Other unique identifiers, which indicate other component identifiers that may reference relevant databases.
  - Dependency relationship, which describes the dependency relationships among components.
  - Author of SBOM data, which represents the name of the entity that creates the SBOM.
  - Timestamp, which records the date and time of the SBOM generation.

- *Automation Support*, related to software tools for producing or consuming SBOMs.

- *Practices and Processes*, devising strategies to integrate SBOMs into the software development life cycle—*e.g.,* a new SBOM must be produced and made available to those who need it whenever a component is updated or new information on that component becomes available like disclosed vulnerabilities.

In addition to the minimum elements for SBOMs, the United States Department of Commerce and NTIA [185] published *recommended data fields*. They are:

- Component hash, a fixed-size string generated by a hash function from a component's binary or source code, uniquely identifying the component and verifying its integrity.

- Life cycle phase, indicating when (*e.g.,* pre-, post-, or within-building) and from where (*e.g.,* source or binary code) the data reported in the SBOM was collected.

- Other component relationships, indicating kinds of relationship (*e.g.,* derivation or descendant) different from the dependency relationship.

- License information, needed to manage licensing risks due to employing components with incompatible licenses.

As for security, the United States Department of Commerce and NTIA suggest storing vulnerability information in separate data structures linked to SBOMs [185]. This requirement emerges because vulnerability information is dynamic and evolves over time—a software product previously deemed not secure may become insecure as new vulnerabilities are discovered. As such, vulnerability information in SBOMs cannot be assumed to be complete and up-to-date unless very specific conditions and processes are in place [185].

SBOM generation should produce a comprehensive and accurate list of all the components a software product depends on, which might vary depending on the stage of software development (*e.g.*, considering pre- vs. post-build artifacts) [120]. Manual SBOM generation is impractical due to the large number of dependencies a software product can have. Therefore, it is necessary to rely on automated tools, possibly integrated into the development pipeline. For example, some SBOM generators can be plugged into package managers. In such a case, the SBOM generator is declared among the project's dependencies in the package manager's build/lock files. For example, there are plugins to be declared in `pom.xml` files for Java Maven projects, which allow the automatic generation of SBOMs based on the dependencies listed in those `pom.xml` files. As a result, current tools typically capture dependencies only at the package or library level, overlooking finer-grained artifacts such as code snippets [161] copied from Q&A platforms such as Stack Overflow [28, 109]—this may introduce potential licensing conflicts and security risks [90] that remain invisible in current SBOMs.

Multiple standards have been defined to make SBOMs machine-readable and allow their consumption by other tools. An *SBOM standard* is a machine-readable schema conceived to provide a uniform structure for describing the composition of a software product so that the SBOM can be consumed by other tools (*e.g.*, vulnerability or license scanners). As pointed out by Xia *et al.* [195], the most commonly adopted standards are *SPDX* [154] and *CycloneDX* [38].

SPDX is an open standard with origins in the Linux Foundation. Initially conceived to improve software license compliance, SPDX was then extended to support further use cases like those related

to security. In 2021, SPDX was approved as an ISO/IEC standard [86]. Another open standard for SBOMs is CycloneDX from the OWASP Foundation. Similar to SPDX, it supports different use cases, from software license compliance to security-related ones.

Both SPDX and CycloneDX incorporate existing specifications to identify components, including Package URL (purl), Common Platform Enumeration (CPE), and SWID tags. Indeed, SBOM is the glue that maps external data to software products/components, supporting complex and straightforward use cases (*e.g.,* mapping to vulnerability databases to retrieve information on disclosed vulnerabilities and their mitigations, or ongoing monitoring and risk analysis of third-party components) [185]. The same component can be identified through different kinds of identifiers. Also, a database containing component information might support a certain kind of identifier rather than another. For instance, the National Vulnerability Database (NVD) supports only CPE. It is worth noting that hashes can also serve as software identifiers. However, they are reported in SBOM in specific data fields dedicated to hash values (unlike other component identifiers); also, there are situations when a hash cannot be generated (*e.g.,* partial or complete absence of the binary/source code from which to generate the hash) or require further information on how it will be generated (*e.g.,* while generating a hash from a source file is straightforward, for an executable file there might be some differences based on the packages employed) [185].

SPDX and CycloneDX allow describing virtually the same component information, albeit in different ways and using different terminology. Table 2.1 shows the SPDX and CycloneDX object models, depicting the sections an SBOM must/can have. *SPDX Document Creation Information* and *BOM Metadata Information* are, respectively, the SPDX and CycloneDX sections describing the software product for which the SBOM was produced and how the SBOM was created, by whom, and when. In CycloneDX, the *Components Information* section documents first- and third-party components (including software and hardware components, AI/ML models, source code, and configurations) while the *Services Information* documents the usage of external APIs. The same information (*i.e.,* internal components and external services) can

Table 2.1: Object models of SPDX and CycloneDX.

SPDX	CycloneDX
SPDX Document Creation Information	BOM Metadata Information
Package Information	Components Information
File Information	Services Information
Snippet Information	Dependency Relationships
Other Licensing Information Detected	Compositions
Relationships between SPDX Elements	Vulnerabilities
Information	Formulation
Annotations Information	Annotations
Review Information	Extensions

be virtually described in SPDX through the *Package Information*, *File Information*, and *Snippets Information* sections. In both SPDX and CycloneDX, licensing information can be described along with the declared components. However, SPDX also provides a section named *Other Licensing Information Detected*. *Relationships between SPDX Elements Information* and *Dependency Relationships* are, respectively, the SPDX and CycloneDX sections outlining direct and transitive dependencies involving first- and third-party components, according to a dependency graph representation. CycloneDX provides a *Vulnerabilities* section to document any information on disclosed vulnerabilities within first- and third-party components; similarly, it is possible to document this information in SPDX along with the component information in the *Package Information*, *File Information*, and *Snippets Information* sections. Compared to other formats, CycloneDX potentially allows for a more in-depth description and characterization of vulnerabilities, *e.g.,* to what extent an attacker could exploit a vulnerability in a component. The reason for this better support is that CycloneDX was explicitly conceived to better support vulnerability management in complex software supply chains.

The SPDX and CycloneDX standards define the structure of SBOMs; as for how to report component information, it is necessary to rely on other documents, such as the Department of Commerce and NTIA's guidelines on minimum elements for SBOMs [185] and the mapping of these elements to the SPDX and CycloneDX standards [114]. Indeed, these documents outline

```
{
  "SPDXID" : "SPDXRef-DOCUMENT",
  "spdxVersion" : "SPDX-2.3",
  "creationInfo" : {
    "created" : "2023-12-31T18:48:20Z",
    "creators" : [ "Tool:
    ↪  spdx-maven-plugin"],
    "licenseListVersion" : "3.22"},
  "name" : "Template",
  "dataLicense" : "CC0-1.0",
  "documentNamespace" :
  "http://spdx.org/spdxpackages/template",
  "packages" : [ {
    "SPDXID" : "SPDXRef-gnrtd0",
    "name" : "Template",
    "primaryPackagePurpose" : "LIBRARY",
    "versionInfo" : "1.0-SNAPSHOT"},
    {"SPDXID" : "SPDXRef-gnrtd1",
    "description" : "Apache Log4j 1.2",
    "homepage" :
    ↪  "http://logging.apache.org/log4j/1.2/",
    "licenseDeclared" : "Apache-2.0",
    "name" : "Apache Log4j",
    "originator" : "Organization:Apache
    ↪  Software Foundation",
    "summary" : "Apache Log4j 1.2",
    "versionInfo" : "1.2.17" } ],
  "relationships" : [ {
    "spdxElementId" : "SPDXRef-DOCUMENT",
    "relationshipType" : "DESCRIBES",
    "relatedSpdxElement" :
    ↪  "SPDXRef-gnrtd0"},{
    "spdxElementId" : "SPDXRef-gnrtd0",
    "relationshipType" : "DYNAMIC_LINK",
    "relatedSpdxElement" :
    ↪  "SPDXRef-gnrtd1"}]
}
```


(a) SPDX SBOM.


```
{
  "bomFormat" : "CycloneDX",
  "specVersion" : "1.4",
  "serialNumber" :
  ↪  "urn:uuid:f8b8ab5d-0c68-47d9-82fe-65df75d30544",
  "version" : 1,
  "metadata" : {
    "timestamp" : "2023-12-31T17:46:04Z",
    "tools" : [ {
        "vendor" : "OWASP Foundation",
        "name" : "CycloneDX Maven plugin",
        "version" : "2.7.9"],
    "component" : {
      "group" : "com.example",
      "name" : "Template",
      "version" : "1.0-SNAPSHOT",
      "type" : "library",
      "bom-ref" :
      ↪  "pkg:maven/com.example/Template?type=jar"}},
  "components" : [ {
      "publisher" : "Apache Software Foundation",
      "name" : "log4j",
      "version" : "1.2.17",
      "licenses" : [ { "license" : { "id" :
      ↪  "Apache-2.0" } } ],
      "type" : "library",
      "bom-ref" :
      ↪  "pkg:maven/log4j/log4j@1.2.17?type=jar"} ],
  "dependencies" : [ {
      "ref" :
      ↪  "pkg:maven/com.example/Template?type=jar",
      "dependsOn" : [
      ↪  "pkg:maven/log4j/log4j@1.2.17?type=jar"]
    }, {
      "ref" : "pkg:maven/log4j/log4j@1.2.17?type=jar",
      "dependsOn" : [ ] } ]
}
```


(b) CycloneDX SBOM.

Figure 2.1: Examples of SBOMs generated using the SPDX and CycloneDX plugins for Maven.

how to put SBOMs into practice. For instance, SPDX and CycloneDX standards define sections to report dependencies among components but not the minimum *depth* in the dependency graph; in this respect, according to NTIA [185], an SBOM should contain at least all *direct* dependencies among components, as well as all *transitive* dependencies or sufficient details to retrieve transitive dependencies recursively[1].

The typical file format for SPDX and CycloneDX SBOMs is JSON or XML. However, SPDX SBOMs can also be produced as a plain text file. Figure 2.1 provides examples of two SBOMs in JSON format, one is related to the SPDX standard and the

1 A direct dependency indicates that a component X depends on a component Y, while a transitive dependency indicates that a component X depends on a component Z because there is a component Y such that X depends on Y and Y depends on Z.

other to CycloneDX. Both SBOMs report component information about the same system, *i.e.,* a Java Maven project that imports only one dependency into the `pom.xml` file, namely `Apache Log4j`. The SBOM files were generated using the SPDX and CycloneDX plugins for Maven.

SBOMs can be tailored to describe a wide range of inventory types [31], which has led to conceiving the following specifications:

- *Software-as-a-Service Bill of Materials* (*SaaSBOM*). It offers a comprehensive inventory for cloud-native applications, detailing services, endpoints, data flows, and data classifications. SaaSBOMs complement Infrastructure-as-Code (IaC) by providing a logical representation of a complex system, complete with an inventory of all services, their reliance on other services, endpoint URLs, data classifications, and the directional flow of data between services. The types of services SaaSBOMs can represent include microservices, Service-Oriented Architectures (SOA), Function-as-a-Service (FaaS), and broader System-of-Systems architectures. Optionally, they may also include the underlying software components that constitute each service.

- *Hardware Bill of Materials* (*HBOM*). It extends SBOM capabilities to embedded systems and IoT devices, bridging the gap between traditional *Engineering Bills of Materials* (*EBOM*) for hardware-based systems. This makes HBOMs well-suited for applications in consumer electronics, IoT, industrial control systems, and other embedded environments.

- *Manufacturing Bill of Materials* (*MBOM*). It documents the formulations and processes behind the creation of both software and hardware components.

- The *Cryptography Bill of Materials* (*CBOM*). It describes cryptographic assets and their dependencies, a critical step in enabling the transition to quantum-safe systems. Cryptography can be difficult to manage as it is often deeply embedded within the components that make up software systems. As part of an agile cryptographic approach, organizations

can use CBOMs to gain visibility into the cryptographic assets they rely on, enabling them to assess associated risks and establish a baseline for mitigation efforts.

- The *Operations Bill of Materials* (*OBOM*). It encompasses runtime environments and component configurations, offering a comprehensive, full-stack inventory.
- *Bill of Vulnerabilities* (*BOV*). It supports detailed tracking and communication of vulnerabilities across components. BOVs can represent complex vulnerability details, including the source, references, multiple severity levels, risk ratings, technical descriptions, remediation guidance, and the specific affected software or hardware components along with their versions.
- *Vulnerability Disclosure Report* (*VDR*). It conveys both known and unknown vulnerabilities affecting components and services. VDRs supports the disclosure of vulnerabilities, such as those inherited from third-party and open-source software.
- *Vulnerability Exploitability eXchange* (*VEX*). It describes the extent to which a known vulnerability in a component is actually exploitable within the context of the specific system where the component is used. As a subset of VDR, VEX addresses the common scenario where the mere inclusion of a vulnerable component does not necessarily mean the system is affected. VEXs clearly communicate the exploitability status of vulnerabilities, distinguishing those that pose a real risk from those that do not.
- *AIBOM* (also referred to as *MLBOM* or *DataBOM*). It documents AI components by representing detailed information about AI models, datasets, and their dependencies. Further details about AIBOMs are provided in Section 2.3.

2.2.1 *Empirical Studies on Software Bills of Material*

Linux Foundation's SBOM readiness survey in 2022 [171] with 412 worldwide organizations showed gaps in familiarity with,

production planning for, and consumption of SBOMs. For example, 40% of the organizations were unclear about whether the industry is committed to mandating SBOMs, raising concerns about whether the software industry is abiding by President Biden's Executive Order 14028 on Improving the Nation's Cybersecurity [186].

Xia *et al.* [195] conducted 17 interviews and a survey with 65 respondents to investigate *(i)* the adoption of SBOM in practice, *(ii)* the SBOM tool support, and *(iii)* the main concerns related to the use of SBOMs. Based on the collected results, the authors derived 26 statements describing the SBOM adoption and its challenges. Those are related, for example, to the still limited awareness, the need for better incentives in using SBOMs, a limited consensus on what should be the content of SBOMs, and the presence of sub-optimal and non-standardized tools, especially for what concerns the support for handling vulnerabilities.

Chaora *et al.* [26] performed an ethnographic study analyzing discourse from public meetings held around SBOMs. The discourse occurred from September 20, 2022, to March 20, 2023, during SBOM community engagement meetings facilitated by the Cybersecurity & Infrastructure Security Agency (CISA) in the United States Department of Homeland Security. To promote SBOM adoption, the authors foresee regulations and a publicly recognized agency that audits and validates SBOMs submitted to a common infrastructure. This infrastructure could restrict information on SBOM components to the authorized stakeholders and monitor vulnerabilities within them. The authors also foresee that SBOMs may face other challenges, such as supervising artificial intelligence.

Stalknaker *et al.* [170] conducted a study combining surveys and interviews and aimed to investigate the challenges behind SBOM adoption for five different groups of stakeholders (practitioners familiar with SBOMs, contributors to critical open-source projects, experts of AI and ML, developers of cyber-physical systems, and legal experts), and the ways such challenges could be circumvented. The results of their study, which involved 138 participants, identified 12 different challenges and possible solutions to address them. Important challenges include the need for multi-dimensional SBOM specifications, improving SBOM-related tools,

aiding SBOM verification/consumption, and promoting SBOM adoption.

Kloeg *et al.* [99] analyzed business stakeholder groups directly involved in SBOM production and consumption to determine which groups can drive or inhibit SBOM adoption and why. Through semi-structured interviews, the authors observed that system integrators and software vendors had the highest SBOM adoption potential. In contrast, business-to-business customers and individual developers had the lowest motivation.

Balliu *et al.* [12] compared six state-of-the-art tools that generate CycloneDX SBOMs for Java Maven projects and discovered that these tools capture different sets of dependencies, which might be incorrect or incomplete. The authors encourage new research that deepens the analysis of SBOMs with respect to different SBOM standards and programming languages.

Torres-Arias *et al.* [182] performed a preliminary empirical investigation on NTIA's minimum data fields. They analyzed SPDX SBOMs from an SBOM dataset that was artificially created through different SBOM generators. The authors found that about one percent of the analyzed SBOMs fully complied with NTIA's minimum data fields.

As far as SBOM generators, Yu *et al.* [197] assessed GitHub's dependency graph, Trivy, Syft, and Microsoft SBOM Tool. Yu *et al.* [197] analyzed SBOMs generated from 7,876 open-source projects across nine programming languages. Their results revealed that all SBOM generators suffer from inaccuracies in generating SBOMs. Among the causes of such inaccuracies, Yu *et al.* [197] reported incomplete metadata parsing, limitations in resolving transitive dependencies, limited support for version constraints, and inconsistent package naming conventions and dependency definitions.

2.3 ARTIFICIAL INTELLIGENCE BILL OF MATERIALS

In addition to existing governmental regulations mandating SBOMs and software supply chain management (*e.g.,* [56, 57, 186]), new regulatory frameworks are emerging to govern the use of AI. For example, the EU AI Act [58] introduces a risk-based classification system for systems with embedded AI, requiring

continuous assessment throughout their lifecycle. High-risk applications, in particular, are subject to stringent requirements aimed at ensuring transparency, accountability, and legal compliance.

The behavior of systems with embedded AI components (*e.g.*, AI/ML models) is largely shaped by such components. Ensuring transparency for such systems requires SBOMs also to capture details specific to AI components (*e.g.*, model version, training information, attribution/licensing requirements, and bias/fairness issues). To this end, an extension to SBOM specifications has recently been conceived to document AI components, namely AIBOM [212]. Traditional SBOM standards need to be adapted to serve as an optimal foundation for the generation and consumption of AIBOMs. For example, since version 1.5, released in 2023, CycloneDX introduced fields to track and manage AI components and their associated metadata [20]. This extension, also referred to as ML-BOM [212] or DataBOM [170], allows representing AI-specific components such as:

- AI/ML models documenting architectures, weights, hyperparameters, and configurations;
- Datasets used for training, validation, and testing, ensuring data provenance and integrity;
- Algorithms and frameworks describing the underlying AI frameworks, libraries, and optimization techniques applied;
- Dependencies and external components tracking third-party models, pre-trained embeddings, and associated dependencies, which are critical for vulnerability management and reproducibility.

2.3.1 *Empirical Studies on Artificial Intelligence Bills of Material*

The research community has recently started investigating the need for complementing SBOMs with AIBOMs [145, 170, 195, 196]. In their qualitative study, Xia *et al.* [195] pointed out that the generation of AIBOMs is different from that of traditional SBOMs as AIBOMs include AI/ML-specific data. Xia *et al.* [196] acknowledged that a key difference between SBOMs and AIBOMs is the dynamic nature that the latter should have. While traditional

software systems remain static in their behavior once deployed, the behavior of systems with embedded AI could evolve as they process new data. This difference would require ad-hoc tools for the generation of AIBOMs. Stalnaker *et al.* [170] also conducted a qualitative study in which almost all respondents were unaware of any tool support for the generation of AIBOMs. Although no AIBOM generator was reported, the authors identified as "quasi-AIBOM" generators two tools: DVC [42] and ML-Flow [108]. In their work, Radanliev *et al.* [145] proposed an AIBOM schema and a tool to represent its instances visually. However, they did not propose any specific tool to generate AIBOMs.

One potential approach to generating AIBOMs is to leverage the information provided in model cards. These were introduced by Mitchell *et al.* [112] to improve transparency in the documentation of AI models. Indeed, model cards and AIBOMs share common goals, as both aim to describe AI components. However, model cards lack a standardized structure, which limits their machine readability. Hugging Face, a widely used platform hosting over 1.5M AI models and 300k datasets, supports the use of model cards. Several studies investigated the content of Hugging Face model cards. For example, Pepe *et al.* [139] conducted an empirical study to evaluate how Hugging Face models declare datasets, possible bias, and license incompatibilities, finding that such information is poorly documented. Ait *et al.* [5] confirmed that Hugging Face offers a comprehensive set of features and a diverse range of data related to AI assets via model cards. Nonetheless, the authors also found certain limitations, particularly regarding incomplete information and limited data access.

2.4 SOFTWARE COMPOSITION ANALYSIS

The management of the software supply chain and the generation of SBOMs can be supported by SCA tools [7]. SCA aims at identifying and profiling the artifacts that compose a software system [7]. These may include, for example, third-party libraries for which the system depends, directly or indirectly, but also source code elements (*e.g.*, code snippets) or other artifacts being included in the software system. SCA can be performed on both source and binary code [197].

In addition to SBOM generation, applications of SCA include: identifying and tracing buggy or vulnerable components a software project depends upon; licensing compatibility analysis; or, in general, any analysis to check whether a software system depends on components that can create trustworthiness problems (*e.g.,* because being developed by non-trusted contributors or in hostile countries).

An example of an SCA tool is *GitHub's dependency graph* [76], available since 2017. It shows, for those repositories for which it is enabled, the lists of *dependencies* and *dependents*, respectively.[2] GitHub infers the dependency graph by analyzing manifest and lock files—*i.e.,* files, such as `pom.xml` for Maven or `requirements.txt` for pip, specifying project build dependencies. For this reason, GitHub's dependency graph is only available for projects written with programming languages for which package managers are supported (*e.g.,* the dependency graph is not available for C because none of its package managers are supported) [77]. Other than leveraging manifest/lock files, users can manually add dependencies to their project's dependency graph using the dependency submission API. The dependency graph is automatically updated when a commit changing a (supported) manifest/lock file to the default branch is pushed on GitHub or when anyone pushes a change to a dependent repository.

Since GitHub is the most popular hosting platform for open-source projects [168], practitioners and researchers have been extensively leveraging its dependency graph. For instance, practitioners have been using tools that depend on GitHub's dependency graph. One of them is Dependabot [73], a GitHub tool to monitor security vulnerabilities in the dependencies of a given repository and keep the dependencies up-to-date—in 2022, Dependabot opened more than 75 million pull requests to update repositories' dependencies [180]. Another tool based on GitHub's dependency graph is the one that allows generating SBOMs, released by GitHub in 2023. While most existing SBOM-generation tools directly leverage manifest/lock files, GitHub's tool allows for SBOM generation from a repository's dependency

2 According to GitHub's terminology, dependencies are repositories and packages the repository depends on, while dependents are repositories that depend on that repository's packages.

page without cloning the repository or using a third-party tool locally. GitHub's dependency graph has also been leveraged by researchers. For example, some studies analyzed different types of ecosystems—*e.g.,* ML-intensive projects [140] or projects leveraging pre-trained models [139]—by mining the dependencies of given libraries through GitHub's dependency graph. The accuracy of the SCA conducted by the tools mentioned above (*i.e.,* Dependabot, the SBOM generator, or any other tool leveraging GitHub's dependency graph) strongly depends on the data available to perform the analysis [197].

2.4.1 *Empirical Studies on Software Composition Analysis*

A few studies have investigated the vulnerability detection capability of SCA tools. Imtiaz *et al.* [87] conducted a case study to compare vulnerability analysis reports of nine industry-leading SCA tools on a large web app dependent on Maven and npm components. The results indicate that SCA tools vary widely in the reporting of disclosed vulnerabilities and may detect non-negligible portions of false positives. Bottner *et al.* [24] investigated the vulnerability detection capabilities of two SCA tools, namely Eclipse Steady and OWASP Dependency-Check, in the context of Java projects. The results indicate that OWASP Dependency-Check detected significantly more true positives than Eclipse Steady. Zhao *et al.* [202] proposed a model to evaluate the performance of SCA tools in detecting dependencies and vulnerabilities of Java Maven projects. According to this model, the authors assessed six SCA tools and found that none of them supported their model well.

In their qualitative study, Pashchenko *et al.* [136] investigated developers' decisions for selecting, managing, and updating software components. To this end, they performed 25 semi-structured interviews. Their results indicate that security is prioritized when selecting components if it is enforced by company policy. In case of a disclosed vulnerability within a component, developers mostly rely on community support and social channels to address that vulnerability. In fact, some developers deemed SCA tools to generate many irrelevant or low-priority alerts.

Part II

STUDIES ON SOFTWARE BILL OF MATERIALS

3

INTERVIEW STUDY ON THE ADOPTION OF SBOMS AMONG ITALIAN SOFTWARE PRACTITIONERS

This chapter is based on the paper entitled "If it's not SBOM, then what? How Italian Practitioners Manage the Software Supply Chain" [156], published in the proceedings of the 40th IEEE International Conference on Software Maintenance and Evolution.

While SBOMs are expected to shortly become the de facto standard for any organization that develops or maintains software systems, their uptake remains limited, and the reasons for this are unclear. This chapter presents the findings of semi-structured interviews with ten software practitioners from companies operating in Italy. The aim was to understand whether and how Italian software practitioners are using SBOMs, as well as the strategies employed when SBOMs are not used. In particular, the interviews covered the following aspects of the software supply chain: (i) regulations with which software practitioners must comply, (ii) the production and distribution of component inventories, (iii) the management of component updates, (iv) licensing compliance, and (v) software provenance.

CHAPTER STRUCTURE. Section 3.1 presents the study design. The results are reported in Section 3.2, along with possible practical recommendations. A discussion of the results concludes the chapter in Section 3.3.

3.1 STUDY DESIGN

This section describes the design of the interview study.

3.1.1 *Study Protocol*

The study has been conducted in the form of semi-structured interviews, which fits its exploratory nature. Semi-structured

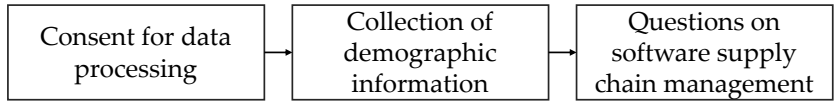


Figure 3.1: High-level view of the interview script.

interviews also allow asking for clarifications and/or details if needed or when a question is not clear to the participant [137, 163].

Ten software practitioners from six companies were recruited. Those six companies are national or the Italian division of multinational companies. Due to internal policies and the nature of the investigation, the companies preferred not to have their names disclosed. Four software practitioners were interviewed in person, while the other six were interviewed remotely (*i.e.*, through a Microsoft Teams video call) due to logistical and distance reasons. Each interview session involved the same interviewer and one participant at a time. The participants gave their consent to audio-record the session at the beginning of each interview session. Participants were also informed that the gathered data would be treated confidentially and only for research purposes. Each interview lasted between 20 and 62 minutes (on average, 33 minutes). According to the guidelines on qualitative studies [137, 163], the questions of the semi-structured interviews were not necessarily asked in the order presented in the interview script. The reason is that, depending on the evolution of the conversation, some questions could have been addressed earlier than others. Therefore, semi-structured interviews enabled improvisation and exploration of the investigated phenomenon.

The interviews followed a script to ensure that all relevant topics were covered [137, 163]. Figure 3.1 depicts the high-level view of such a script. At the beginning of each interview, the interviewer asked the participant for consent to data processing. Then, the interviewer asked demographic questions regarding the participants' education, working experience, and the types of projects and customers/clients they worked on and for, respectively. These questions are shown in Table 3.1 and aim to characterize the study context better. For instance, the development or maintenance of different software systems may entail

Table 3.1: Questions for collecting demographics information.

Questions
1. What is your educational qualification?
2. How long have you worked as a software practitioner?
3. What company do you work for, and in what division?
4. What is your role in the company?
5. How many years have you been in that role?
6. What domain do your past and present projects fit under? Who commissioned them?
7. Do you develop software for the Italian public administration? Have they placed any constraints on software components?
8. Do you develop software for EU customers? Have they placed any constraints on software components?
9. Do you develop software for US customers? Have they placed any constraints on software components?

distinct requirements [126]. The last part of the interview involved questions on how the participants' companies managed the software supply chain. These questions are shown in Figure 3.2, while the rationale behind each question is provided in Table 3.2. The questions asked to the participants were different, based on whether or not they had used SBOMs in their company. As shown in Figure 3.2, if the participant had used SBOMs, the interviewer would have asked the questions reported in the *white* boxes and denoted as Qx^s. If the participant had never used SBOMs, the interviewer would have asked only the questions reported in the *gray* boxes and denoted as Qx^o – other. Note that if the participant did not answer a question denoted by the '^{s}' symbol (*e.g.,* because it did not use SBOMs for the reported activity), the matching question marked with the '^{o}' symbol and indicated by the dashed arrow would have been asked, *i.e.,* to gather data on if there was an alternative to SBOM to address the given issue. Note also that the participants who had used SBOMs were also asked the first and the last questions in the gray boxes to understand better how their companies managed the software supply chain; in this case, the arrows connecting the white questions to the gray ones are not dashed but solid.

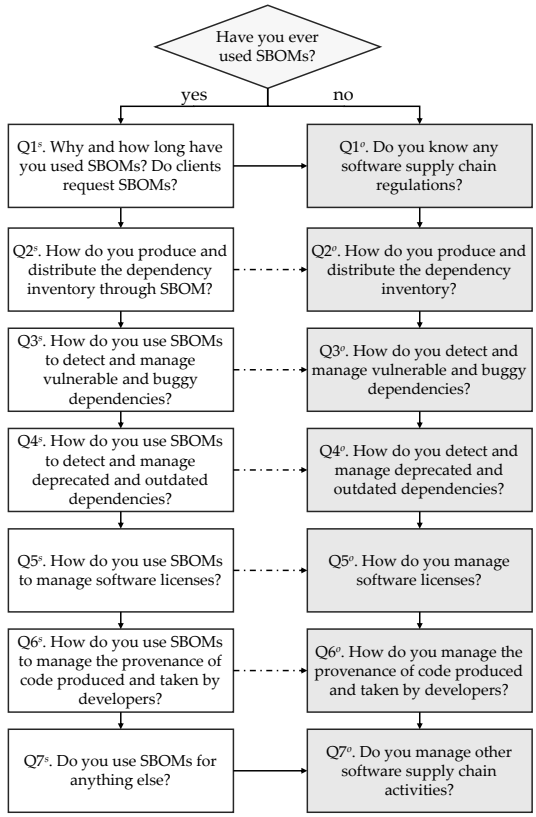


Figure 3.2: Questions for collecting information on software supply chain management.

Table 3.2: Rationale of questions on software supply chain management.

Questions
Q1. Identify software supply regulations followed by companies
Q2. Comprehend if and how component inventory is managed (*e.g.*, technologies like SBOMs, tools like Microsoft Excel)
Q3. Comprehend if and how vulnerable and buggy components are managed
Q4. Comprehend if and how deprecated and outdated components are managed
Q5. Comprehend if and how software licenses are monitored and verified
Q6. Comprehend if and how code provenance is managed (*e.g.*, code taken from Q&A forums, ownership of code written by the company's developers)
Q7. Identify other software supply chain activities (that can eventually be supported through SBOMs)

3.1.2 *Study Participants*

Table 3.3: Study participants' demographics.

ID	Experience	Company Division	Role	Job Tenure
P1[a]	17 years	Security	IT Manager	6 years
P2[a]	5 years	Security	Team Leader	5 months
P3[b]	30 years	Top Management	Chief Executive Officer	9 years
P4[c]	26 years	Software Development	Team Leader	2 years
P5[c]	22 years	Software Development	Team Leader	5 years
P6[d]	30 years	Security	Solutions Architect	5 years
P7[d]	4.5 years	Software Development	Team Leader	11 months
P8[e]	21 years	Application and Infrastructure Services	Solutions Architect	5 years
P9[f]	20 years	Top Management	IT Manager	7 months
P10[d]	8 years	IT Monitoring and Automation	Team Leader	2 years

Each letter (*a*, *b*, *c*, *d*, *e*, and *f*) indicates one of the six companies to which the participants belonged; thus, participants with the same letter worked for the same company.

Table 3.3 reports the participants' demographics resulting from the answers to the questions in Table 3.1. Each participant is referred to by an ID (from P1 to P10) to preserve their anonymity. The IDs of the participants have been established according to the order in which they were interviewed. To guarantee the anonymity of the companies where the participants worked at the time of the interview, each company is identified by a letter (from *a* to *f*).

The participants in this research were from companies within the contact network of the researchers conducting the study. These companies were asked to recruit participants with knowledge of software supply chain and how the latter was handled within their own company. Therefore, the companies shared an internal notice with the employees satisfying these criteria. The notice described the aims of the study and the types of activities proposed. At the end of this process, there were two IT managers,

five team leaders, a chief executive officer, and two solutions architects. Their working experience ranged from 4.5 years (P7) to 30 years (P6). The average working experience was about 18 years.

The participants were from six different companies, all being either Italian or an Italian branch of a multinational company. The participants of the same company had different roles and/or experience, *e.g.,* they could have worked on different projects or with different customers. In particular, P1 and P2 worked for the Italian branch of the same multinational company (*a*), which operates in the field of cybersecurity and personal data protection and has more than 500 employees (15,000 worldwide). P3 was the chief executive officer of an Italian company (*b*), which operates in the field of smart and sustainable IoT environments and has about ten employees. P4 and P5 worked for the same branch of an Italian company (*c*), which mainly develops software services and solutions and has nearly 200 employees. P6, P7, and P10 worked for the Italian branch of the same multinational company (*d*), although in different divisions (*i.e.,* security, software development, and IT monitoring/automation, respectively). This company's main business is the sale of infrastructure, software, and services in the information and communications technology field. The Italian branch of this company has nearly 2,000 employees (100,000 worldwide). P8 worked for the Italian branch of a multinational company (*e*), which mainly develops software services and solutions and has more than 5,000 employees (almost 200,000 worldwide). Lastly, P9 worked for an Italian company (*f*), which operates in the finance sector and has about 1,500 employees.

3.1.3 *Data Analysis*

After transcribing the recordings of each interview, the transcripts were analyzed using a thematic analysis approach, namely template analysis [96]. Template analysis is a flexible and fast data analysis approach that allows researchers to develop a list of codes, each identifying a theme within the transcripts. The codes are arranged in a template whose structure is usually hierarchical. Such a structure highlights the relationships among the

Table 3.4: Initial template of codes.

Theme	Focus
1) Regulations	Regulations for managing the software supply chain
2) Component Inventory	Production and distribution of the component inventory
3) Vulnerable Components	Detection and management of vulnerable components
4) Buggy Components	Detection and management of buggy components
5) Deprecated Components	Detection and management of deprecated components
6) Licenses	Software license management
7) Provenance	Source code provenance

Table 3.5: Final template of codes.

Theme	Sub-theme
1) Regulations	a) Security
	b) Release
2) Component Inventory	c) Software documentation
	d) Package managers
	e) SBOM
3) Component Update	f) Project type
	g) Bulletins
	h) CI/CD pipelines
4) Licenses	i) License Compliance
5) Provenance	j) Code produced by developers
	k) Code taken by developers

identified themes. In template analysis, the researchers analyze the transcripts using an initial template of codes. The purpose is to delimit portions of text in transcripts related to the themes of interest. Following the recommendations of King [96], an initial template was developed based on the interview script. As customary, the initial template was revised during the analysis [96]. Table 3.4 shows the initial template of codes, while Table 3.5 shows the final version. In particular, the first-level codes *Vulnerable Components*, *Buggy Components*, and *Deprecated Components* were aggregated in a single first-level code, namely *Component Update*. This is because, based on the participants' answers, vulnerable, buggy, and deprecated components are detected, managed, and

fixed similarly. In fact, a vulnerable component typically includes a (security) bug and becomes deprecated when the vulnerability-fixing version is released. Moreover, for each first-level code, two or three sub-codes were identified in the final template. The template analysis was concluded when all portions of transcripts relevant to the goal of the study were coded. To ease the analysis of the transcripts, the ATLAS.ti [2] tool was used. This tool supports qualitative data analyses, including template analysis.

3.2 RESULTS

In this section, the results emerging from the interview study are presented. These results are presented according to the main themes identified by the first-level codes of the final template shown in Table 3.5. For each main theme, a summary is provided along with recommendations for software practitioners.

3.2.1 *Regulations*

This theme includes two sub-themes related to software supply chain regulations, namely *security* and *release*.

3.2.1.1 *Security*

As for software supply chain security regulations, three participants (*i.e.,* P2, P6, P9) heard about the Cyber Resilience Act. In particular, P2 heard about it at a conference, while P6 and P9 heard about it through their company communication channels (*e.g.,* newsletters). In the same way, P6 and P9 also heard about President Biden's Executive Order 14028 on Improving the Nation's Cybersecurity. None of the participants knew any details about such regulations. Additionally, P2 mentioned two other security regulations their company follows and for which they are certified, namely DORA and TIBER-EU. They mentioned how such certifications are necessary to work for certain customers. However, according to P2, such regulations are more concerned with security testing than with the software supply chain. Therefore, they do not regulate the use of SBOMs.

P9 and P10 knew and followed some software supply chain security regulations despite not knowing the names of any of them. More in detail, P9 stated that their company must comply with precise software supply chain security regulations to operate in such kinds of domains (*i.e.*, banking, finance, and billing). Such regulations involve the handling of (i) third-party components that must not be used and (ii) services the company provides to its customers (*e.g.*, verifying software security or providing security certifications at any time). The P9's company follows such regulations to operate within the bank domain. P10 was aware of the existence of software supply chain regulations concerning SBOMs. P10 was the only participant who knew—and used, to some extent—SBOMs. In this respect, P10 mentioned that SBOMs have been a hot topic for the past year and is among the standards followed by their company. Note that P6 and P7 belonged to the same company as P10 but did not know SBOMs. One possible explanation is that the company has been gradually and experimentally introducing SBOMs through pilots in different divisions. Although this hypothesis was discussed with P10, the participant did not have a comprehensive view of the company's strategies and, therefore, could not confirm it.

3.2.1.2 *Release*

P3 pointed out that each country may have different regulations regarding the deployed software. For example, these regulations could limit the kinds of features software provides in each of these countries (*i.e.*, a functionality could be available in one country while it cannot be released in another country). According to P3, practitioners should always pay attention to their software. This clearly highlights the potential benefits of a tool designed to support the management of the software supply chain.

Summary

Despite the interviewed software practitioners knowing little about SBOM and its related regulations, they tend to follow good practices concerning the management of the software supply chain. More advanced practices and regulations are almost unknown.

Recommendation

Companies should not neglect software supply chain regulations and keep their employees informed about them. A better knowledge transfer about software supply chain practices, SBOMs, and, above all, their implications in specific domains such as service-oriented systems is highly desirable. To this end, as other companies have done, informative communications can be made through dedicated channels (such as the company's newsletters).

3.2.2 *Component Inventory*

This theme includes two sub-themes related to the production and distribution of the inventory of software components. In this respect, the following three sub-themes were identified in the participant transcripts: *software documentation*, *package managers*, and *SBOM*.

3.2.2.1 *Software Documentation*

P1, P3, and P9 relied on software documentation to produce and distribute the inventory of software components. According to P1, such documentation is based mainly on UML diagrams, *e.g.*, component or class diagrams. However, UML-based documentation does not describe fine-grained software components (*e.g.*, libraries). P3 used spreadsheets to report information on software components (*e.g.*, version, licenses). This information is entered manually. Therefore, updating it can be prone to errors and not match the components that are really present in the source code. P9 produces documentation on component inventory through Archi, a modeling toolkit for describing, analyzing, and visualizing architectures within and across business domains [11]. This toolkit is also capable of capturing dependencies between software components. As part of the software documentation, the component inventory is created and then distributed to the customers who have the software as part of the contract.

3.2.2.2 *Package Managers*

P5 and P8 relied only on package managers (*e.g.*, Maven) for the component inventory. Developers can declare the software components in the package manager files (*e.g.*, `pom.xml`). Therefore, package manager files represent the inventory of software components and are distributed to the customers who owns the software as part of the software itself. However, the customers must have enough technical knowledge to understand the information reported in package manager files.

3.2.2.3 *SBOM*

Only one of the six companies used SBOMs to generate and distribute component inventories. Interviews revealed that, among the three participants from that company (P6, P7, and P10) only P10 was aware of using SBOMs. P6 referred to SBOMs as general "reports" of component inventories. P7 stated to declare software components within package manager files. P7 also reported information on such components in an external text file that was handed over to the DevOps team of the company (of which P10 was part). As a team leader of the DevOps team, P10 was responsible for processes involving dependency management, including using SBOMs. The difference in SBOM usage among P6, P7, and P10 may be attributed to the company's large size and the fact that those participants belong to different divisions. Each division likely has a limited view of others, with new technologies being introduced at varying times and with different strategies.

P10 described the process followed by their company to produce and distribute SBOMs. Specifically, the company sets up its Continuous Integration and Continuous Delivery (CI/CD) pipelines to generate SBOMs periodically or at the occurrence of an event. As P10 reported, their company uses three SBOM generators, namely Syft [206], Trivy [10], and XRay [89] (all supporting both SPDX [154] and CycloneDX [38] SBOM standards). Such tools extract software components by analyzing package manager files. Then, they retrieve further information (on transitive components, suppliers, *etc.*) from external data sources. The choice to use multiple SBOM generators is because different tools can generate different SBOMs, *i.e.*, different lists of components.

Therefore, the use of multiple SBOM generators increases the chance of identifying all components. P10 also mentioned that SBOM generators can have recall and precision concerns. In this respect, P10 also told that, in the past, the company used another SBOM generator, namely Anchore [204], which was abandoned due to the production of too many false positives. Since more than one SBOM generator is used, the company has to merge the information from multiple SBOMs into a single one. Moreover, to verify (and eventually integrate) the SBOM generators, single developers (such as P7) have to manually annotate each software component they introduce and its version. As previously mentioned, the developers write this information in an external textual file, which is handed over to the DevOps team. Lastly, once the SBOM is completed, the component inventory is declared within an in-house monitoring system. When an issue emerges in a component, all projects that have declared such a component in the in-house monitoring system are notified. P10 claimed that SBOMs are particularly useful for distributing the component inventory because they have different formats (*e.g.*, JSON) that can be manipulated through various tools. Thus, even stakeholders without specific technical skills can interact with the software component inventory.

Summary

SBOMs are effective in managing software component inventory at finer-grained levels.

Recommendation

The software industry can take advantage of SBOM to document component information at different levels of granularity, including at the library level, which cannot be effectively represented by common software documentation, *e.g.*, UML diagrams. The use of multiple SBOM generators, as well as manual component annotation, can enhance the reliability of the provided component information. However, further work needs to be done to provide reliable, automated SBOM generators.

3.2.3 *Component Updates*

This theme includes three sub-themes related to detecting and managing software components, namely *project type*, *security bulletins*, and *CI/CD pipelines*.

3.2.3.1 *Project Type*

The strategy for managing the component update changes according to the project type. As stated by P5, customers who own legacy software may prefer not to update any components as long as their software is working properly. P10 pointed out that software implemented with older technologies could be too difficult, if not impossible, to update. When a component update is not possible, isolating the software (*e.g.,* through containers) is a way to limit the risk of security issues.

When the software requires evolutionary and/or corrective maintenance, software practitioners may adopt a more proactive approach to component updates by keeping components to third-party libraries always up-to-date at the latest available version and resolving incompatibilities incrementally as soon as they occur. This is done to reduce the risk that the presence of several too-old, incompatible components causes a hard-to-resolve building breakage.

3.2.3.2 *Bulletins*

P1 and P2 rely on bulletins to detect and manage components, especially vulnerable ones. The information on vulnerable components is reported in security bulletins. These bulletins are published daily/weekly by authoritative entities such as NIST, which also maintains the NVD and is responsible for publishing CVEs. Security bulletins may contain the remediation for the reported vulnerable components (*e.g.,* changing the component version or applying a fix in the source code).

A Security Operation Center (SOC) may supervise the validation of software components against the vulnerabilities disclosed in security bulletins. Each customer who owns the software establishes with the SOC the level of service desired (*e.g.,* 24 hours a day, five days a week). This may also be done on single

components—*i.e.,* it is possible to retrieve and monitor all the security bulletins of a specific component.

3.2.3.3 *CI/CD Pipelines*

P3, P8, and P10 pointed out CI/CD as an important part of the software supply chain. Software development may involve the production of several modules that, after being implemented, must be integrated. Each module can evolve independently but must always be able to integrate with the others, which could be quite complex. CI/CD pipelines are employed to verify the integration of the different components that constitute software. CI/CD pipelines can also verify the delivery/deployment of software in specific environments and according to specific technological constraints. To this end, CI/CD pipelines perform tasks periodically and/or at the occurrence of certain events (*e.g.,* pull requests, project milestones, or customers' requests), including security tasks. To perform all these tasks, CI/CD pipelines leverage different program analysis tools, such as those based on static analysis, SCA, and SBOM. It is possible to define certain criteria (*e.g.,* not using a particular version of a component) so that the CI/CD pipeline fails if such criteria are not met.

SCA tools analyze the software to retrieve the list of the vulnerable and deprecated components, along with their remediation, if any. Among the SCA tools, P1 declared to mainly use Fortify [132], a commercial product. In the past, they also used Checkmarx [27]. P8 recommended the use of Trivy as a SCA tool. As previously mentioned, Trivy also supports the generation of SBOMs, although P8 stated not to know and use SBOMs.

CI/CD pipelines may also involve SBOM generation and consumption. Similar to SCA tools, SBOM consumption tools allow for checking vulnerable and deprecated components reported in SBOMs. The company of P10 had the DevOps team analyze the SBOMs generated through Trivy and Grype [205]. Being part of the DevOps team, P10 defined the user stories to resolve the detected vulnerable and deprecated components. P6 also pointed out the use of Nessus [177] as a vulnerability scanner. Moreover, according to P6, the company's in-house monitoring system checks daily for any newly published CVEs that refers to com-

ponents declared by any of its projects and eventually notifies them. More generally, such an in-house system checks for all components that need to be updated, including deprecated ones.

Summary

Software components may need to be updated due to issues affecting them, such as vulnerabilities. These components can be detected by CI/CD processes and divulged by bulletins published by authoritative entities. Nevertheless, updating them may differ depending on the type of project and the technologies it uses.

Recommendation

Companies should integrate SCA or SBOM tools in their CI/CD pipelines to identify components that need to be updated. Tool developers should enhance SCA/SBOM capabilities to better support legacy software systems.

3.2.4 *Licenses*

This theme includes only one sub-theme related to the management of software licenses, namely *license compliance*.

3.2.4.1 *License Compliance*

According to P7, some customers prefer employing (free) open-source components to avoid incurring further costs. In addition, since the source code of open-source components is available, customers could request modifications to it according to their specific needs. This indicates a severe problem: customers may not know the difference between open-source and free software. This is crucial because, according to P3, customers are primarily responsible for license compliance.

A possible solution for managing license compliance is to (manually) track information about each license on a spreadsheet. There are also solutions that (partially) automate licensing management. For instance, P6's company mainly relies on XRay and Trivy that, however, simply identify components along with their

licenses, leaving the incompatibility identification to software practitioners. The reason for using multiple tools is to gather as much information (on licenses) as possible. However, different tools' outputs may conflict. P10 recalled that, in the past, it happened that XRay and Trivy produced different licenses for the same component. Therefore, they manually checked the incorrect licensing-related information and observed that XRay produced the incorrect output. They then had to open a ticket to the vendor of this tool reporting the detected problem. Lastly, through its in-house monitoring system, the same company checks for any software components that could violate its company policies. The licensing incompatibility identification was a challenge for the company in which P6, P7, and P10 worked (they all worked in the same company), especially when third-party components have several different types of licenses. Although the company used SBOM-based technologies (which help with license management), such verification was still pretty manual and did not appear overly structured.

Summary

The management of software licenses is still pretty manual.

Recommendation

The software industry should adopt SBOMs to improve the awareness of the software licenses they adopt and automate their management.

3.2.5 *Provenance*

This theme includes two sub-themes related to the management of component provenance, namely *code produced by developers* and *code taken by developers*.

3.2.5.1 *Code Produced by Developers*

Most participants (P1, P4, P5, P6, P7, P8, P9, and P10) were aware of laws (including penal laws) and company policies regulating code reuse by developers. These laws and policies mainly focused

on code written by developers. According to them, code written by developers cannot generally be reused outside of the software it was written for. Some of these laws may also prohibit disclosing information about the software being developed. This concern is particularly relevant when developers write code for customers of their company, as they are the ones owning the software.

P1, P3, P4, P7, P10 felt it was still too difficult—if not impossible—to identify developers who reuse code illegally. They believed that developers' responsibility and the deterrent provided by laws/policies are practically the only means of reducing the risk of illegal source code reuse. P7 believed there might be source code whose reuse is not a concern. P7 shared the experience of implementing a graphics component that was subsequently turned into a reusable library, emphasizing that it did not contain sensitive code. P7 believed that source code reuse is a problem, especially for the back-end, since that code might involve more sensitive code than that of the front-end.

The only participant who was aware of tools to identify illegal code reuse was P9. This participant stated that their company has software that scans public Git repositories to verify that company code is not being outsourced and made public. Previously, the company discovered a developer who had illegally reused its source code. This was why this company equipped itself with such a tool.

3.2.5.2 *Code Taken by Developers*

Most participants (*i.e.,* P3, P4, P6, P7, P8, P9, P10) explicitly stated that their companies do not have guidelines for code taken from external sources, such as generative AI (*e.g.,* ChatGPT, GitHub Copilot) or Stack Overflow. As for generative AI, P3 and P5 heard about the EU AI Act [58] by reading online blogs; however, they knew nothing more detailed about regulations regarding the use of AI in software development. P7 believed that the ethics of individual developers play a key role in the use of generative AI. As soon as ChatGPT was released, the company emailed employees, warning them of the possible dangers of generative AI (*e.g.,* sharing sensitive data). P7 mostly used ChatGPT to generate code from scratch rather than modify existing code;

in the latter case, P7 avoided providing sensitive information or complete code snippets. The companies of P1 and P8 were working to regulate the use of generative AI. P1 explicitly stated that, to date, their company forbids the use of generative AI because it has not yet been regulated, and their customers do not require the use of such technology, but neither do the customers forbid its usage. The results related to this sub-theme open an interesting point related to the software supply chain and the generated source code.

It must be noted that, currently, SBOMs are primarily designed to describe components at the level of software libraries and packages. However, this would not preclude the possibility of tracing artifacts at varying levels of granularity (*e.g.*, files reused from other projects) and, when the technology allows it, also trace the chain of AI-generated artifacts.

Software companies can take various measures to deter developers from reusing code illegally. Among them, participants reported awareness-raising actions (*e.g.*, training courses), signing non-disclosure agreements, and using secured equipment provided by the company. In the latter case, participants referred to computers that contain the software to work on and that are restricted in the actions developers can perform. For example, these mentioned computers do not allow for copying and pasting of code. They are also configured to prevent developers from accessing online resources that are not verified by the company.

P3, P9, and P10 believed that using code from generative AI is not particularly dangerous, as their CI/CD pipelines would inspect the code and detect any concerns.

Summary

According to the participants' answers, 5 out of 6 companies prohibit developers from redistributing their code (*e.g.*, on hosting platforms like GitHub), while only one company forbids the use of generative AI in software development. Despite the adoption of such policies, it is still difficult to track and manage software provenance.

Recommendation

SBOMs should be used to support software provenance tracking and monitoring. To this end, the vendors of SBOM tools should integrate features for detecting the provenance of reused code, as well as identifying AI-generated code.

3.3 DISCUSSION

The findings indicate that the adoption of SBOMs is low, yet the interest of the software industry in the software supply chain is high. A possible reason behind this outcome is that the software industry has limited knowledge of SBOMs and software supply chain regulations. Although some participants showed a growing interest in SBOMs, the Italian software industry seems to respond less promptly to this technology.

In comparison with previous studies [170, 171, 195], mainly conducted in a US-based context or in a wider context, both commonalities and differences were found:

- The awareness about the importance of generating SBOM is quite low in Italy, if compared to what happens, for example, in the US [170, 171, 195]. This might also be due to the lack of strict regulations—similar to President Biden's Executive Order 14028 on Improving the Nation's Cybersecurity [186]—that forces software suppliers to generate and distribute SBOMs in certain circumstances. That being said, related EU Acts such as the Cyber Resilience Act [56], but also the EU AI Act [58], are posing the basis for better transparency and management of the software supply chain.

- One challenge clearly emerging from this study that confirms the previous findings [170, 171, 195] is the limited availability of tools for SBOM generation and consumption and, in general, for supply chain management. In the Italian software industry, several activities related to dependency management and licensing compliance analysis are mostly performed manually or by combining multiple tools, often providing contradictory results. Therefore,

> the road toward reliable tools and their wider adoption in the industry—especially in the studied context—is still long. These findings suggest that an intense technology transfer from academia to industry on supply chain and SBOM-related topics is highly desirable.

As SBOM adoption becomes increasingly critical—driven by emerging regulations such as the EU Cyber Resilience Act—software suppliers will need to stay ahead of new software supply chain requirements. Suppliers in regions with lower adoption rates (such as Italy) may face challenges, as they are not yet prepared for the potential pressure to provide SBOMs.

Software choosers, including large enterprises and government agencies, are likely to demand SBOMs more consistently to assess the security, compliance, and traceability of software components. As regulatory pressure grows, procuring organizations may make SBOMs a requirement, especially in highly regulated sectors. However, due to suppliers' ongoing reliance on manual or fragmented dependency management, choosers may encounter difficulties verifying that software meets expected requirements, leading to inefficient procurement and increased risk.

Software operators may struggle to manage dependencies and maintain compliance, given their limited supply chain tooling and reliance on manual processes. As regulatory requirements tighten, operators will need to adopt tools that automate and streamline the process of SBOM consumption, especially when managing complex software supply chains.

3.3.1 *Threats to Validity*

This section discusses potential threats to the validity of the results. According to the validity schema by Wohlin *et al.* [194], these threats can be categorized as construct, reliability, external, and internal validity. Threats to internal validity are not discussed because such threats concern causal relationships [194], and causality was not investigated in this interview study.

Threats to *construct validity* concern the relation between theory and observation [194]. The sampling method used in the interview study might have affected the results. Participants in

the interview study might not have answered truthfully because they were scarcely motivated or afraid of being judged. To mitigate this threat, participation in the interview was voluntary because volunteers are generally more motivated [194]. This was why the companies involved in the study were asked to recruit participants through an internal notice for the employees with knowledge on the software supply chain (see Section 3.1.2). The participants were also informed (before the interview took place) that the gathered data would be treated confidentially. Participants could not have understood the themes or points. To mitigate this threat, the themes were approached top-down, starting with the general ones and moving to the specific ones. The study did not concern sensitive information regarding companies and their employees. Therefore, confidentiality should not impact the validity of the results from the semi-structured interviews. In any case, it was done the best to control this kind of threat while participants discussed the themes of interest of the study with the interviewer.

Threats to *reliability validity* concern the extent to which the data collection and analysis depend on the individual researchers involved [194]. Researchers might unconsciously influence the results based on their expectations. Such a threat was mitigated by involving more than one researcher in the analyses of the gathered data (*i.e.,* investigator triangulation [179] was applied).

Threats to *external validity* concern the generalizability of the findings [194]. The number of interviews might threaten the validity of the results. However, Guest *et al.* [82] observed that a sample of six interviews may be sufficient to derive meaningful themes and interpretations. Based on the analysis of the ten interviews, a point of diminishing return [82] appears to have been reached, where conducting additional interviews was unlikely to yield more evidence. Finally, the results may not hold outside the context where the study was conducted. That is, since the results are based on evidence collected from software practitioners belonging to six different software companies operating in Italy, it is not possible to claim that such results apply to all the software companies operating in Italy and around the world.

4

MINING SOFTWARE REPOSITORIES STUDY ON THE ADOPTION OF SBOMS IN OPEN-SOURCE SOFTWARE SYSTEMS HOSTED ON GITHUB

This chapter is based on the paper entitled "On the adoption of software bill of materials in open-source software projects" [158], published in the Journal of Systems and Software, which extends the work presented in "Software Bill of Materials Adoption: A Mining Study from GitHub" [156], published in the proceedings of the 39th IEEE International Conference on Software Maintenance and Evolution.

Previous research on SBOM has primarily focused on qualitative insights from practitioners, particularly within the software industry. However, there is a notable gap in understanding how SBOMs are used in practice, especially in the open-source context. To address this gap, a mining software repository study was conducted to quantitatively investigate the adoption and usage of SBOMs in open-source systems hosted on GitHub. Specifically, the study examined the adoption of SBOM generators, as well as the distribution and updating of SBOMs. Additionally, the compliance of distributed SBOMs was assessed with established standards (*i.e.*, SPDX and CycloneDX) and the NTIA's minimum and recommended data fields.

CHAPTER STRUCTURE. Section 4.1 presents the study design. The results are reported in Section 4.2, while their discussion concludes the chapter in Section 4.3.

4.1 STUDY DESIGN

The *goal* of this exploratory mining study is to analyze the adoption of SBOMs on open-source systems with the *purpose* of understanding (i) to what extent SBOMs are adopted; (ii) what is their adoption trend; (iii) how SBOMs are distributed; (iv) with which frequency SBOMs made available in software repositories get updated; (v) to what extent they comply with the standards; and

(vi) what pieces of information SBOMs report, including NTIA's minimum and recommended data fields [185]. The *perspective* is that of software suppliers, choosers, operators, and researchers. The *context* is represented by open-source software systems available on GitHub and depending on SBOM generators owned by SPDX and CycloneDX.

4.1.1 *Research Questions*

Based on the above-mentioned study goal, the following six Research Questions (RQs) were formulated and investigated.

RQ1. *Are SBOMs Adopted by Open-source Software Systems?*

This RQ aims to understand whether open-source software systems adopt SBOMs by looking at systems that use SBOM generators based on the SPDX and CycloneDX standards.

RQ2. *What is the Adoption Trend of SBOMs by Open-source Software Systems and their Owners?*

This RQ focuses on the adoption trend of SBOMs, by looking at certain events of interest (*e.g.*, President Biden's Executive Order 14028 on Improving the Nation's Cybersecurity [186]) that might increase the adoption of SBOMs by both open-source software systems and their owners.

RQ3. *How do Open-source Software Systems Distribute SBOMs?*

This RQ investigates how SBOMs are managed by open-source software systems and, more specifically, how SBOMs are distributed, namely where SBOMs are placed (*e.g.*, under version control or just in public release versions).

RQ4. *How Frequently SBOMs Get Updated within Open-source Software Systems?*

This RQ contributes to understanding how SBOMs are managed in open-source software systems and, more specifically, with which frequency developers update the SBOMs they make available for these systems.

RQ5. *To What Extent do SBOMs Comply with the Standards?*

This RQ aims to verify whether SBOMs comply with the versions of the SPDX and CycloneDX standards they declare to be compliant with, since uncompliant SBOMs might lead to failures during their ingestion by SBOM consumption tools.

RQ6. *What are the Pieces of Information Reported by the Found SBOMs?*

This RQ aims to understand whether SBOMs comply with NTIA's minimum and recommended data fields [185] and what other information they report (*e.g.,* whether or not vulnerability information is included).

4.1.2 *Study Context and Planning*

The study context consists of open-source software systems available on GitHub and that depend on SBOM generators owned by SPDX or CycloneDX according to GitHub's dependency graph [76]. As highlighted by Kalliamvakou *et al.* [93], the analysis of data retrieved from GitHub can lead to conclusions that are not representative of the context that is intended to be investigated. To deal with such an issue, we excluded those repositories that:

- *Were archived*, and *whose last push date was more than one month before the query date (March 5, 2023)*. The rationale was to avoid selecting inactive software systems [93].
- *Were forks*. This was to avoid selecting duplicate software systems [93].
- *Had a number of commits less than 100*. The rationale here was to avoid selecting software systems with a commit history too short and to mitigate the risk of selecting personal/toy systems [93].
- *Were not related to software development*. This was to discard tutorials, books, students' assignments, and researchers' papers [93]. To do so, it was searched within the name of the repositories and their owner for any instance of the terms `paper`, `book`, `tutorial`, `learn`, `bootcamp`, `sample`,

`example`, `demo`, `test`, `mirror`, `copy`, `community`, `docs`, `theme`, and `template`. Then, it was manually checked the suitability of the matched repositories and decided whether or not to discard them.

General information regarding the repositories (*e.g.,* number of commits and stars) and their owners was collected through PyGithub [144], a Python library that leverages GitHub API v3. The commit-level information (*e.g.,* file modification dates and messages) was retrieved through PyDriller [167].

4.1.3 *Data Analysis*

This section describes the data analysis per RQ.

4.1.3.1 *RQ1. Are SBOMs Adopted by Open-source Software Systems?*

To identify the software systems that adopted SBOMs, we first selected SBOM generation tools among those whose software repository was available on GitHub and whose owner was SPDX or CycloneDX. Then, for each of the selected SBOM generation tools, we retrieved the list of repositories that depended on them by leveraging GitHub Dependency Graph. This list of repositories represents the open-source software systems that adopted SBOMs.

According to the classification of SBOM generators provided by SPDX [151] and CycloneDX [34], SBOM generators can be classified as:

- *Build-integration*, indicating tools to be integrated within build systems (*e.g.,* Maven) so that SBOMs can be automatically created when building software systems. Note that a build-integration tool could even be executed as a command-line tool within a CI/CD workflow.

- *GitHub Action*, when the tool is specifically conceived to work in a GitHub Actions CI/CD workflow. Such workflows are typically triggered when certain events (*e.g.,* a push on the main branch) occur.

Table 4.1: List of CycloneDX and SPDX constituent parts.

Standard	Constituent part
CycloneDX	`"bomFormat":"CycloneDX"`
	`xmlns="http://cyclonedx.org`
	`"$schema":"http://cyclonedx.org`
	`'bomFormat':'CycloneDX'`
	`xmlns='http://cyclonedx.org`
	`'$schema':'http://cyclonedx.org`
	`<vendor>CycloneDX</vendor>`
	`"$schema":"https://cyclonedx.org`
	`'$schema':'https://cyclonedx.org`
	`xmlns="https://cyclonedx.org`
	`xmlns='https://cyclonedx.org`
SPDX	`SPDXVersion:SPDX-`
	`SPDXID:SPDXRef-DOCUMENT`
	`"SPDXID":"SPDXRef-DOCUMENT"`
	`"spdxVersion":"SPDX-`
	`"SPDXID":"SPDXRef-`
	`'SPDXID':'SPDXRef-DOCUMENT'`
	`'spdxVersion':'SPDX-`
	`'SPDXID':'SPDXRef-`

- *Library*, indicating modular components that allow developers to generate, consume, or transform SBOMs using a programming approach.

We assumed that software systems using SBOM generators integrated into build systems or GitHub Actions workflows generate SBOMs. As for SBOM generators classified as libraries, to ensure that they were used to generate SBOMs, we looked for the presence of SBOMs under version control or in the latest public release versions and excluded those systems that did not have any. To this end, we locally cloned each repository and downloaded the latest public release version. Then, we searched within each file (including archive files such as TAR or ZIP) for the presence of the constituent parts of the SPDX and CycloneDX standards to determine whether or not that file was an SBOM. The list of the CycloneDX and SPDX constituent parts used to perform the search is shown in Table 4.1.

For each system, we deduced the adopted SBOM standards (SPDX or CycloneDX) based on the SBOM generators. Then, we counted how many times each standard was adopted by each software system, making a distinction based on whether the system was related to the domain of SBOMs or not. This was accomplished by looking for the following terms in the repository's name, description, and README file: `SBOM`, `CycloneDX`, `SPDX`, `Bill-of-Materials`, and `Supply-Chain`. If we found the occurrence of one of these terms for a system, that system was considered to be related to the domain of SBOMs. This distinction helped determine how many of the systems using SBOMs were related to the SBOM domain, not merely utilizing SBOM generators, but actively contributing to the SPDX or CycloneDX ecosystems.

4.1.3.2 *RQ2. What is the Adoption Trend of SBOMs by Open-source Software Systems and their Owners?*

For each repository, we collected information about the commit introducing an SBOM generator dependency by performing a `git log` operation on the manifest and lock files (*e.g.*, `pom.xml` for Maven or `.yaml` for GitHub Actions). We linearly examined the commit history of these files, from the earliest to the oldest version, until the commit introducing the SBOM generator dependency was identified, along with its date.

We counted the number of adoptions over time of SBOM generators with respect to the software systems, but also to owners, to control the influence that a few prolific owners might have compared to the whole sample size. Also, we took into account historically significant events related to SBOMs (*e.g.*, publication of standards or guidelines) to see whether they had an impact on the spread of SBOM generators.

Finally, we assessed whether the introduction of a dependency towards an SBOM generator was explicitly mentioned in the message of the commit that introduced that dependency. To do so, we first searched into each commit message (taking into account only the commits introducing SBOM generator dependencies) for the following terms: `SBOM`, `CycloneDX`, `SPDX`, `Bill-of-Materials`, and `Supply-Chain`. Then, each matched message was deemed

an explicit mention of SBOM adoption, while non-matching messages were manually examined.

4.1.3.3 *RQ3. How do Open-source Software Systems Distribute SBOMs?*

For each dependency towards an SBOM generator, we assessed the presence of SBOMs under version control or in the latest public release version by searching for the constituents of SPDX and CycloneDX (similar to what was done for RQ1 when analyzing SBOM generators classified as a library). Also, for each system, we investigated whether the SBOMs appeared (i) under version control, (ii) in the latest public release version, (iii) on both version control and last public release, or (iv) nowhere.

Finally, as for the software systems using GitHub Actions to generate SBOMs, we checked what kind of event (*e.g.,* push, pull request) was specified in the CI/CD workflow to trigger the SBOM generation.

4.1.3.4 *RQ4. How Frequently SBOMs Get Updated within Open-source Software Systems?*

For each repository, we counted the number of commits that modified its SBOMs, starting from the commit in which the SBOMs were first created up to the most recent commit at the time of the analysis (April 1, 2023). This count was then converted to a monthly rate to estimate the frequency of SBOM changes.

For each repository that published SBOMs in the latest public release (using GitHub's release feature), we computed the percentage of release versions containing SBOMs, either as part of the release archive or as a separate file.

Finally, for each commit in which an SBOM was modified, we checked whether it was associated with a `git tag` and whether that `git tag` was directly associated with a release of the repository—this was to understand whether the generation of a new SBOM occurred in correspondence with a new release. If there was no `git tag` associated with the commit that modified the SBOM, then the SBOM generation did not occur in correspondence with a new release.

4.1.3.5 *RQ5. To What Extent do SBOMs Comply with the Standards?*

A software system can distribute multiple SBOMs. This can happen when: (i) there are duplicated SBOMs (*i.e.,* same SBOM released in different file formats); (ii) SBOMs are targeted to different hardware/software configurations (*e.g.,* an SBOM is specific for the Windows operating system while another one is specific for Linux); or (iii) SBOMs are both under version control and available in public release versions. We discarded duplicated SBOMs from the analysis to avoid a misrepresentation of the phenomenon under investigation, while those that were targeted to different hardware/software configurations were kept since they contained different component information. As for SBOMs that were both under version control and available in public release versions, we kept the most up-to-date to reflect the current practice as much as possible. Then, we verified the validity of the distributed SBOMs for the SPDX and CycloneDX standards. We employed sbom-utility (v0.13.0) [36], a tool that verifies whether a given SPDX or CycloneDX SBOM conforms to the specified version of the respective standard. This tool works with SBOMs in the JSON format; therefore, SBOMs that were not compliant with this format were translated. We used cyclonedx-cli (v0.25.0) [35] to translate any SPDX and CycloneDX SBOMs from XML to JSON format. As for SBOMs released as plain text files (which were only in the SPDX standard), we translated them into JSON using SPDX Online Tool [152].

The sbom-utility tool validates SBOMs against the JSON schema of the corresponding standard version, through the *validate* command. Table 4.2 reports the CycloneDX and SPDX schema supported by this utility tool. To comply with a scheme, an SBOM must indicate the standard and the version thereof, along with all the required attributes for that standard version. It is worth mentioning that the required attributes for a given standard version do not necessarily correspond to the data fields deemed mandatory by other organizations (*e.g.,* NTIA's minimum data fields [185]). To specify the standard and version in an SBOM, CycloneDX provides the attributes `bomFormat` and `specVersion`, while SPDX provides the attributes `SPDXID` and `spdxVersion`. As for the other required attributes, they can vary

Table 4.2: List of CycloneDX and SPDX schemas supported by sbom-utility [36].

Standard	Version	Schema URL
CycloneDX	1.5	https://raw.githubusercontent.com/CycloneDX/specification/master/schema/bom-1.5.schema.json
	1.4	https://raw.githubusercontent.com/CycloneDX/specification/master/schema/bom-1.4.schema.json
	1.3	https://raw.githubusercontent.com/CycloneDX/specification/master/schema/bom-1.3.schema.json
	1.2	https://raw.githubusercontent.com/CycloneDX/specification/master/schema/bom-1.2.schema.json
SPDX	2.3	https://raw.githubusercontent.com/spdx/spdx-spec/development/v2.3/schemas/spdx-schema.json
	2.3.1	https://raw.githubusercontent.com/spdx/spdx-spec/development/v2.3.1/schemas/spdx-schema.json
	2.2.2	https://raw.githubusercontent.com/spdx/spdx-spec/v2.2.2/schemas/spdx-schema.json
	2.2.1	https://raw.githubusercontent.com/spdx/spdx-spec/v2.2.1/schemas/spdx-schema.json

among different versions of the same standard. For example, a CycloneDX SBOM with version 1.5 does not require any other attribute to be valid, while a CycloneDX SBOM with version 1.4 also requires the `version` attribute (which refers to the version of the SBOM, not to the standard version).

When an SBOM is not compliant with the schema of its standard version (*e.g.,* absence of required attributes or incorrect values for some attributes), the sbom-utility signals that the SBOM is not valid and reports the validity errors encountered.

Summing up, the output of the sbom-utility tool allowed us to check how many SBOMs were valid with respect to their standard versions and quantify the kind of validity errors encountered.

4.1.3.6 *RQ6. What are the Pieces of Information Reported by the Found SBOMs?*

To identify which information is reported in CycloneDX and SPDX SBOMs, we parsed the JSON files of the SBOMs retrieved in the software systems—these are the same SBOMs studied in RQ5—and then the (JSON) attributes along with their values

were extracted. To map NTIA's minimum data fields [185] with the extracted attributes, we leveraged NTIA's mapping with respect to the SPDX and CycloneDX standards [185]. Then, we counted how many SBOMs, systems, and owners contained each data field. As for NTIA's recommended data fields [185], each of these data fields was mapped with the remaining attributes and the number of times each data field (and its value) appeared was counted, along with the number of SBOMs, systems, and owners involved.

The study of the *other unique identifiers* recommended data field was deepened by looking at which component/SBOM identifiers (*e.g.*, purl, CPE, *etc.*) were used and to what extent (*i.e.*, how many SBOMs, systems, and owners were involved). Similarly, the study of the *component hash* recommended data field was deepened by looking at which hashing algorithms were used and to what extent; we also verified whether the same hash value (for a given hashing algorithm) was associated with different components.

We built dependency graphs to better analyze the dependency information reported in SBOMs. To this end, for each SBOM, we first identified the root node (*i.e.*, the node corresponding to the software system for which the SBOM was created) and then the direct and transitive dependencies from the root node. The dependency graphs were generated using NetworkX [123], a Python package for creating, manipulating, and studying complex networks. We finally counted the number of SBOMs, systems, and owners where direct and transitive dependencies appeared.

4.2 RESULTS

This section presents the results organized by RQ.

4.2.1 *RQ1. Are SBOMs Adopted by Open-source Software Systems?*

Table 4.3 and Table 4.4 report, respectively, the SPDX and CycloneDX SBOM generators, their classification (*i.e.*, Build-integration, GitHub Action, or Library), the supported language, and the number of dependent software repositories that were retrieved through GitHub's dependency graph. The majority of the dependent software repositories were associated with build-

Table 4.3: CycloneDX SBOM generators considered in this study, with the number of dependent repositories.

Tool Classification	Tool Name	Language	#
Build-integration	cyclonedx-cocoapods	Swift, Objective-C	0
	cyclonedx-conan	C, C++	0
	cyclonedx-dotnet	.NET	0
	cyclonedx-gomod	Go	2
	cyclonedx-gradle-plugin	Java	0
	cyclonedx-maven-plugin	Java	74
	cyclonedx-node-module	JavaScript	7
	cyclonedx-node-npm	JavaScript	3
	cyclonedx-php-composer	PHP	5
	cyclonedx-python	Python	64
	cyclonedx-ruby-gem	Ruby	0
	cyclonedx-rust-cargo	Rust	0
	cyclonedx-webpack-plugin	JavaScript	3
	Total		*158*
GitHub Action	cdxgen-action	SourceImages, ContainerImages	0
	gh-cocoapods-generate-sbom	Swift, ObjectiveC	0
	gh-dotnet-generate-sbom	.NET	0
	gh-gomod-generate-sbom	Go	13
	gh-node-module-generatebom	JavaScript	2
	gh-php-composer-generate-sbom	PHP	1
	gh-python-generate-sbom	Python	1
	Total		*17*
Library	cyclonedx-core-java	Java	2
	cyclonedx-dotnet-library	.NET	0
	cyclonedx-go	Go	4
	cyclonedx-javascript-library	JavaScript	0
	cyclonedx-node-pnpm	JavaScript	0
	cyclonedx-node-yarn	JavaScript	0
	cyclonedx-php-library	PHP	0
	cyclonedx-python-lib	Python	1
	Total		*7*

integration SBOM generators (*i.e.*, 166 occurrences in total, 158 of which for CycloneDX and eight for SPDX), followed by those classified as GitHub Action (*i.e.*, 17 occurrences, all for CycloneDX). It is worth mentioning that 11 repositories depended on more

Table 4.4: SPDX SBOM generators considered in this study, with the number of dependent repositories.

Tool Classification	Tool Name	Language	#
Build-integration	spdx-maven-plugin	Java	8
	Total		*8*
Library	Spdx-Java-Library	Java	1
	spdx-tools-js	JavaScript	0
	tools	Java	1
	tools-golang	Go	4
	tools-java	Java	0
	tools-python	Python	1
	Total		*7*

Table 4.5: Some descriptive statistics about the systems dependent on SBOM generators.

Metric	Mean	SD	Min	Median	Max
# Commits	2,463.72	7,367.49	104	579.5	63,328
# Stars	937.14	3,674.42	0	3	35,126
# Subscribers	45.41	186.28	0	3	2,056
# Contributors	35.85	81	1	4.5	364
# Forks	431.7	2,273.05	0	1	26,661

than one SBOM generator; therefore, the total number of distinct repositories that relied on an SBOM generator was 186.

Table 4.5 reports some descriptive statistics—*i.e.*, mean, standard deviation (SD), minimum (min), median, and maximum (max)—about the dependent repositories that were studied. The values of these descriptive statistics suggest that the sample is quite heterogeneous in terms of the number of commits, stars, subscribers, contributors, and forks. Furthermore, the number of commits and contributors is greater than 579.5, with a median value of 4.5. This indicates a noticeable collaborative development activity in these software systems.

Table 4.6 shows information on the adoption of the SBOM standards, taking into account whether or not they were related to the SBOM domain. CycloneDX is the most widely used standard, accounting for 94% of the cases (*i.e.*, $(173 + 2)/186 * 100$). Two

Table 4.6: Adoption of SPDX and CycloneDX among the studied systems.

Domain	Standard	# Systems
SBOM-related	Only SPDX	8
	Only CycloneDX	22
	Both SPDX and CycloneDX	1
SBOM-unrelated	Only SPDX	3
	Only CycloneDX	151
	Both SPDX and CycloneDX	1
Overall	Only SPDX	11
	Only CycloneDX	173
	Both SPDX and CycloneDX	2

software systems adopted both the CycloneDX and SPDX standards. Also, 31 (17%) software systems using SBOM generators are related to the SBOM domain.

Answer to RQ1

SBOMs are not widely adopted in open-source systems, since only 186 systems adopting SBOM generators have been identified across all repositories hosted on GitHub. The most used standard is CycloneDX, and the preferred way to generate SBOMs is through build-integration tools.

4.2.2 *RQ2. What is the Adoption Trend of SBOMs by Open-source Software Systems and their Owners?*

Figure 4.1 depicts when, over the evolution histories, repositories and owners started adopting SBOM generators. In particular, the blue and orange lines highlight, year by year, how many new owners and repositories started adopting SBOM generators, respectively. The gray bars indicate major events involving SBOMs and software supply chain practices (*e.g.,* the conception and release of the SPDX and CycloneDX standards [37, 153], President Biden's Executive Order 14028 on Improving the Nation's Cybersecurity [186]). Based on the gathered data, SBOM generators have started to be adopted by open-source software systems

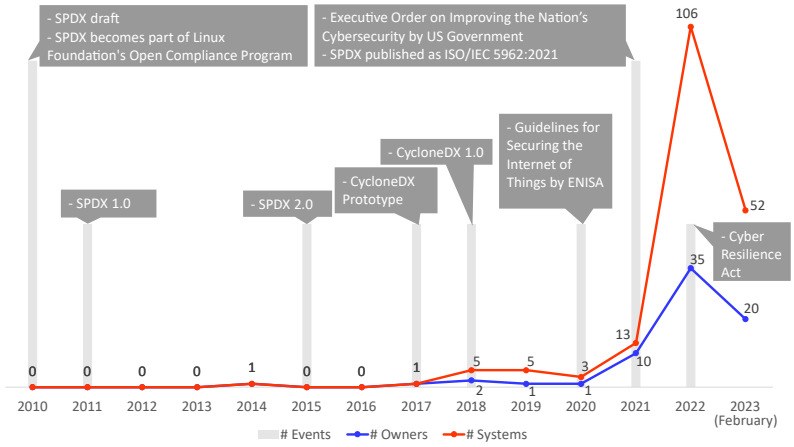


Figure 4.1: SBOM generators adoption over time by the identified software systems.

Table 4.7: Systems that have adopted SBOM generators till 2020.

Owner	System	SBOM Adoption Date
spdx	spdx-maven-plugin	2014-01-13
CycloneDX	cyclonedx-maven-plugin	2017-06-24
spdx	LicenseListPublisher	2018-01-29
stevespringett	Alpine	2018-05-02
CycloneDX	cyclonedx-core-java	2018-05-30
DependencyTrack	dependency-track	2018-09-18
DependencyTrack	hyades-apiserver	2018-09-18
ottlinger	mailclena	2019-05-02
spdx	Spdx-Java-Library	2019-06-09
DependencyTrack	frontend	2019-11-01
CycloneDX	cyclonedx-php-composer	2019-11-19
spdx	spdx-java-rdf-store	2019-12-12
spdx	tools-java	2020-01-06
spdx	spdx-java-jackson-store	2020-01-18
MO-Movia	licit	2020-10-21

hosted on GitHub since 2014. As shown in Table 4.7, initially these tools were mainly adopted by those systems whose owner was related to the SBOM domain (*e.g.*, SPDX and CycloneDX).

Since 2021, the interest in SBOM generators has increased, coinciding with (i) President Biden's Executive Order 14028 on Improving the Nation's Cybersecurity [186] and (ii) the publication of the SPDX standard as ISO/IEC 5962:2021 [86]. A further boost to the adoption of SBOM generators seems to be due to the Cyber Resilience Act proposed in 2022 [56]. Some owners were particularly prolific in the adoption of SBOMs for their repositories (blue line of Figure 4.1). However, it is worthwhile noting an increasing number of new authors are utilizing SBOM generators (orange line of Figure 4.1).

Finally, SBOMs were explicitly mentioned in the commit messages that introduced the dependencies towards SBOM generators for 90 out of 186 repositories (*i.e.*, in 48% of the studied software systems).

Answer to RQ2

SBOMs started to be adopted by open-source software systems whose owners relate to the SBOM domain; indeed, the systems that first adopted SBOMs were those created between 2014 and 2018. Since 2021, there has been a growing interest in SBOMs, even from those not related to the SBOM domain. Despite we do not have causal evidence, this seems to coincide with both President Biden's Executive Order 14028 on Improving the Nation's Cybersecurity and the publication of the SPDX standard as ISO/IEC 5962:2021. The Cyber Resilience Act proposed in 2022 seems to increase interest in SBOMs.

4.2.3 *RQ3. How do Open-source Systems Distribute SBOMs?*

Out of 186 software systems, 55 kept SBOMs under version control only, 14 distributed SBOMs in public release versions only, and 16 used both strategies to distribute SBOMs (*i.e.*, SBOMs were both under version control and in public release versions). Thus, SBOMs are available under version control or published in public release versions in 46% of the software systems identified. For the remaining 101 software systems using SBOM generators, no SBOM was found. This suggests that either SBOMs are dis-

tributed differently or those interested in SBOMs have to generate SBOMs on their own (*e.g.,* by building the software system).

As for the software systems using GitHub actions to generate SBOMs, it was observed that the generation of SBOMs is triggered by a push and pull request in 88% and 59% of the systems, respectively. Only one software system generated SBOMs during the release activity. In other words, in the systems that generate SBOMs by relying on GitHub actions, SBOMs tend to be continuously generated.

Very few software systems distributed multiple SBOMs beyond simply including them in both version control and public release versions. That is, 3% of the software systems provided SBOMs in different file formats (*i.e.,* JSON and XML). Another 3% of software systems generated distinct SBOMs to reflect the components required to run/deploy the software according to specific software/hardware (*e.g.,* SBOM tailored for different operating systems, such as Windows or Linux).

Answer to RQ3

In most systems, SBOMs are not versioned nor attached to public release versions. Hence, they are likely to be distributed differently. In the remaining cases, it appears that system owners prefer to keep SBOMs under version control rather than including them in public release versions. Systems that leverage GitHub actions to generate SBOMs, in most cases, do so at every push or pull request.

4.2.4 *RQ4. How Frequently SBOMs Get Updated within Open-source Software Systems?*

Considering those software systems with SBOMs under version control, it was observed that the median number of changes per month is 0.96. However, this number is highly variable, as indicated by the SD (1.74), min (0.03), and max (9.72) values. Overall, this suggests periodic SBOM updates, which could be associated with public release versions.

Limiting the attention to software systems that distribute SBOMs in public release versions, SBOMs appear on average

Table 4.8: Number of analyzed SPDX and CycloneDX SBOMs, along with the number of systems and owners distributing them.

Standard	# SBOMs	# Systems	# Owners
CycloneDX	89	78	16
SPDX	30	6	6
Total	*119*	*84*	*22*

in 61% of all public releases of a given software system (and 51% on median). Considering only the release versions starting from the SBOM adoption date, only three software systems (out of all 30 software systems having SBOMs in their release versions) have some release versions without SBOMs. In particular, one software system has one release version without SBOMs, another has two, and yet another has six.

Finally, we observed that only two public release versions (out of all public release versions publishing SBOMs) are tagged with a commit that modified SBOMs in the repository. There are eight more tags representing commits in which SBOMs were modified, but no release version is associated with them. Due to the low number of matches, the generation of a new SBOM did not usually occur in correspondence with a new release.

Answer to RQ4

SBOMs under version control are usually updated every month, while those distributed through release versions are usually updated at every release. In this respect, publishing new release versions does not occur at the time of SBOM generation, and release versions are typically not tagged with commits that modify SBOMs.

4.2.5 *RQ5. To What Extent do SBOMs Comply with the Standards?*

Table 4.8 shows the number of SPDX and CycloneDX SBOMs that were analyzed, along with the number of software systems and owners distributing them. 119 SBOMs were analyzed—89 referred to the CycloneDX standard, while 30 referred to SPDX—

Table 4.9: Number of selected SPDX and CycloneDX SBOMs valid with respect to their SBOM standard version.

Standard	Schema Version	Valid	Not Valid
CycloneDX	1.2	1	0
	1.3	0	1
	1.4	81	3
	1.5	3	0
SPDX	2.2	1	0
	2.3	19	10

Table 4.10: Validity errors for the analyzed SBOMs with respect to their standard version.

Standard	Error Type	Attribute	Description	# Errors
CycloneDX	Format	externalReferences	Does not match format "iri-reference"	8
	Pattern	serialNumber	Does not match pattern "^urn:uuid:[0-9a-f]{8}[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{12}$"	1
	Required	version	Version is required	1
SPDX	Unknown	unknown	Unknown	11

from a total of 84 systems and 22 owners, respectively. An SPDX SBOM, released as a plain text file, was discarded because the SPDX Online Tool failed to translate it into the JSON format.

Table 4.9 shows the number of SBOMs (valid or not) with respect to the version of the SBOM standard they had to comply with. Out of 89 CycloneDX SBOMs, 85 (96%) were valid. The invalid CycloneDX SBOMs belonged to different software systems. As for SPDX, the number of valid SBOMs was equal to 20 (67%) out of 30 SPDX SBOMs. The invalid SPDX SBOMs were all distributed with the same software system.

Table 4.10 shows the errors resulting from the SBOMs not being valid with respect to their standard version. As for CycloneDX, most errors relate to URL (Uniform Resource Locator) formatting in the `externalReferences` attribute. This attribute allows specifying external references, which provide a way to document

systems, sites, and information relevant to the objects (*e.g.*, components, services, or even SBOMs) with which they are associated. An external reference can be defined through a URL and accept any URL scheme [32]. The errors that were found are due to having reported incorrect URLs (*e.g.*, `https://github.com:ethlo/itu` or `git://github.com:appendium/flatpack.git/flatpack`). The other two types of validity errors in CycloneDX involve pattern mismatch for the value of a `serialNumber` attribute and the absence of a `version` attribute. The `serialNumber` attribute indicates the unique serial number of the SBOM, which must conform to RFC-4122.[1] As for the version attribute, its value indicates the version number of the SBOM: it is one by default and should be incremented by one whenever the existing SBOM is modified.

As for SPDX, the error type returned by the sbom-utility tool was always `unknown`—*i.e.*, the tool could not comprehend the reasons why these SBOMs were not valid. Therefore, these SPDX SBOMs were manually checked to understand the reasons for unknown errors. We found that these SBOMs contained four attributes (`predicate`, `statement`, `envelope`, and `bundle`) not belonging to the SPDX standard, and one of them (*i.e.*, `predicate`) contained the SPDX SBOM data. This is because these SBOMs were compliant with the in-toto attestation framework, which allows securing the integrity of software supply chains [181, 214].

The above-mentioned validity errors might negatively affect tools consuming SBOMs depending on the kind of errors, the SBOM use case, and/or the SBOM consumption tool. For instance, it would be impossible to check for the presence of vulnerabilities in the case of the SPDX SBOMs with unknown validity errors (the tool failed with the following error message: `[ERROR] format not supported: unknown format`). On the other hand, this tool would not fail for the CycloneDX SBOMs having the validity errors reported above. However, SBOM consumption tools supporting other SBOM use cases might fail. The causes behind the validity errors that were found can be different. For instance, the unknown validity errors are due to the use of the in-toto

1 The RFC-4122 specification defines a Uniform Resource Name (URN) namespace for UUIDs (Universally Unique IDentifier), which are 128 bits long and can guarantee uniqueness across space and time [91].

Table 4.11: NTIA's minimum data fields discovered in the analyzed SBOMs (in round brackets, occurrences of non-asserted data fields).

Data Field	Standard	SBOMs		Systems		Owners	
		#	%	#	%	#	%
Supplier name	CycloneDX	7	*8*	7	*9*	4	*25*
	SPDX	1 (1)	*3 (3)*	1 (1)	*17 (17)*	1 (1)	*17 (17)*
Component name	CycloneDX	80	*90*	70	*90*	16	*100*
	SPDX	30	*100*	6	*100*	6	*100*
Component version	CycloneDX	80 (1)	*90 (1)*	70 (1)	*90 (1)*	16 (1)	*100 (6)*
	SPDX	30	*100*	6	*100*	6	*100*
Other unique identifiers	CycloneDX	89	*100*	78	*100*	16	*100*
	SPDX	30	*100*	6	*100*	6	*100*
Dependency relationship	CycloneDX	78	*88*	68	*87*	14	*88*
	SPDX	1	*3*	1	*17*	1	*17*
Author of SBOM data	CycloneDX	87	*98*	77	*99*	16	*100*
	SPDX	30 (1)	*100 (3)*	6 (1)	*100 (17)*	6 (1)	*100 (17)*
Timestamp	CycloneDX	86	*97*	76	*97*	15	*94*
	SPDX	30	*100*	6	*100*	6	*100*

attestation framework. As for the incorrect values of attributes (*i.e.,* incorrect formats and patterns), this could originate from the data sources that the SBOM generators use to generate SBOMs.

Answer to RQ5

A limited fraction of SBOMs exhibit validity errors with respect to their SBOM standard version. These errors concern mismatched patterns for the values specified for some attributes and invalid SBOM structure. These validity errors might compromise SBOM consumption.

4.2.6 *RQ6. What are the Pieces of Information Reported by the Found SBOMs?*

Table 4.11 reports, for each NTIA's minimum data field, the number (and percentage) of SBOMs reporting it, along with

the number (and percentage) of systems and owners involved. The number of SBOMs, systems, and owners containing *non-asserted* data fields—*i.e.*, those for which the value is not provided (*e.g.*, *none* or *null* value)—is reported in round brackets. The analyzed SPDX SBOMs reported all NTIA's minimum data fields at least once but the *supplier name* and *dependency relationship*, appearing in only one SBOM. Therefore, only 3% ($1/30 * 100$) of the analyzed SPDX SBOMs reported all NTIA's minimum data fields. As for CycloneDX SBOMs, the supplier name appeared at least once in only seven SBOMs, resulting in the least appearing data field across CycloneDX SBOMs. These seven CycloneDX SBOMs reported at least once all NTIA's minimum data fields. This is to say that only 8% ($7/89 * 100$) of the analyzed CycloneDX SBOMs reported all NTIA's minimum data fields. The only data field that all CycloneDX SBOMs reported at least once is *other unique identifiers*. Only two and three CycloneDX SBOMs did not report any occurrence of the *author of SBOM data* and *timestamp* data fields, respectively. As for *dependency relationships*, this data field is reported at least once in 78 CycloneDX SBOMs (out of 89). The *component name* and *component version* data fields are reported at least once in 80 CycloneDX SBOMs. This indicates that nine CycloneDX SBOMs did not report any components. The presence of SBOMs that do not report components highlights an improper use of SBOMs. However, considering the number of owners, all of them reported the *component name* and *component version* data fields in some of their SBOMs.

Table 4.12 shows the different kinds of *other unique identifiers* used to identify software components and even SBOMs, along with the number (and percentage) of SBOMs, systems, and owners reporting them. As for CycloneDX SBOMs, purl is the most reported identifier for software components, appearing in 90% ($80/89 * 100$) of them. Only one CycloneDX SBOM reports two CPE identifiers. As for SPDX SBOMs, the most introduced identifier for software components is SPDX ID, which appears in all of them. To follow, purl and CPE appeared in 97% ($29/30 * 100$) and 93% ($28/30 * 100$) of SPDX SBOMs. In addition, each SPDX SBOM declared an SPDX ID to identify the SBOM (which always assumes the value `SPDXRef-DOCUMENT`). Similarly, each SPDX SBOM provided a document-specific namespace as a URI (Uniform Re-

Table 4.12: Identifier information reported for *other unique identifiers* data field.

Standard	Identifier Type	Target	SBOMs		Systems		Owners	
			#	%	#	%	#	%
CycloneDX	CPE	Software component	1	*1*	1	*1*	1	*6*
	purl	Software component	80	*90*	70	*90*	16	*100*
	UUID	Software component	53	*60*	53	*68*	3	*19*
		SBOM	89	*100*	78	*100*	16	*100*
SPDX	CPE	Software component	28	*93*	4	*67*	4	*67*
	SPDX ID	Software component	30	*100*	6	*100*	6	*100*
		SBOM	30	*100*	6	*100*	6	*100*
	purl	Software component	29	*97*	5	*83*	5	*83*
	URI	SBOM	30	*100*	6	*100*	6	*100*

Table 4.13: NTIA's recommended data fields discovered in the analyzed SBOMs (in round brackets, the number of non-asserted data fields is reported).

Data Field	Standard	SBOMs		Systems		Owners	
		#	%	#	%	#	%
Component hash	CycloneDX	26	*29*	16	*21*	13	*81*
	SPDX	19	*63*	4	*67*	4	*67*
Life cycle phase	CycloneDX	0	*0*	0	*0*	0	*0*
	SPDX	0	*0*	0	*0*	0	*0*
Other component relationships	CycloneDX	0	*0*	0	*0*	0	*0*
	SPDX	30	*100*	6	*100*	6	*100*
License information	CycloneDX	30	*34*	19	*24*	14	*88*
	SPDX	30 (30)	*100 (100)*	6 (6)	*100 (100)*	6 (6)	*100 (100)*

source Identifier). Also, all CycloneDX SBOMs have an instance of UUID (Universally Unique IDentifier) to identify SBOMs. UUID identifiers are also used to identify software components in 60% ($53/89 * 100$) of the Cyclonedx SBOMs.

Similarly to NTIA's minimum data fields, Table 4.13 reports, for each NTIA's recommended data field, the number (and percentage) of SBOMs reporting it, along with the number (and percentage) of systems and owners involved. Only 29% ($26/89 * 100$)

Table 4.14: Component hash information reported in the SBOMs.

Standard	Hashing Algorithm	SBOMs		Systems		Owners	
		#	%	#	%	#	%
CycloneDX	MD5	7	8	7	9	4	25
	SHA-1	8	10	8	10	5	31
	SHA-256	25	28	15	19	12	75
	SHA3-256	6	7	6	8	3	19
	SHA3-384	6	7	6	8	3	19
	SHA3-512	6	7	6	8	3	19
	SHA-384	7	8	7	9	4	25
	SHA-512	8	9	8	10	5	31
SPDX	SHA-1	1	3	1	17	1	17
	SHA-256	18	60	3	50	3	50

of the CycloneDX SBOMs report at least once a component hash; however, the percentage of CycloneDX owners reporting component hash is high, namely 81% ($13/16 * 100$). As for SPDX SBOMs, the percentages of SBOMs and owners reporting component hash at least once are similar, namely 63% ($19/30 * 100$) and 67% ($4/6 * 100$), respectively. Delving deeper into the analysis of the component hash data field, Table 4.14 conveys information on the different kinds of hashing algorithms used. The most commonly used hashing algorithm is SHA-256 for both SPDX and CycloneDX SBOMs. It is worth noting that multiple hashes can be provided for the same component. For instance, SHA3-256 hashes are always reported along with SHA3-384 and SHA3-512 ones. This might come in handy when external sources/databases support different hash types. In this respect, it was verified that, across all SBOMs, each component was identified by only one hash value for each different hashing algorithm. However, it was found that the same hash value could map to different components. More specifically, it was found that this happened only once in CycloneDX SBOMs and involves the following two components:

- `pkg:maven/io.grpc/protoc-gen-grpc-java@1.53.0?classifier=osx-aarch_64&type=exe`

Table 4.15: Types of dependencies reported in the SPDX SBOMs.

Relationship Type	SBOMs		Systems		Owners	
	#	%	#	%	#	%
Describes (SPDXRef-DOCUMENT describes SPDXRef-A)	29	97	5	83	5	83
Dynamic link (SPDXRef-A dynamically links to SPDXRef-B)	2	7	2	33	2	33
Generates (SPDXRef-A generates SPDXRef-B)	1	3	1	17	1	17
Test case of (SPDXRef-A is a test case used in testing SPDXRef-B)	1	3	1	17	1	17

- `pkg:maven/io.grpc/protoc-gen-grpc-java@1.53.0?classifier=osx-x86_64&type=exe`

As shown above, these components have similar purl identifiers, differing only in one parameter specifying the kind of operative system architecture they are compliant with. In other words, these two purl identifiers refer to the same component. As for SPDX SBOMs, 210 hash values were found to map more than one component each, all associated with CPE identifiers. We verified whether these CPE identifiers mapped by the same hash values were similar to each other—for instance, what changes is the use of a different character (*e.g.,* an underscore instead of a dash) or the presence of a few extra words specifying something more about the component. These CPE identifiers refer to the same component. Having different CPE (or purl) identifiers referring to the same component might be an issue since external sources/-databases might not know all CPE (or purl) identifiers for a given component. Given a larger number of cases using the same hash value mapping different components identified by CPE, it might be inferred that purl allows for a more robust representation of components than CPE.

As shown in Table 4.13, no SPDX or CycloneDX SBOM ever reported the *life cycle phase* data field; however, from the used SBOM generators, it can be inferred that the SBOM generation happens within- or post-building. As for the *other component relationship* data field, no occurrence was ever found in CycloneDX SBOMs, while all SPDX SBOMs reported some introductions. Table 4.15 shows the different kinds of relationships (different

Table 4.16: Number of direct and transitive dependencies reported in the SBOM files.

Dependency	Standard	SBOMs		Systems		Owners	
		#	%	#	%	#	%
Direct	CycloneDX	23	*26*	13	*17*	10	*63*
	SPDX	30	*100*	6	*100*	6	*100*
Transitive	CycloneDX	23	*26*	13	*17*	10	*63*
	SPDX	2	*7*	2	*33*	2	*33*

from dependency) specified in the SPDX SBOMs, along with a brief description according to the SPDX documentation [150]—note that `SPDXRef-A` and `SPDXRef-B` refer to generic components (or also files), while `SPDXRef-DOCUMENT` to the SBOM itself. The most common relationship different from dependency was `describes`, which appeared in 29 SPDX SBOMs, each with one occurrence. `Dynamic link` appeared in only two SPDX SBOMs. Both `generates` and `test case of` appeared in one SBOM only.

Considering the *license information* data field (see Table 4.13), at least one occurrence was found in 34% (30/89*100) of the analyzed CycloneDX SBOMs, while all SPDX SBOMs reported license information at least once.

Table 4.16 reports the number (and percentage) of SBOMs having direct and transitive dependencies within their dependency graphs, along with the number of systems and owners involved. First, the number of SBOMs describing the *primary component* (*i.e.*, the software system for which the SBOM was generated) in the sections reporting component information is 26 for CycloneDX and 30 for SPDX. Every node in the dependency graphs of these SBOMs is also described in the sections reporting component information. However, the opposite is not true. Namely, cases where a component was described in the section reporting component information was found but it did not appear as a node in the dependency graph. All 23 CycloneDX SBOMs having direct dependencies also have some transitive dependencies; on the other hand, only 2 SPDX SBOMs have transitive dependencies. Each SPDX SBOM reported the primary component as a node in the dependency graph, while only 23 CycloneDX SBOMs do

Table 4.17: NTIA's minimum and recommended data fields discovered in the analyzed SBOMs. In round brackets, it is reported the number of non-asserted data fields.

Data Field	SBOMs		Systems		Owners	
	#	%	#	%	#	%
Life cycle phase	0	0	0	0	0	0
Supplier name	8 (1)	7 (1)	8 (1)	10 (1)	5 (1)	23 (5)
Other component relationships	30	25	6	7	6	27
Component hash	45	38	20	24	17	77
License information	60 (30)	50 (25)	25 (6)	30 (7)	20 (6)	91 (27)
Dependency relationship	108	91	74	88	20	91
Component name	110	92	76	90	22	100
Component version	110 (1)	92 (1)	76 (1)	90 (1)	22 (1)	100 (1)
Timestamp	116	97	82	98	21	95
Author of SBOM data	117 (1)	98 (1)	83 (1)	99 (1)	22 (1)	100 (5)
Other unique identifiers	119	100	84	100	22	100

it (see the number of SBOMs having direct dependencies). However, we observed that not all nodes were linked to the primary component nodes in the dependency graphs. In particular, a path between the primary component node and any other node was found in only 20 CycloneDX and 28 SPDX SBOMs, respectively.

Table 4.17 provides an overview of how frequently NTIA's minimum and recommended data fields were discovered in all analyzed SBOMs, without distinguishing per standard. The data fields are sorted from the least discovered to the most discovered in SBOMs. The only data field that is completely missing is *life cycle phase*, which may suggest that the considered SBOM generators do not provide support for generating this data field. *Supplier name* was discovered in only 7% ($8/119 * 100$) of SBOMs, and, in one of them, its value might not be asserted. This means that very few SBOMs include supplier information.

The *other component relationships* were reported in 25% of SBOMs ($30/119 * 100$), while *component hash* of 38% of them ($45/119 * 100$). *License information* was discovered in 50% of SBOMs ($60/119 * 100$), and in 25% ($30/119 * 100$) possibly with non-asserted values. The remaining data fields appear in more than 90% of SBOMs. In detail, *dependency relationship* was discovered in 91% of SBOMs ($108/119 * 100$), *component name* and

component version in 92% (110/119 ∗ 100), *timestamp* in 97% (116/119 ∗ 100), and *author of SBOM data* in 98% (117/119 ∗ 100). The presence of one non-asserted instance of *component version* and *author of SBOM data* involves just one SBOM. *Other unique identifiers* was the only data field always discovered in the analyzed SBOMs.

As shown in Table 4.17, similar considerations can also be made for systems and owners on how frequently NTIA's minimum and recommended data fields were discovered in all analyzed SBOMs. This omission of information about NTIA's minimum and recommended data fields would hinder the pursuit of basic use cases (*e.g.*, handling software inventory or verifying license compliance) as well as advanced use cases (*e.g.*, tracking software provenance or pedigree) for the identified software systems.

Finally, no vulnerability information was reported in either CycloneDX or SPDX SBOMs. Providing information about vulnerabilities has been advocated in previous work [170, 182, 195]. Moreover, a recent study experimented with the usefulness of vulnerability-augmented SBOMs [67].

Answer to RQ6

Only a few SBOMs report information on all NTIA's minimum data fields, with some not reporting any component information. SBOMs mostly use purl and then CPE to identify components; no occurrence of SWID tags has ever been found. As for NTIA's recommended fields, no SBOM reports information on all of them, as no information on the life cycle phase was ever reported. No vulnerability has ever been reported in any SBOM.

4.3 DISCUSSION

In the following, the main findings of the study are summarized, and implications for software suppliers, choosers, operators, and researchers are delineated.

Finding #1

Based on systems' dependencies, a limited number of open-source software systems adopt SBOM generators. However, such a number is progressively increasing.

Although the study was based on 35 SBOM generators officially released by SPDX or CycloneDX—the two major SBOM standards [195]—, only 186 open-source systems adopting SBOMs were identified. This suggests that SBOMs are still neglected by the open-source community, probably due to the lack of interest and awareness in the software supply chain among software suppliers and choosers. The limited adoption of SBOMs can be considered consistent with what was reported by Xia *et al.* [195]. It can be conjectured that the low adoption of SBOMs in open-source systems might be due to the lack of incentives for generating SBOMs [99, 170, 195]. For example, open-source software systems may not be required to have an SBOM [170], unlike software systems commercialized to the United States federal agencies [186]. However, open-source software systems may soon be pressured to supply an SBOM because commercialized software could depend on them [178]. The adoption of SBOMs is expected to grow due to the demand of major players, including governmental organizations. Software suppliers cannot wait any longer and should begin to identify effective strategies for producing and distributing their SBOMs. Software suppliers should begin integrating SBOM generation into their development pipelines, anticipating possible regulations that could mandate SBOM compliance. For example, SBOM generators like cyclonedx-maven-plugin or spdx-maven-plugin can be integrated with minimal effort in systems already using Maven as a package manager.

Finding #2

The preferred way to generate SBOMs is through build-integration tools, followed by GitHub Action. The generation of SBOMs via libraries is less common than in the ways mentioned above.

As pointed out by Xia *et al.* [195], one of the challenges restricting the adoption of SBOMs is the limited availability of suitable tools and the features offered by such tools. However, standard creators (*i.e.,* SPDX and CycloneDX) and forges (*e.g.,* GitHub) are working to make available to software suppliers convenient tools for SBOM generation. In particular, a wider adoption of SBOMs via build-integration tools (*e.g.,* Maven) and CI/CD tools (*i.e.,* GitHub Action) was observed compared to the adoption of SBOMs via libraries. This is probably due to the ease of use of both build-integration and CI/CD tools: the former allows SBOMs to be generated when building a software system, while the latter when a certain event occurs (*e.g.,* push). Moreover, it can be expected an increase in the adoption trend of SBOMs since, recently (*i.e.,* on March 28, 2023), GitHub has introduced a new feature to export SBOMs that allows anyone with read access to a GitHub repository to generate an SPDX SBOM by clicking on a button on the dependency graph page of the repository [75]. This self-service feature not only enables on-demand SBOM generation but could also be integrated into CI/CD pipelines and build-automation processes. Therefore, we can expect that more open-source software systems will adopt SBOMs in the future.

One potential concern might be software suppliers excluding SBOM generation from the CI/CD pipeline—thereby avoiding continuous SBOM generation—due to possible performance overhead. However, considering the SBOM granularity provided by state-of-practice SBOM generators (*i.e.,* package level) and the way such tools work (*i.e.,* by leveraging package managers), once the CI/CD pipeline has resolved the direct and indirect dependencies (necessary to perform the build anyway), the SBOM generation would take a few seconds even for relatively large systems. The reason is that package managers have already retrieved, during dependency resolution, the required information for SBOM generation. In other words, such minimal overhead would perfectly align with CI/CD practices [49]. The same applies to software systems leveraging GitHub's SBOM generation feature. Clearly, if a software supplier needs to keep up-to-date SBOMs for software written in programming languages for which there is no package manager, then such a task may become cumbersome and left to the developers' responsibility.

Further recommendations emphasize the importance of integrating SBOM generators into the development pipeline. Embedding such tools is important for software suppliers to enhance the transparency and security of their software systems. Recent initiatives, including GitHub's native SBOM generation feature [75], represent a promising step toward mainstream adoption, enabling developers to automatically generate and maintain SBOMs as part of their CI/CD pipeline.

Finding #3

Only a few software systems follow recommendations on how SBOMs should be distributed.

According to NTIA [117], software suppliers should bundle SBOMs with every release version of the software systems, as well as update them at each code change. Furthermore, there is a broad consensus on this matter among the respondents of the Linux Foundation's SBOM readiness survey [171]. Nevertheless, it appears that most of the studied systems ignore or are unaware of such recommendations, thus limiting the potential benefits due to the adoption of SBOMs. Advice for software suppliers is to distribute SBOMs following the NTIA's recommendations [117], including delivering updated SBOMs at each code change and release version. Consistent and timely distribution of SBOMs is also essential for maintaining transparency, enabling effective vulnerability management, and supporting software operators in assessing system risks.

Finding #4

The same software system may distribute multiple SBOMs.

Multiple SBOMs are distributed for the same software system because:

- SBOMs are provided in different formats (*e.g.,* JSON or XML).
- Different SBOMs may be generated to represent the specific hardware or software configurations (*e.g.,* operating

systems or processor architectures) required to deploy/run the system.

- SBOMs can be maintained both under version control and in public release versions of a software repository.

Software suppliers should generate SBOMs using widely adopted and well-supported file formats, such as JSON or XML. Since no universally accepted file format exists, using multiple file formats can improve interoperability across the SBOM toolchain. For example, tools such as sbom-utility [36] support only the JSON format. While tools like cyclonedx-cli [35] can convert SBOMs between file formats, whenever possible, SBOMs should be generated natively in the required file formats, ideally using tools that support multi-format outputs (*e.g.*, cyclonedx-maven-plugin supports both JSON and XML file formats). Moreover, software suppliers should distribute SBOMs tailored to each supported hardware or software configuration to accurately represent the components required to deploy/run that software. Generating distinct SBOMs for different deployment targets enhances the precision of the reported components. This practice facilitates more accurate vulnerability management, license compliance, and other SBOM-driven use cases [115].

Finding #5

Not all SBOMs are valid with respect to their SBOM standard version, thus potentially compromising the capability of tools consuming SBOMs.

When SBOMs are not valid with respect to their SBOM standard version, tools consuming invalid SBOMs (*e.g.*, vulnerability or license scanners) might fail their ingestion. This would limit the potential benefits due to the adoption of SBOMs. To let software choosers and operators get the best out of SBOMs, software suppliers should check the compliance of the SBOMs they generate. To that end, convenient tools like sbom-utility [36] might come in handy to them. Software suppliers could incorporate SBOM validation tools like sbom-utility into their CI/CD pipelines to ensure that generated SBOMs conform to their declared standard and version (*e.g.*, SPDX 2.3 or CycloneDX 1.5).

Validity errors could originate from inaccuracies in SBOM generators or data sources from which information is retrieved to feed SBOMs. For example, a data source could be populated with incorrect information that, if retrieved to feed an SBOM, would make that SBOM not valid with respect to its standard version. In this respect, researchers should conduct further investigations.

Finding #6

SBOMs can integrate with software supply chain frameworks.

Some SPDX SBOMs have been found to integrate with the in-toto attestation framework [213, 214], which provides a specification for generating verifiable claims about the software supply chain of a software artifact. Frameworks like in-toto are also supported by CycloneDX SBOMs [33]. On the one hand, integrating SBOMs with software supply chain frameworks highlights their flexibility, allowing SBOMs to be tailored to specific needs. On the other hand, such integrations may introduce deviations from the standard SBOM schema. For instance, the in-toto integration adds attributes to SBOMs that fall outside the scope of the standard specification, which could lead to errors when these SBOMs are processed by tools that expect strict adherence to the SBOM standard. Software suppliers should integrate SBOMs with software supply chain frameworks while carefully managing potential errors that may arise when consuming SBOMs that deviate from the standard specification.

Finding #7

Most SBOMs do not report all NTIA's minimum data fields; in this respect, the supplier name of components is the least reported data field, and some SBOMs do not report any component information.

NTIA's minimum data fields document baseline information about each component that should be tracked and maintained. This baseline information is needed to uniquely and unambiguously document components and then enable (i) the tracking of components across the software supply chain and (ii) their

mapping to other beneficial sources of data (*e.g.,* vulnerability or license databases) [185]. It is easy to grasp that not reporting this baseline information, as it was observed, compromises basic use cases like the management of vulnerabilities, software inventory, and licenses, as well as advanced use cases (*e.g.,* software provenance and pedigree). This is clearly of interest to software suppliers, choosers, and operators. This finding is also consistent with previous literature [182, 195]. According to Xia *et al.* [195], one of the reasons for not reporting all NTIA's minimum data fields is the unavailability of data to feed SBOMs. However, when this happens, software suppliers should explicitly assert it [185]. Instead, a limited number of non-asserted data fields were witnessed. Software suppliers should treat NTIA's minimum data fields as non-negotiable. In case a data field cannot be provided (*e.g.,* the supplier name is unknown), they should explicitly assert the missing/unavailable information as per NTIA's guidelines [185]. This would promote transparency for the software system documented by the SBOM. In critical or security-sensitive systems, software suppliers should consider post-processing or enriching generated SBOMs to fill in missing fields, especially in the case of known vulnerabilities.

Regarding the supplier name, the poor reporting of this data field is likely due to the shortage of global identification mechanisms like those used for components (*e.g.,* CPE or purl). Similar to NTIA [118], researchers should define mechanisms to globally identify suppliers (similar to those used for components): this should ease the retrieval of information on them from external databases when producing SBOMs.

Finally, some SBOMs did not report any component information. This finding suggests a very incorrect and shallow use of SBOMs, which makes these SBOMs less useful for both software suppliers, choosers, and operators.

Finding #8

Overall, only half of the SBOMs report licensing information. The presence of licensing information is more common in SPDX SBOMs than CycloneDX ones.

We can postulate that the problem of poor reporting of licensing information in SBOMs is due to the limitations of SBOM generators in retrieving such information. Although a comparative study between SPDX and CycloneDX SBOM generators was not performed, the higher prevalence of licensing information in SPDX SBOMs seems to suggest that SBOM generators owned by SPDX are more effective than those owned by CycloneDX in retrieving licensing information to feed SBOMs. This finding is not surprising since the SPDX standard was initially conceived to improve software license compliance, while the CycloneDX standard is developed and maintained by OWASP, which operates in the security context.

Another reason for the poor reporting of licensing information in SBOMs can be the lack of interest in licensing issues by system owners, most of whom have been using CycloneDX SBOM generators. The advice to software suppliers is not to underestimate the importance of documenting licensing in their SBOM to facilitate licensing compliance analysis from the software chooser's side (*e.g.,* those who want to integrate a library in their systems). Software suppliers should ensure that every component in their SBOMs includes license information and, eventually, add it manually in case of a missing license. Neglecting license compliance could expose organizations to legal and financial liabilities, especially in commercial and enterprise contexts where the reuse of open-source components is audited.

Finding #9

A limited fraction of SBOMs report component hashes; those reporting more hash values (using different hashing algorithms) for the same components are even less.

As reported by NTIA [185], robust identifiers are important for mapping the existence of components to external data sources (*e.g.,* vulnerability database). A component hash would provide a foundational element to allow this mapping, besides assisting in instances of component renaming and whitelisting. Component hashes also allow verifying that components have not been altered in an unauthorized fashion (*i.e.,* component integrity verification). Moreover, providing more hashes (using

different hashing algorithms) for the same components enhances interoperability—*i.e.,* it increases SBOM compatibility with third-party tools or external data sources working with specific kinds of hashes [12]. Summing up, the recommendation to software suppliers is to always provide more than one hash (each generated with a different hashing algorithm) for each component. Increasing SBOM compatibility would be beneficial to software choosers and operators as well.

Finding #10

The most reported kind of component identifier in SBOMs is purl, followed by CPE. No component was ever identified via SWID tags.

The higher prevalence of purl in SBOMs, as compared to CPE, is not surprising since CPE identifiers are assigned by a central authority (*i.e.,* NIST), while purl identifiers are not. This is to say that some components might not yet have CPE identifiers. Nevertheless, it is worth recalling that: (i) component identifiers like purl, CPE, and SWID tags are foundational elements to retrieve information on components in external data sources; and (ii) different data sources can support different kinds of component identifiers—*e.g.,* while NVD exploits CPE for component identification, Sonatype OSS Index exploits purl. Similar to the recommendation on component hashes, a suggestion is to report software components in SBOMs using different kinds of identifiers. This would be beneficial to software suppliers and operators. On the other hand, enhancements in SBOM generation and consumption tools are welcome so that these can support different kinds of component identifiers.

Finding #11

Most SBOMs (all concerning CycloneDX) do not describe primary components. Also, they do not report primary component nodes within dependency graphs.

According to NTIA [114], an SBOM must describe at least one primary component, which represents the subject of the SBOM. The declaration of the primary component is needed to properly

document direct and transitive dependencies. Also, an SBOM should contain at least all primary component dependencies (*i.e.,* the direct dependencies), with all transitive dependencies listed or sufficient detail to retrieve transitive dependencies recursively [185]. Therefore, reporting information on the primary component is crucial to understanding which dependencies are direct or transitive and then adequately supporting SBOM use cases, such as vulnerability and licensing risk management. Software suppliers should list the primary components and their direct dependencies. This would produce benefits for software choosers and operators as well.

Finding #12

Although previous work indicated the usefulness of augmenting SBOMs with vulnerability information, such information was never found in the analyzed systems.

It is recommended to store vulnerability information in separate data structures linked to SBOMs (*i.e.,* VEX documents) unless very specific conditions and processes are in place to make SBOMs constantly complete and up-to-date [32, 185]. Therefore, possible reasons for not reporting vulnerabilities in the studied SBOMs might be:

- Information on vulnerabilities, including their exploitability, is stored in separate data structures linked to SBOMs (*i.e.,* VEX documents). However, no occurrence of VEX documents conforming to the CycloneDX standard was found; accordingly, this reason seems unlikely.
- Owners of software systems are not interested in publicly disclosing information on vulnerabilities in their software supply chain.
- Owners of software systems would like to disclose this information, but are worried that vulnerability information can be used as a road map for attackers.
- SBOM generators lack support for retrieving and producing up-to-date information on vulnerabilities in the software supply chain.

The last three points were also raised in past studies [67, 170, 195, 199]. In particular, regarding the last point, Fucci *et al.* [67] found that information on vulnerabilities might be reported by security tools already in use by open-source software systems, and such information could not be easily integrated into SBOMs (*e.g.*, due to a lack of standardization). Following NTIA recommendations [116], software suppliers should integrate information on vulnerabilities in SBOMs (or by using data structures linked to SBOMs) since attackers can still obtain it in other ways. For example, software suppliers can link SBOMs to vulnerability information using VEX. This would enable the consumption of SBOM and VEX within CI/CD pipelines, which can periodically check for newly disclosed vulnerabilities and alert developers when any risks emerge [67]. Researchers should improve SBOM generators to retrieve information on vulnerabilities and generate better SBOMs.

4.3.1 *Threats To Validity*

This section discusses potential threats to the validity of the results. According to the validity schema by Wohlin *et al.* [194], these threats can be categorized as construct, external, conclusion, and internal validity. Threats to internal validity are not discussed because such threats concern causal relationships [194], and causality was not investigated in this mining study.

Threats to *construct validity* concern the relation between theory and observation [194]. The approach that was used to detect the adoption of SBOMs might affect the results. In this respect, the adoption of SBOMs could be detected by using one of the following approaches: (i) by identifying the use of SBOM generators through GitHub's dependency graph or (ii) by searching for SBOMs within systems hosted on GitHub. We followed the first approach because a system could use an SBOM generator without versioning any SBOM or attaching SBOMs to any release version (as observed in 101 out of 186 systems, see Section 4.2.2). However, the accuracy of GitHub's dependency graph might affect the obtained results [44, 45]. To deal with this issue, we ensured that the repositories were actually dependent on the SBOM generators by verifying that their dependencies appeared

in their dependency graph and package manager files. This resulted in the exclusion of 11 repositories. Another threat is related to the fact that GitHub's dependency graph does not support all build-automation tools (*e.g.,* Gradle is not supported by GitHub's dependency graph but supported by CycloneDX) and programming languages (*i.e.,* those without a package manager like C, which is not supported by GitHub's dependency graph but supported by CycloneDX). Last but not least, some systems might still adopt SBOMs, yet there is no visible trace of dependencies towards SBOM generators on GitHub (*e.g.,* because developers installed them on their machine), nor any presence of SBOM files in the repository (*e.g.,* because the SBOM is provided when the system releases are redistributed through channels different from GitHub). Another threat to construct validity pertains to the identification of systems related to the domain of SBOMs. This was accomplished by looking at the terms "SBOM," "CycloneDX," "SPDX," "Bill-of-Materials," and "Supply-Chain" in the repository's name, description, and README file. However, the presence of these terms does not ensure that these systems are related to the domain of SBOMs. The metrics used to answer the RQs might pose another threat to construct validity. However, there is no accepted metric to quantitatively assess the adoption of SBOMs. At the same time, NTIA has outlined a set of required and recommended SBOM data fields [114, 119], and therefore, the assessment was based on the coverage of such fields. To some extent, the defined and used metrics represent another scientific contribution of this research.

Threats to *external validity* concern the generalizability of results [194]. The most important threat is that this study is limited to (i) open-source systems hosted on GitHub and (ii) SBOM generators owned by SPDX and CycloneDX. The current study does not observe SBOM practices in closed-source systems. Also, it does not observe SBOM practices and content when the GitHub SBOM generator is used. This is because (i) the study is based on data extracted before such a tool was released, and (ii) such a tool can be virtually used on any GitHub repository (provided that it has a dependency graph), and therefore searching repositories through dependencies towards SBOM-related tools would not work. Systems whose package manager was not supported by

GitHub's dependency graph could not be examined (for example, systems developed in C/C++). While the gathered empirical evidence on the adoption, spread, and evolution of SBOMs cannot be considered conclusive, this evidence can justify further research.

Threats to *conclusion validity* concern the ability to draw the correct conclusions [194]. Since the study is exploratory, the RQs were addressed using descriptive statistics and graphical representations. Another threat to conclusion validity relates to the analysis of the SBOM data fields, as it relies on the correct interpretation of the data fields reported in the identified SBOMs. This threat was mitigated by following the mapping of SBOM data fields provided by NTIA [114, 119]. The adopted SBOM generators determined the content of the identified SBOMs (*e.g.*, [197]), which may have impacted the study results.

DATA AVAILABILITY

The replication package of the study is available online [125].

5

EMPIRICAL EVALUATION OF GITHUB'S SBOM GENERATOR

This chapter is based on the paper entitled "An empirical study on the accuracy of GitHub's dependency graph and the nature of its inaccuracy" [45], published in the Information and Software Technology, which extends the work presented in "On the Accuracy of GitHub's Dependency Graph" [44], published in the proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering.

One of the key barriers to the widespread adoption of SBOMs lies in the limitations of existing SBOM generators. To investigate these limitations, this chapter presents an empirical evaluation of GitHub's SBOM generator—a feature introduced on March 28, 2023, and expected to see broad adoption due to GitHub's popularity and the large number of software systems it hosts. To assess the accuracy of this SBOM generator, the empirical evaluation focused on the underlying dependency graph used to generate the SBOMs. The evaluation was conducted from two complementary perspectives: (i) a backward analysis, focusing on the accuracy of the dependencies, as they are reported in GitHub's dependency graph of a given repository; and (ii) a forward analysis, focusing on the accuracy of the dependents. On top of such analyses, the dependencies extracted from GitHub's dependency graph were compared with those declared in the corresponding manifest and lock files. Beyond these quantitative analyses, a statistically significant sample of mismatches in the dependency graph was qualitatively analyzed to identify possible error causes. It is worth mentioning that the empirical evaluation of GitHub's dependency graph might have implications not only for guiding the development of more reliable SBOM generators but also for helping practitioners assess how much trust they can place in GitHub and similar software forges when relying on them as sources of dependency information.

CHAPTER STRUCTURE. Section 5.1 presents the overall study design. The data analyses and results are reported in Section 5.2 while the result discussion concludes the chapter in Section 5.3.

5.1 STUDY DESIGN

The *goal* of this study is to analyze GitHub's dependency graph with the *purpose* of (i) assessing its accuracy and (ii) identifying the likely causes behind inaccuracies (if any) in GitHub's dependency graph. The *perspective* is that of software suppliers, choosers, and operators who rely on GitHub's dependency graph or any tool built upon it, including GitHub's SBOM generator. The *context* consists of open-source software systems hosted on GitHub written in Java and Python, using Maven and pip/Poetry as package managers, respectively.

5.1.1 *Research Questions*

Based on the above-mentioned study goal, two RQs were formulated and then studied:

RQ1. *To what extent does GitHub's dependency graph provide accurate information?*

This RQ assesses the accuracy of both dependencies and dependents provided by GitHub's dependency graph. From the perspective of software suppliers, choosers, and operators, an inaccurate graph may result in incomplete or unreliable SBOMs, which would jeopardize the management of the software supply chain (*e.g.*, undetected vulnerabilities or license compliance issues might lead to security incidents or legal liabilities).

LONGITUDINAL ANALYSIS To understand if the accuracy of GitHub's dependency graph changed over time, we compared two snapshots:

- **First snapshot.** This is the snapshot of the analyses performed in the conference paper [44] on the accuracy of

the dependencies and dependents provided by GitHub's dependency graph. This snapshot was taken in April 2023.

- **Second snapshot.** This is the last snapshot of the analyses on the accuracy of the dependencies and dependents provided by GitHub's dependency graph. This snapshot was taken in September 2024.

The comparison mentioned above should allow for the discovery of whether the accuracy of GitHub's dependency graph has improved, regressed, or remained relatively stable over time.

RQ2. *What are the likely causes for the error cases occurring in GitHub's dependency graph?*

This RQ aims to investigate the error cases encountered during the analysis of GitHub's dependency graph, to deliver to software suppliers, choosers, and operators the likely causes behind them. Understanding the likely error causes would provide insights into possible mitigations, improvements to GitHub's (and other) SBOM generators, and best practices to minimize errors in dependency metadata.

5.1.2 *Study Context and Planning*

The context of this study consists of open-source software systems hosted on GitHub written in Java and Python, the former using Maven as a package manager, the latter employing pip or Poetry as a package manager. The focus was on the Java and Python programming languages because of their popularity—according to PYPL index, Python is the most popular programming language, followed by Java [134]. The choice of the package managers is driven by compatibility with GitHub's dependency graph. Maven is the only package manager supported for Java, while pip and Poetry are the only ones supported for Python [76].

GitHub Search (GHS) [43] was leveraged to search for software systems on GitHub. This tool facilitates the selection of repositories hosted on GitHub. While doing so, the perils of mining GitHub identified by Kalliamvakou *et al.* [93] were taken into consideration. Therefore, using GHS, we searched for repositories that met the following inclusion criteria:

1. *Developed in Java or Python.* This was to focus on these two popular programming languages [134].

2. *Not archived and with at least one commit made in the month preceding the query date (April 2023).* This was to avoid selecting software systems that are inactive [93].

3. *With at least 100 stars, at least 100 commits, and at least one fork.* This was to mitigate the risk of selecting personal repositories [93].

4. *Not fork.* This was to limit the risk of selecting duplicate software systems [93].

The search with GHS returned 7,172 software repositories (2,296 Java repositories and 4,876 Python repositories) satisfying Criteria 1-4. Later, an additional inclusion criterion that GHS did not support was applied. Specifically, we ensured that the considered software repositories:

5. *Used Maven, pip, or Poetry as package managers.* This ensured that the included systems used package managers compatible with GitHub's dependency graph.

To check whether a software repository met Criterion 5, we detected the use of the package manager (*i.e.,* Maven, pip, and Poetry) by looking for the corresponding manifest/lock files in the repository. More specifically, the use of Maven in Java systems was detected by checking—using the PyGitHub library [144]—for the presence of `pom.xml` files. Regarding Python systems, the use of pip was detected by verifying the presence of `requirements.txt`, `pipfile`, or `pipfile.lock` files, while the use of Poetry was detected from the presence of `poetry.lock` files. It should be noted that pip and Poetry can leverage other manifest/lock file formats (*e.g.,* `setup.py` and `pyproject.toml`), different from those mentioned above. However, their use is not recommended with GitHub's dependency graph [76]. For this reason, the focus was placed only on `requirements.txt`, `pipfile`, `pipfile.lock`, and `poetry.lock` files for Python systems. We retrieved a total of 1,330 Java systems using Maven and 2,948 Python systems using pip/Poetry, representing the populations of interest. Due to the large size of these populations,

two statistically significant random samples were built with a confidence level equal to 95% and a margin of error equal to 5%. As for the population of Java systems using Maven, the required sample size resulted in 298. The sample size required for the Python systems using pip/Poetry was equal to 340. When sampling software repositories from the populations of interest, each repository was manually checked (by reading the repository's description/README) to determine whether or not it was:

6. *Intended for software development.* This avoided the selection of books, tutorials, or students' assignments [93].

If a repository did not meet Criterion 6 (or the repository's description/README was not in English), it was discarded, and a new candidate system was randomly selected for inclusion in the sample, with the criterion reapplied. The check of Criterion 6 was postponed to the time of sampling to avoid manually verifying each repository in the populations of interest. This design choice does not negatively affect the statistical significance of the samples. This is because if a repository were not for software development, it should also be removed from the population of interest.

Finally, we retrieved and then analyzed both dependencies and dependents of each selected project by leveraging GitHub's dependency graph at two different times. As mentioned above, we considered two snapshots: the first snapshot was taken in April 2023, while the second snapshot was taken in September 2024. To retrieve both dependencies and dependents, given the lack of GitHub-specific APIs for retrieving information from the dependency graph, we parsed the dependency graph pages of each selected system using a web scraping library, *i.e.,* BeautifulSoup [148]. It should be noted that, while retrieving and analyzing dependencies and dependents in the first snapshot, we found three systems (one developed in Java and two developed in Python) to be no longer available after sampling. Consequently, these three systems were discarded. Similarly, one Java system was no longer available in the second snapshot. These systems were not replaced in the random samples (in either the first or second snapshot) because, after recomputing the required sizes of the random samples considering the systems discarded for

Table 5.1: Descriptive statistics of the studied Java and Python systems.

Snapshot	Language	Metric	Sum	Mean	SD	Min	Median	Max
First	Java	# Commits	1,070,792	3,605.36	10,448.09	100	1,262	134,597
		# Contributors	11,822	39.8	40.47	1	27	306
		# Stars	260,554	877.29	1,509.73	100	331	11,879
		# Forks	83,520	281.21	474.5	8	127	4,121
	Python	# Commits	460,759	1,363.19	2,392.21	101	662	21,259
		# Contributors	11,354	33.59	49.3	1	17	402
		# Stars	382,797	1,132.54	2,639.82	100	339	25,777
		# Forks	66,757	197.51	448.52	4	76	4,461
Second	Java	# Commits	1,052,994	3,557.41	10,433.00	100	1,259	134,597
		# Contributors	11,681	39.46	40.1	1	27	306
		# Stars	260,526	879.82	1,511.66	100	332	11,879
		# Forks	83,338	281.72	475.23	8	126	4,121
	Python	# Commits	460,759	1,363.19	2,392.21	101	662	21,259
		# Contributors	11,354	33.59	49.3	1	17	402
		# Stars	382,797	1,132.54	2,639.82	100	339	25,777
		# Forks	66,757	197.51	448.52	4	76	4,461

not meeting Criterion 6, the samples remained large enough. Also, we wanted to keep the set of systems fixed along the two snapshots. That is, our random samples were still statistically significant with a confidence level of 95% and an error margin of 5%. In summary, we ended up with 297 Java and 338 Python projects in the first snapshot and 296 Java and 338 Python systems in the second snapshot (*i.e.*, the same systems as the first snapshot except for one).

In Table 5.1, we report some descriptive statistics—*i.e.*, sum, mean, SD, min, median, and max—on the number of commits, contributors, stars, and forks of the systems studied in the first and second snapshots. Regardless of the snapshot, the number of commits, contributors, stars, and forks summarized in Table 5.1 were extracted at the time of the query to search software systems on GitHub (April 2023). The only difference between the studied systems in the first snapshot and the second snapshot concerns one Java system, which was not available anymore at the time of the second snapshot. Therefore, while the descriptive statistic

Table 5.2: Descriptive statistics for the analyzed dependencies.

Snapshot	Language	% Repositories with Dependencies	# Dependencies*					
			Sum	Mean	SD	Min	Median	Max
First	Java	100% (297/297)	60,803	204.72	425.36	1	57	3,027
	Python	100% (338/338)	14,901	44.09	85.58	1	16	1,158
Second	Java	99% (292/296)	63,981	219.11	453.98	1	59	3,418
	Python	97% (327/338)	15,762	48.20	89.55	1	18	1,133

* Excluding repositories without dependencies.

Table 5.3: Descriptive statistics for the analyzed dependents.

Snapshot	Language	% Repositories with Dependents	# Dependents*					
			Sum	Mean	SD	Min	Median	Max
First	Java	63% (188/297)	1,761,072	9,367.40	33,147.31	1	342	279,714
	Python	57% (192/338)	307,369	1,600.88	8,167	1	48	97,074
Second	Java	67% (197/296)	2,312,381	11,737.97	44,878.61	1	448	360,411
	Python	64% (217/338)	533,284	2,457.53	9,526.35	1	67	65,618

* Excluding repositories without dependents.

values shown in Table 5.1 are the same in the first snapshot and the second snapshot for Python systems, there are some slight differences in these values between the two snapshots when considering Java systems.

Table 5.2 and Table 5.3 summarize the number of dependencies and dependents returned by GitHub's dependency graph, which were analyzed in the first snapshot and the second snapshot, respectively. The number of dependencies and dependents varies at different times, *i.e.*, April 2023 for the first snapshot and September 2024 for the second snapshot. For this reason, the same system may have a different number of dependencies or dependents when passing from the first snapshot to the second snapshot. As shown in Table 5.2, in the first snapshot, 60,803 dependencies in Java systems and 14,901 in Python systems were analyzed. All systems included at least one dependency. As for the second snapshot, the analysis covered 63,981 dependencies in Java systems and 15,762 in Python systems. The majority of systems had at least one dependency, with 99% Java systems and

97% Python systems meeting this criterion. As shown in Table 5.3, in the first snapshot, 63% of Java systems and 57% of Python systems had at least one dependent. For the second snapshot, the analysis revealed that 66% of Java systems and 64% of Python systems had at least one dependent. Excluding systems without dependents, the average number of dependents per Java and Python system was 11,737.97 and 2,457.53, respectively. In total, 2,312,381 dependents for Java systems and 533,284 dependents for Python systems were analyzed.

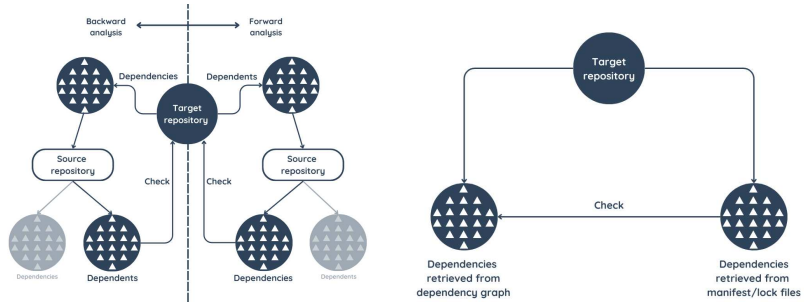
5.2 RESULTS

This section presents, for each RQ, the analysis procedure and the results obtained from the backward, forward, and manifest/lock file analyses.

5.2.1 *RQ1. To What Extent Does GitHub's Dependency Graph Provide Accurate Information?*

Before detailing the analysis procedure used to answer RQ1, the terminology used in the remainder of the chapter is introduced. Specifically, *target repository* refers to the repository that is the object of analysis. In contrast, *source repository* denotes any repository that either depends on or is dependent on a target repository.

To study RQ1, three distinct analyses were conducted through Python scripts to assess the accuracy of the dependencies and dependents reported in GitHub's dependency graph: backward analysis, forward analysis, and manifest/lock file analysis. Backward analysis refers to the process aimed at analyzing dependencies, while forward analysis refers to the process aimed at analyzing dependents. The manifest/lock file analysis aims at analyzing the package manager files stored in the target repository. These three analyses are designed to evaluate different aspects of the accuracy of GitHub's dependency graph, ensuring a comprehensive assessment thereof. A description of these analyses follows:



(a) Backward and forward analysis. (b) Manifest/lock file analysis.

Figure 5.1: Graphical representations of the three analyses performed on the GitHub's dependency graph.

- *Backward analysis.* This analysis evaluates the accuracy of the dependencies listed in GitHub's dependency graph. The backward analysis is graphically depicted on the left side of Figure 5.1a. First, a target repository (*i.e.*, one of the sampled systems) was selected and its dependencies, as reported in GitHub's dependency graph, were examined. Next, for each dependency, the corresponding source repository was accessed and verified to determine whether the target repository was listed among its dependents.

- *Forward analysis.* It focuses on the accuracy of the dependents reported in GitHub's dependency graph. The right-hand side of Figure 5.1a provides a graphical representation of the forward analysis. Starting with a target repository (*i.e.*, one of the sampled systems), its dependents were first identified. Then, for each dependent, the corresponding source repository was accessed and checked to determine whether its listed dependencies included the target repository.

- *Manifest/lock file analysis.* This analysis evaluates how accurately GitHub's dependency graph reflects the dependency information contained in manifest/lock files. Figure 5.1b shows a graphical representation of the process used to analyze the manifest/lock files. For each target repository, its manifest/lock files were retrieved at the time they were analyzed by GitHub's dependency graph to ensure the anal-

Table 5.4: Backward analysis results of Java and Python dependencies.

Snapshot	Detection	Java		Python	
		Sum	%	Sum	%
First	Match	45,707	75.17	12,589	84.48
	Mismatch	15,096	24.83	2,312	15.52
	Total	*60,803*	-	*14,901*	-
Second	Match	46,866	73.25	12,272	77.86
	Mismatch	17,115	26.75	3,490	22.14
	Total	*63,981*	-	*15,762*	-

ysis of the same file versions. This was possible because, for each dependency, GitHub's dependency graph provides the dependency detection date. Therefore, the manifest/lock files were checked out on that date and parsed to extract the list of dependencies contained. Finally, a script was used to verify whether the dependencies retrieved from the manifest/lock files matched those in the dependency graph for that target repository. The match took into account (i) repository name, (ii) dependency name, (iii) path to the manifest/lock file, and (iv) dependency version. If a dependency in GitHub's dependency graph was not found in the manifest/lock file (or vice versa), the dependency was labeled as *match not found*, otherwise as *match found*.

5.2.1.1 *Backward Analysis*

The results of the backward analysis for Java and Python systems are presented in Table 5.4, for both the first snapshot (April 2023) and second snapshot (September 2024). As shown in Table 5.4, in the second snapshot, a total of 79,743 dependencies were examined, with the majority originating from Java systems (63,981) and a smaller portion from Python systems (15,762). In terms of accuracy, 73.25% of Java dependencies were correctly identified, indicating a match between the dependencies being analyzed and the dependents of those dependencies. On the other hand, the dependency graph produced incorrect information in 26.75% of

Table 5.5: Occurrences of error cases for Java and Python dependencies.

Snapshot	Error Case	Java		Python	
		Sum	%	Sum	%
First	Occurrence not found	687	4.55	1,612	69.74
	Package not found	1,945	12.89	165	7.14
	Source repository not found	53	0.35	30	1.30
	No link to source repository	12,411	82.21	505	21.84
Second	Occurrence not found	2,557	14.94	2,533	72.58
	Package not found	3,522	20.59	541	15.50
	Source repository not found	46	0.27	14	0.40
	No link to source repository	10,990	64.20	402	11.52

cases (mismatch). The accuracy for Python systems was greater than for Java, with 77.86% dependencies correctly detected.

As for the error cases for the incorrect detection of dependencies, the following ones were identified:

- *Occurrence not found.* It happens when the dependency of a target repository does not list the target repository among its dependents.
- *Package not found.* It happens when GitHub's dependency graph divides a target repository into multiple packages, and the dependency of a package does not have that package listed among its dependents.
- *Source repository not found.* It happens when the dependency of a target repository is linked to a source repository that does not exist.
- *No link to source repository.* It happens when the dependency of a target repository lacks a connection to any source repository.

Table 5.5 shows the occurrences of the identified error cases across the analyzed systems for both snapshots. In the second snapshot, the most relevant errors for Java repositories were related to *no link to source repository*, which accounted for over 64% of all detected issues. This suggests a systemic problem in how GitHub's dependency graph resolves and associates source

Table 5.6: Repositories, excluding those without dependencies, affected by errors during the backward analysis.

Snapshot	Repositories	Java		Python	
		Sum	%	Sum	%
First	With errors	269	90.57	247	73.08
	Without errors	28	9.43	91	26.92
	Total	*297*	-	*338*	-
Second	With errors	267	91.44	260	79.51
	Without errors	25	8.56	67	20.49
	Total	*292*	-	*327*	-

repositories for Java packages. Instead, Python systems showed a different error distribution. The most frequent error was *occurrence not found*, representing nearly three-quarters (about 73%) of all Python-related errors. This reflects possible weaknesses in how GitHub tracks backward links from dependencies in the Python ecosystem, perhaps owing to its more dynamic dependency structures.

Finally, the number of repositories (excluding those without any dependency) in which at least one of the error cases listed above was identified during the backward analysis was analyzed. The results of this analysis are shown in Table 5.6 for both snapshots. In the second snapshot, about 91% of Java repositories and 80% of Python repositories incurred at least one incorrect dependency. This means that the dependency graph provided accurate information for all dependencies in only 9% of Java repositories and 20% of Python ones.

LONGITUDINAL ANALYSIS Compared to the second snapshot, a lower number of dependencies (75,704) were examined in the first snapshot (see Table 5.4). However, similar to the second snapshot, the majority of dependencies originated from Java systems (60,803) and a smaller portion from Python systems (14,901). Moreover, the accuracy of the dependency graph was higher for Python systems (84.48% of dependencies correctly identified) than for Java systems (75.17% of dependencies correctly identi-

fied). To determine whether the proportions of match/mismatch dependencies significantly changed between the first and second snapshots, we used Fisher's exact test.[1] This test returned p-values equal to 8.97×10^{-15} and 2.2×10^{-16} for Java and Python systems, respectively, indicating the presence of statistically significant differences in the proportions of match/mismatch dependencies between the two snapshots for both Java and Python systems. As Table 5.4 suggests, these statistically significant differences are due to a worsening of the accuracy of the dependency graph, in terms of dependencies, between the first and second snapshots. Finally, as shown in Table 5.6, in the first snapshot, 90.57% of Java and 73.08% of Python repositories had errors in their dependencies. Consequently, GitHub's dependency graph provided accurate dependency information for only 9.43% of Java and 26.92% of Python repositories. To assess whether the proportions of repositories affected or not by errors in their dependencies significantly changed between the first and second snapshots, Fisher's exact test returned p-values of 0.774 for Java systems and 0.056 for Python systems. Therefore, we cannot conclude that these proportions changed significantly between the two snapshots.

5.2.1.2 *Forward Analysis*

The results of the forward analysis for Java and Python systems are presented in Table 5.7, for both the first snapshot and the second snapshot. As shown in Table 5.7, in the second snapshot, a total of 2,845,665 dependents were examined, with the majority originating from Java systems (2,312,381) and a smaller portion from Python systems (533,284). In terms of accuracy, 98% of Java dependents were correctly identified, indicating a match between the dependents being analyzed and the dependencies of those dependents. Therefore, the dependency graph provided inaccurate information in 2% of Java dependents. Regarding Python systems, the accuracy was lower, with only 89.54% of dependents being correctly detected.

1 For any statistical test run, the significance level, α, was fixed at 0.05. This means a difference was considered statistically significant if the p-value returned by the test was less than or equal to α.

Table 5.7: Forward analysis results of dependents.

Snapshot	Detection	Java		Python	
		Sum	%	Sum	%
First	Match	1,701,626	96.62	218,044	70.94
	Mismatch	59,446	3.38	89,325	29.06
	Total	*1,761,072*	-	*307,369*	-
Second	Match	2,266,158	98	477,513	89.54
	Mismatch	46,223	2	55,771	10.46
	Total	*2,312,381*	-	*533,284*	-

As for the error cases for the incorrect detection of dependents, the following cases were identified:

- *Occurrence not found.* It happens when the dependent of a target repository does not have that target repository among its dependencies.
- *Source repository not found.* It happens when the dependent of a target repository is linked to a source repository that does not exist.
- *Repository not available.* It happens when the dependent of a target repository is unavailable due to policy violations (*e.g.*, violations of GitHub's Terms of Service or takedowns under the Digital Millennium Copyright Act).

As done for the backward analysis, Table 5.8 reports how often the error cases for the incorrect detection of dependents occurred. In the second snapshot, the majority of errors in Java and Python systems are related to *occurrence not found*, where the expected dependent repository does not appear in GitHub's dependency graph. This suggests a systematic shortcoming in the way GitHub reconstructs the reverse dependencies. Looking at the less frequent error cases, such as *source repository not found* and *repository not available*, it seems that the accuracy of GitHub's dependency graph is also affected by temporal and structural factors, such as repository renaming, deletion, or inaccessibility over time. These issues highlight the inherent fragility of dependency tracking in dynamic and evolving software.

Table 5.8: Occurrences of error cases for Java and Python dependents.

Snapshot	Case	Java		Python	
		Sum	%	Sum	%
First	Occurrence not found	31,885	53.64	86,292	96.60
	Source repository not found	27,410	46.11	3,028	3.39
	Repository not available	151	0.25	5	0.01
Second	Occurrence not found	41,667	90.14	54,978	98.58
	Source repository not found	4,016	8.69	790	1.42
	Repository not available	540	1.17	3	< 0.01

Table 5.9: Repositories, excluding those without dependents, affected by errors during the forward analysis.

Snapshot	Repositories	Java		Python	
		Sum	%	Sum	%
First	With error	163	86.7	150	78.13
	Without errors	25	13.3	42	21.88
	Total	*188*	-	*192*	-
Second	With error	172	87.3	215	99.08
	Without errors	25	12.7	2	0.92
	Total	*197*	-	*217*	-

Table 5.9 reports the number of repositories, excluding those without any dependents, in which at least one of the error cases listed above was detected during the forward analysis. In the second snapshot, about 88% of Java repositories and 99% of Python repositories incurred at least one incorrect dependent. This means that the dependency graph provided accurate information for all dependents in only 12% of Java repositories and 1% of Python repositories.

LONGITUDINAL ANALYSIS As shown in Table 5.7, in the first snapshot, a total of 2,068,441 dependents were examined (*i.e.*, a lower number compared to the second snapshot). Similar to the second snapshot, the majority of dependents originated from Java systems (1,701,626) and a smaller portion from Python sys-

Table 5.10: Number of dependencies retrieved from manifest/lock files and dependency graph.

Snapshot	Source	# Dependencies	
		Java	Python
First	Manifest/lock files	68,004	14,858
	Dependency graph	60,803	14,901
Second	Manifest/lock files	75,537	19,569
	Dependency graph	63,981	15,762

tems (307,369). Moreover, the accuracy of the dependency graph, in terms of dependents, was higher for Java systems (96.62% of dependents correctly identified) than for Python systems (70.94% of dependents correctly identified). Fisher's exact test was also run to determine whether the proportions of match/mismatch dependents significantly changed between the first and second snapshots. The test returned a p-value of 2.2×10^{-16}, for Java systems, and 2.2×10^{-16}, for Python ones, indicating statistically significant differences in the proportions of match/mismatch dependents between the two snapshots for both Java and Python systems. By looking at Table 5.7, it is easy to grasp that the accuracy of the dependency graph, in terms of dependents, improved between the two snapshots. Finally, as shown in Table 5.9, in the first snapshot, mismatch dependents affected 86.7% of Java and 78.13% of Python repositories. Fisher's exact test did not indicate statistically significant differences for Java (p-value of 0.881), while it was the case for Python (p-value of 4.59×10^{-13}).

5.2.1.3 *Manifest/lock File Analysis*

Table 5.10 shows the number of dependencies retrieved during the analysis of manifest/lock files, compared with dependencies returned by the dependency graph, for Java and Python systems (in both the first and second snapshots). During the analysis of manifest/lock files, six manifest/lock files—four related to Java systems and two related to Python systems—could not be retrieved from the corresponding commits. This occurred in both the first and second snapshots for the same manifest/lock files.

Table 5.11: Comparison between the number of dependencies retrieved from manifest/lock files and those retrieved from the dependency graph in the analyzed repositories (excluding repositories without dependencies and/or those for which manifest/lock files could not be retrieved).

Snapshot	Case	Java		Python	
		Sum	%	Sum	%
First	Lower	21	7.07	19	5.62
	Equal	84	28.28	284	84.02
	Greater	192	64.65	35	10.36
	Total	*297*	-	*338*	-
Second	Lower	11	3.79	19	5.83
	Equal	59	20.34	255	78.22
	Greater	220	75.86	52	15.95
	Total	*290*	-	*326*	-

As shown in Table 5.10, 75,537 Java and 19,569 Python dependencies were identified in manifest/lock files in the second snapshot. We analyzed the extent to which, across the studied systems, the number of dependencies retrieved from manifest/lock files is smaller, equal to, or greater than the number retrieved from GitHub's dependency graph. The results of this analysis, for both snapshots, are reported in Table 5.11. In the second snapshot, it was found that, for the vast majority of Java repositories (about 76%), the manifest/lock files revealed more dependencies than those captured by GitHub's dependency graph. This suggests that GitHub's extraction mechanisms may be systematically under-reporting dependencies in Java, likely due to challenges in parsing Maven's hierarchical and variable-driven configuration structures. Conversely, the picture is different for Python, where, for the vast majority of repositories (about 78%), the number of dependencies matched exactly between the two sources, indicating a stronger alignment between the manifest/lock files and GitHub's dependency graph. Discrepancies were relatively limited. These findings reinforce the view that GitHub dependency extraction performs more reliably in Python systems, where dependency declarations are typically simpler and more central-

Table 5.12: Match between the dependencies retrieved from the manifest/lock files and those from the dependency graph.

Snapshot	Match	Java		Python	
		Sum	%	Sum	%
First	Found	52,490	77.19	14,220	95.71
	Not found	15,514	22.81	638	4.29
	Total	*68,004*	-	*14,858*	-
Second	Found	57,691	76.37	13,114	67.01
	Not found	17,846	23.63	6,455	32.99
	Total	*75,537*	-	*19,569*	-

ized, while Java's more complex configuration introduces higher chances of omission or misinterpretation. Moreover, for both Java and Python systems, in light of previous analyses, it is possible to better interpret cases where fewer dependencies are found in the manifest/lock files than in GitHub's dependency graph. Specifically, this is likely due to situations where dependencies were removed from the system's manifest/lock files but are still present in GitHub's dependency graph due to outdated indexing or delayed updates. This reinforces the idea that GitHub's dependency graph does not always promptly reflect changes in a system's configuration and may retain obsolete dependencies that no longer exist in the actual codebase. Such stale data can distort the representation of a system's current dependency structure and must be considered when using the dependency graph for research or analysis purposes.

Finally, to assess the extent to which the information from manifest/lock files is consistent with that retrieved from the dependency graph, Table 5.12 reports the number and percentage of dependencies retrieved from manifest/lock files that match those retrieved from the dependency graph. In the second snapshot, the dependencies of Java repositories retrieved from manifest/lock files matched those from GitHub's dependency graph in about 76% of cases. This percentage drops to 67% for the dependencies of Python repositories.

LONGITUDINAL ANALYSIS As Table 5.10 shows, 68,004 Java and 14,858 Python dependencies were identified in manifest/lock files in the first snapshot. In comparison with the dependencies extracted from the dependency graph, Java repositories showed 7,201 additional dependencies declared in manifest/lock files, while Python repositories had 43 fewer dependencies declared in manifest/lock files. Moreover, in the first snapshot, the number of dependencies retrieved from manifest/lock files was greater, equal, and lower than those from GitHub's dependency graph in 64.65%, 28.28%, and 7.07% of Java repositories, respectively (see Table 5.11). As for Python repositories, the number of dependencies retrieved from manifest/lock files was greater, equal, and lower than those from GitHub's dependency graph in 10.36%, 84.02%, and 5.62% of cases, respectively. Finally, in the second snapshot, the dependencies of Java repositories retrieved from manifest/lock files matched those from GitHub's dependency graph in 77.19% of cases (see Table 5.12). This percentage increases to 95.71% for the dependencies of Python repositories. To determine whether the proportions of the matched/unmatched manifest/lock files changed significantly between the first snapshot and the second snapshot, Fisher's exact test was leveraged. For Java, the test returned a p-value equal to 3×10^{-04}, indicating a statistically significant difference in the proportions of matched/unmatched manifest/lock files between the two snapshots. For Python, the test returned a p-value 2.21×10^{-16}, suggesting again a statistically significant difference in the proportions. As Table 5.12 suggests, these differences are due to a substantial drop in the match between the dependencies declared in manifest/lock files and those retrieved from the dependency graph when passing from the first snapshot to the second snapshot for both Java and Python systems.

Answer to RQ1

GitHub's dependency graph reports inaccurate information, in either the dependencies or dependents, for the majority of the systems analyzed. This derives from up to $\sim$ 25% in the first snapshot and $\sim$ 27% in the second snapshot of incorrect dependencies, and $\sim$ 29% in the first snapshot and $\sim$ 10% in the second snapshot of incorrect dependents. The incorrect dependencies occur with different error types, related to dependency occurrences, packages, source repositories not found, or wrong links are evident from a mismatch between GitHub's dependency graph and the manifest/lock files. Moreover, the dependency information from manifest/lock files and GitHub's dependency graph does not always match: 77.19% in the first snapshot and 76.37% in the second snapshot of Java dependencies have a match, while Python dependencies have a match in 95.71% in the first snapshot and 67.01% in the second snapshot of cases. For Java, the percentage of errors does not consistently decrease over the two observed snapshots, but it does for Python. Anyway, imprecise analyses persist in the dependency graph computation.

5.2.2 *RQ2. What Are the Likely Causes for the Error Cases Occurring in GitHub's Dependency Graph?*

To answer RQ2, occurrences of the error cases identified in the previous analyses—*i.e.*, backward, forward, and manifest/lock file analyses—were analyzed to determine the likely root causes behind them. Given that the total set of errors was too large for exhaustive manual analysis, statistically significant samples of the error case occurrences were built for each of the analyses, with a confidence level of 95% and a margin of error of 5%. To ensure balanced representation of the different error cases across Java and Python systems, stratified sampling was performed, consisting of two strata: (i) the programming language and (ii) the error case.

Table 5.13: Sampling of Error Case Occurrences.

Analysis	Error Case	Java	Python	Total
Backward	Occurrence not found	103	102	205
	Package not found	142	22	164
	Total	*245*	*124*	*369*
Forward	Occurrence not found	156	206	362
	Source repository not found	15	10	25
	Repository not available	10	3	13
	Total	*181*	*219*	*400*
Manifest/lock file	Not found	278	101	379

Below, the analysis procedure for the error cases that occurred in the backward, forward, and manifest/lock file analyses is described, respectively:

- *Backward analysis.* When sampling the error cases that occurred in the backward analysis, the *source repository not found* and *no link to source repository* error cases were excluded because, due to their intrinsic nature (missing link to the repository to be analyzed), the root causes behind them could not be deeply investigated. Starting from a population of 9,153 errors (*i.e.*, 5,090 errors related to *occurrence not found* plus 4,063 errors related to *package not found* across Java and Python systems, see Table 5.5), a statistically significant random sample was built(with a confidence level and a margin of error equal to 95% and 5%, respectively). This sample included 369 error case occurrences: 205 errors were related to *occurrence not found* and 164 errors were related to *package not found*, as shown in Table 5.13. To analyze the *occurrence not found* error case, three pairs of authors (hereafter referred to as annotators) manually assessed the correctness of the reported dependencies, following a structured process:
  1. Access the list of dependents for each source repository using the provided GitHub link.
  2. Determine the correct package:

- If multiple packages are available, select (if any) the correct package corresponding to the dependency under analysis from the drop-down menu (see Figure 5.2).

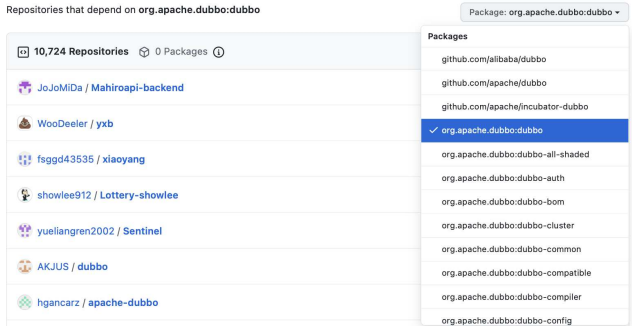


Figure 5.2: Example of package drop-down menu of GitHub's dependency graph.

- If only one package is available, check whether the package name reported in "Repositories that depend on <package name>", corresponds to the dependency under analysis (see the top left corner of Figure 5.3).



Figure 5.3: Example of a single package of GitHub's dependency graph.

3. Determine the likely root cause(s), and whether the target repository is not listed as a dependent.
   - If the target repository name appears to be changed or different from the expected one, take note of the discrepancy and then search for the correct target repository name within the list of dependents reported by GitHub.
   - If the target repository is not found, record a negative match.
4. Access the source code of the target repository and inspect its manifest/lock files to understand whether the

dependency is explicitly declared in the repository's manifest/lock files.

5. If the dependency is found in the manifest/lock files but is missing from GitHub's dependency graph, take note of this.

Throughout the process, observations were recorded by the annotators on a dedicated evaluation sheet. After completing the annotation, a third researcher manually analyzed the notes from each annotator to extract meaningful error labels. In cases of disagreement between the two annotators, the third researcher conducted a thorough investigation of the occurrence to determine the final label. The agreement between the pairs of annotators was measured using Krippendorff's alpha, resulting in a value of $\alpha = 0.862$, indicating a high level of reliability and strong agreement. Regarding the *package not found* error case, it was examined whether these occurrences could be attributed to GitHub's processing limits [78]. Given that GitHub imposes constraints on dependency resolution, some errors might arise from these limitations rather than inaccuracies in the dependency graph itself. To assess this, an automated script included in the replication package [22] was used to analyze the identified occurrences and determine if they aligned with known processing restrictions. Errors linked to these constraints were categorized accordingly.

- *Forward analysis.* Given the 101,994 errors that occurred in the forward analysis (46,223 coming from Java systems and 55,771 from Python ones, as shown in Table 5.8), the resulting significant sample consisted of 400 errors, including 362, 25, 13 errors belonging to *occurrence not found*, *source repository not found*, and *not available*, respectively (see Table 5.13). To analyze the errors identified during the forward analysis, an automated inspection of the version history of manifest/lock files was performed using a script included in the replication package [22]. For each dependent of a target repository, the corresponding source repository was accessed to check whether the dependent under analysis

was ever present and whether it was later removed from the source repository's dependencies.

- *Manifest/lock file analysis*. Starting from the 24,301 errors that occurred in the manifest/lock file analysis (17,846 resulting from Java systems and 6,455 resulting from Python ones, see Table 5.12), the built significant sample consisted of 379 occurrences, as detailed in Table 5.13. The goal of this analysis is to assess the likely root causes of the errors encountered when there was a mismatch between the dependencies recorded in the manifest/lock files and those reported in GitHub's dependency graph. To conduct the analysis, three pairs of authors (*i.e.*, annotators) examined whether the dependencies recorded in the manifest/lock files contained any inconsistencies, such as missing information or the use of variables (*e.g.*, `${james.groupId}:james-server-data-memory`). If an inconsistency was found, the annotators recorded it. Next, the annotators accessed the commit history of the repository and selected the option to view the code at the specific commit for which GitHub's dependency graph was analyzed. This step enabled the retrieval of the exact version of the manifest/lock file that was used in GitHub's dependency detection process. The manifest/lock file's content was then manually examined to verify the presence of the dependency name and to identify potential causes for mismatches between the dependency graph and the actual file content. For the manifest/lock file analysis, the annotators recorded their evaluations in a dedicated sheet. At the end of the annotation process, a third researcher reviewed the annotations and accompanying notes to extract consistent and meaningful labels for each case. In the presence of disagreements, the third researcher conducted a detailed inspection to determine the final classification. The reliability of the manual labeling process was assessed using Krippendorff's alpha. The resulting value was $\alpha = 0.973$, indicating an excellent level of agreement between the annotators and a highly reliable annotation process.

5.2.2.1 *Backward Analysis*

The following likely root causes for errors related to *occurrence not found* were identified:

- *Dependent not yet indexed.* The dependent repository is not listed in GitHub's dependency graph at the time of analysis, likely due to indexing delays or incomplete updates.
- *Name of the repository changed.* The dependent repository name has been modified since GitHub last indexed it, leading to a mismatch between the recorded dependent and the actual system.
- *Owner of the repository changed.* The dependent repository had been transferred to a different owner, which affects how dependencies are tracked and displayed in GitHub's dependency graph.
- *Package name differs from the actual one.* The package name of the dependent repository reported in GitHub's dependency graph did not match the actual name used in the dependent repository.
- *Manifest/lock file not found.* The manifest/lock file where the dependent repository declares the dependency (identified by the dependency graph) has not been found.
- *Archived repository.* The dependent repository has been archived and is now read-only. Consequently, the dependency graph is not enabled anymore for that repository.
- *Inconsistency (hyphen "−" vs. underscore "_").* The package name of the dependent's repository reported in GitHub's dependency graph did not match the actual name of the repository due to misuse of hyphens/underscores. For example, the package name reported in GitHub's dependency graph is "parse-type", but the actual name of the package is "parse_type".
- *Dependent not in manifest/lock files.* The dependent is not found in the manifest/lock files.

Table 5.14: Likely causes for the *occurrence not found* error case encountered in the backward analysis.

Likely Cause	Java		Python	
	Sum	%	Sum	%
Dependent not yet indexed	15	13.39	18	13.95
Name of the repository changed	20	17.86	6	4.65
Owner of the repository changed	18	16.07	34	26.36
Package name differs from the actual one	24	21.43	46	35.66
Manifest/lock file not found	25	22.32	1	0.78
Archived repository	1	0.89	2	1.55
Inconsistency (hyphen "–" vs. underscore "_")	0	0	14	10.85
Dependent not in manifest/lock files	3	2.68	7	5.43
Unknown	6	5.36	1	0.78
Total	*112*	-	*129*	-

- *Unknown.* When the likely reason for an error could not be clearly identified based on the available data.

Table 5.14 shows how often each of the above-mentioned likely root causes was identified. For Java repositories, the most likely error cause was *manifest/lock file not found*, accounting for 22.32% of the cases. Notably, the most frequent cause among Java repositories was the absence of a manifest or lock file, suggesting that GitHub's dependency graph may have limitations in correctly locating or indexing these files for Java systems. Python systems were more often affected by inconsistencies in package naming. The *package name differs from the actual one* issue alone accounted for more than a third (35.66%) of the observed Python errors, highlighting the sensitivity of the graph builder to naming conventions and metadata accuracy. It was also observed that repository ownership or name changes contributed significantly to errors, particularly in Python. These suggest that GitHub's indexing and tracking mechanisms can be brittle in the face of system evolution, such as renaming or transferring ownership, which can affect the reliability of dependency resolution over time. Overall, the analysis indicates that error causes are ecosystem-specific and that the underlying mechanisms of the dependency graph may be more prone to failure under certain

Table 5.15: Likely causes for the *package not found* error case encountered in the backward analysis.

Likely Cause	Java		Python	
	Sum	%	Sum	%
Exceeds max number of manifest/lock files (150)	106	74.65	0	0.00
No processing limit violated	36	25.35	22	100.00
Total	*142*	-	*22*	-

structural or metadata-related conditions, particularly in dynamically evolving systems.

As far as the *package not found* error case, two likely causes were identified:

- *Exceeds max number of manifest/lock files.* It indicates issues with packages not being found due to exceeding the maximum number of manifest/lock files allowed per GitHub repository.
- *No processing limit violated.* It indicates that no processing limit was violated by the repository.

Table 5.15 shows how often each of the above-mentioned likely root causes was identified. The most frequent issue among Java repositories was the violation of GitHub's limit on the number of manifest/lock files, which occurred in 74.65% of the cases. In contrast, all Python repositories (100%) and a minority of Java repositories (25.35%) exhibited no identifiable processing limit violation. It is worth noting that, beyond the *exceeds max number of manifest/lock files* constraint, other documented processing limits [78]—such as file size or file count thresholds—were also examined but found no instances of those limits being exceeded. Once again, this suggests that the observed issues are closely tied to package manager characteristics. In particular, Maven-based Java systems commonly define dependencies across multiple manifest files organized hierarchically, which increases the likelihood of exceeding GitHub's processing thresholds. While such hierarchical structures are technically possible in Python, they are less common in practice.

5.2.2.2 *Forward Analysis*

The following likely root causes for errors related to the forward analysis were identified:

- *Dependency in the DG, but not in the manifest/lock files.* This case occurs when a dependency is listed in GitHub's Dependency Graph (DG), but no corresponding declaration is found in the repository's manifest/lock files.
- *Dependency deleted, dependent of target repository not upgraded.* Here, the dependency was previously declared in manifest/lock files but was later removed. However, GitHub's dependency graph still shows the corresponding dependent of the target repository.
- *Dependency in manifest/lock files, but not in DG.* This refers to a dependency that is correctly listed in manifest/lock files but is missing in GitHub's dependency graph. This suggests a failure in GitHub's dependency extraction or update mechanisms.
- *DG of source repository is not enabled.* The dependency is listed in the manifest/lock files of the source repository, but GitHub's dependency graph is not enabled for that source repository.
- *The file disappears from the commit history, dependent on the target repository not being upgraded.* This case refers to situations where the manifest/lock files containing the dependency were deleted in the repository's commit history, but GitHub's dependency graph still shows the corresponding dependent of the target repository.
- *Error resolved.* A match was observed between the dependent and the dependency in GitHub's dependency graph. This is because GitHub solved the error meanwhile.

The results of the qualitative analysis for the errors detected during forward analysis are presented in Table 5.16. In Java repositories, the most frequent error cause encountered in the forward analysis was *dependency deleted, dependent of target repository not upgraded*, which accounted for over 70% of the cases. This suggests

Table 5.16: Likely causes for the errors that occurred in the forward analysis.

Likely Cause	Java		Python	
	Sum	%	Sum	%
Dependency in the DG, but not in the manifest/lock file	6	3.59	54	26.6
Dependency deleted, dependent of target repository not upgraded	120	71.85	92	45.32
Dependency in the manifest/lock files, but not in DG	29	17.37	6	2.96
DG of source repository is not enabled	2	1.2	48	23.65
The file disappears from the commit history, dependent of target repository not upgraded	3	1.8	1	0.49
Error resolved	7	4.19	2	0.99
Total	*167*	-	*203*	-

that GitHub's dependency graph may retain outdated dependency relationships even after the relevant declarations have been removed from the manifest/lock files. Another frequent issue in Java was the presence of dependencies in manifest/lock files that were not reflected in the dependency graph, indicating possible failures in GitHub's extraction or update mechanisms. Python repositories showed a similar trend, with *dependency deleted, dependent of target repository not upgraded* as the most common cause (about 45%). However, other causes were more prominent than in Java. Notably, over a quarter of the Python errors involved dependencies appearing in the dependency graph without a corresponding declaration in the manifest/lock files.

Overall, the results suggest that many errors in the forward analysis are due to failures in automatically updating the dependency graph, even though manifest/lock files were modified correctly in commits. This raises questions about the reliability of automatic dependency propagation in GitHub's dependency graph.

5.2.2.3 *Manifest/lock File Analysis*

The following likely root causes for the error that occurred in the manifest/lock files analysis were identified:

- *Missing group ID.* The dependency declaration in manifest/lock files lacked the necessary group identifier (only for Java systems using Maven).
- *Group ID defined by a variable.* Instead of a fixed value, the dependency declaration in manifest/lock files uses a variable that GitHub's dependency graph seems not to resolve correctly (only for Java systems using Maven).
- *Version defined by variable.* The manifest/lock files report a dependency declaration where the version is not a fixed value but is defined by a variable that GitHub's dependency graph seems not to resolve correctly (only for Java systems using Maven).
- *Artifact ID defined by variable.* The manifest/lock file reports a dependency declaration where the artifact ID is not a fixed value but is defined by a variable that GitHub's dependency graph seems not to resolve correctly (only for Java systems using Maven).
- *Version numbering schema issues.* Issues in how the versions are specified (*e.g.,* ranges, wildcards, or custom numbering conventions), which GitHub's dependency graph seems not to understand.
- *Manifest/lock files not found in the commit history.* The commit in which GitHub's dependency graph inferred the dependency from manifest/lock files disappeared.
- *Usage of scopes.* The dependency is declared in a scope that limits its visibility/applicability (only for Java systems using Maven). For example, the dependency is defined only for development (`dev` scope) or testing purposes (`test` scope). Such a dependency seems not to be recognized correctly by GitHub's dependency graph.
- *Not found in the manifest/lock file.* The dependency is not found in the manifest/lock file of the repository.
- *Unknown.* Cases where the likely reason for the error could not be clearly identified based on the available data.

Table 5.17: Likely causes for the errors that occurred in the manifest/lock file analysis.

Likely Cause	Java		Python	
	Sum	%	Sum	%
Missing group ID	45	13.64	0	0
Group ID defined by variable	97	29.39	0	0
Version defined by variable	100	30.3	0	0
Artifact ID defined by variable	9	2.73	0	0
Version numbering schema issue	10	3.03	90	84.91
Manifest/lock files not found in commit history	49	14.85	15	14.15
Usage of scopes	9	2.73	0	0
Not found in the manifest/lock file	9	2.73	0	0
Unknown	2	0.61	1	0.94
Total	*330*	-	*106*	-

The results of the qualitative analysis for the errors detected during manifest/lock files analysis are presented in Table 5.17. In Java repositories, the most frequent issues involved dynamic references that GitHub's dependency graph appeared unable to resolve: *version defined by variable* and *group ID defined by variable*. Specifically, dependencies defined using variables for the version or group ID were the dominant causes, accounting for nearly 60% of the observed errors. These highlight a key limitation in the dependency graph parsing logic when dealing with Maven-based Java systems, where such variable declarations are common practice. In contrast, the vast majority of errors in Python repositories (about 85%) stemmed from version specification formats that GitHub's dependency graph failed to interpret correctly.

Based on the results, it can be postulated that many errors in the manifest/lock files analysis stem from failures of the dependency graph to automatically extract information from manifest/lock files. These failures originate from GitHub's dependency graph limitations [78] in parsing the information from manifest/lock files (*e.g.*, version numbering schema) or resolving dynamic references (*e.g.*, versions, group ID, and artifact ID defined through variables).

Answer to RQ2

The observed results suggest that most errors in GitHub's dependency graph are caused by limitations in dealing with the information from GitHub metadata and manifest/lock files. That is, GitHub's dependency graph might not update its dependency information after a change involving a given repository. This change might involve GitHub metadata (*e.g.*, the name of the repository or its owner) or the manifest/lock files declaring system dependencies. The limitations of GitHub's dependency graph also involve the extraction of information from manifest/lock files (*e.g.*, parsing and resolution of dynamic references).

5.3 DISCUSSION

The accuracy of GitHub's dependency graph was assessed in Java and Python open-source software systems through a: (i) backward analysis, focusing on the accuracy of a given repository's dependencies, as reported in the dependency graph; (ii) forward analysis, focusing on the accuracy of the dependents, as reported in the dependency graph of a given repository; and (iii) manifest/lock file analysis, focusing on the correspondence between the information reported in the dependency graph and what was reported in the corresponding manifest/lock files. The analysis was repeated twice, considering a snapshot of the dependency graphs extracted in April 2023 and a snapshot extracted in September 2024.

The study provides evidence that GitHub's dependency graph presents accuracy issues, which affect both backward (dependency) and forward (dependent) analyses. The analyses reported at least one inaccuracy in the majority of the analyzed systems. While some improvements are observed in the accuracy of dependents between the two snapshots—possibly due to improved indexing or metadata refinement—the overall accuracy of dependencies has worsened over time. The consistency between GitHub's dependency graph and manifest/lock files also declined, particularly for Python systems, where the match rate dropped sharply. These results could be due to a lack of ro-

bustness in GitHub's dependency extraction mechanisms and also highlight dependency graph weaknesses to repository-level changes. The quantitative analysis has been complemented with a qualitative one, in which the root causes of the observed dependency graph errors were categorized. Such root causes point out limitations in how the dependency graph is created, particularly how manifest/lock files are analyzed.

Below, the implications of this study's findings are presented.

- The inaccuracy of GitHub's dependency graph can undermine the accuracy of its SBOM generator. This may lead software suppliers to distribute SBOMs containing incomplete or incorrect components. In turn, software choosers and operators may face challenges in managing the software supply chain. The inaccuracies increase the risk of undetected vulnerabilities or license compliance issues, which could ultimately result in security incidents or legal liabilities.

- The systematic categorization of the error types affecting both dependencies and dependents within GitHub's dependency graph is relevant for suppliers of SBOM generators. This categorization offers valuable insights into recurring failure modes and may directly inform efforts to enhance the accuracy and reliability of SBOM generators. Notably, GitHub has recently distributed a survey to its users, seeking feedback on the usage of its dependency graph feature to guide future improvements [74].

The implications from the empirical evaluation of GitHub's dependency graph may also be relevant to other software forges, such as GitLab, when they are used as sources of dependency information. Specifically, it is reasonable to expect that, like GitHub, platforms like GitLab could face inaccuracies in updating dependency information following changes to a given repository (*e.g.*, changes to the repository's owner or name). However, further work is needed to determine if these expectations are met.

5.3.1 *Threats To Validity*

This section discusses potential threats to the validity of the results. According to the validity schema by Wohlin *et al.* [194], these threats can be categorized as construct, external, conclusion, and internal validity. Threats to internal validity are not discussed because such threats concern causal relationships [194], and causality was not investigated in this study.

Threats to *construct validity* in this study relate to the quantitative analyses conducted—namely, the backward, forward, and manifest/lock file analyses to assess the accuracy of GitHub's dependency graph. Widely adopted libraries were employed to analyze the relevant artifacts, scripts were thoroughly tested, and the quantitative analyses were complemented with qualitative categorization of the root causes of errors. It is important to emphasize that the dependencies contained in manifest/lock files were assumed as the "ground truth," as these represent those declared by the systems' developers. This assumption serves the purpose of determining whether mismatches arise when the dependency graph is generated from these files. This does not imply that such dependencies are necessarily correct or error-free; determining the correctness or appropriateness of program dependencies is beyond the scope of this work. Another threat to construct validity involves possible mistakes and subjectivity in the qualitative analysis. To mitigate this, each instance was classified independently by two annotators, with conflict resolution applied to cases of inconsistent labeling. Additionally, labeling reliability was assessed using Krippendorff's alpha inter-rater agreement.

Regarding *conclusion validity*, selecting a subset of systems for analysis might lead to spurious or non-representative results. To mitigate this, systematic and reproducible sampling strategies were employed. Statistically significant samples of Java and Python systems hosted on GitHub were studied, following the inclusion criteria detailed in Section 5.1.2. Similarly, for RQ2, a statistically significant stratified sample—based on programming language and error type—was constructed from observed error occurrences.

A key threat to *external validity* concerns the generalizability of the results to software systems written in languages other than Java and Python on GitHub. Different programming languages have distinct ecosystem characteristics, dependency management practices, and toolchain support levels, all of which may impact the accuracy of GitHub's dependency graph. Notably, significant accuracy differences between Java and Python systems were found, highlighting the need for further research across a broader range of languages supported by GitHub's dependency graph, such as C#, JavaScript, or Ruby. Another external validity threat arises from the applied inclusion and exclusion criteria (see Section 5.1.2), which, while necessary to ensure relevance and quality, restrict the population to which the results apply. For instance, focusing on popular or active repositories may exclude stable but less active or inactive systems, potentially biasing findings. Finally, the results might not fully reflect the current accuracy of GitHub's dependency graph, as some inaccuracies may have been fixed over time. However, the analysis of two snapshots revealed a decline in accuracy between them, and qualitative findings indicated that certain elements in manifest/lock files remain challenging for GitHub's dependency graph extraction mechanisms to capture.

DATA AVAILABILITY

The replication package of the study is available online [22].

6

AIRBORNE: AUGMENTED FINE-GRAINED SOFTWARE BILL OF MATERIALS GENERATOR FOR STACK OVERFLOW

This chapter is based on the paper entitled "AIRBORNE: Augmented fIne-grained softwaRe Bill Of mateRials geNerator for stack ovErflow" [127], submitted to the 48th IEEE/ACM International Conference on Software Engineering.

State-of-practice SBOM generators operate at coarse granularity, capturing only package- or library-level dependencies while overlooking smaller artifacts such as code snippets reused from Q&A forums such as Stack Overflow. To address this gap, this chapter presents AIRBORNE (Augmented fIne-grained softwaRe Bill Of mateRials geNerator for stack ovErflow), a tool that analyzes software systems and generates an augmented fine-grained SBOM enriched with information on self-admitted Stack Overflow references. AIRBORNE integrates source code analysis, repository mining, and clone detection techniques to align Stack Overflow provenance with the CycloneDX SBOM standard. A preliminary evaluation of 383 GitHub repositories with known self-admitted Stack Overflow references indicates that AIRBORNE captures fine-grained dependencies whenever developers explicitly acknowledge snippet reuse, demonstrating its applicability in real-world development contexts.

CHAPTER STRUCTURE. Section 6.1 presents AIRBORNE. The preliminary empirical evaluation is reported in Section 6.2, while an overall discussion concludes the chapter in Section 6.3.

6.1 AIRBORNE

This section outlines the overall workflow of AIRBORNE, demonstrates its usage, and discusses its limitations.

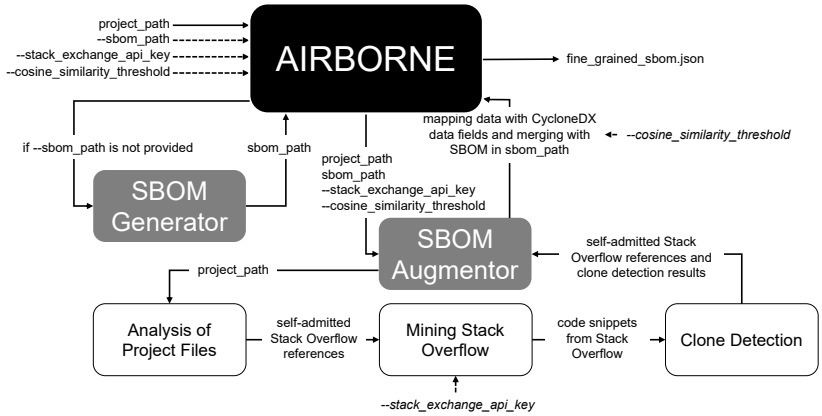


Figure 6.1: Overview of the workflow of AIRBORNE.

6.1.1 *Workflow*

The overall workflow of AIRBORNE—developed in Python 3.13.5—is illustrated in Figure 6.1. To generate the fine-grained SBOM for a given software system, AIRBORNE performs the following steps:

1. If a path to an existing CycloneDX SBOM in JSON format is not provided, AIRBORNE generates a minimal SBOM, which will be augmented with the self-admitted Stack Overflow references.

2. Each file of the software system is analyzed to detect self-admitted Stack Overflow references using the regular expression: `https?://(?:www.)?stackoverflow.com/[^\s)'"<>]+`. To be considered valid, the URLs identified through the regular expression must point to actual posts—either questions or answers—which means the URLs must contain one of the following patterns: `stackoverflow.com/questions/<id>`, `stackoverflow.com/a/<id>`, or `stackoverflow.com/q/<id>`.

3. For each detected self-admitted Stack Overflow reference, AIRBORNE extracts the corresponding `<id>`. If the reference points to an answer, AIRBORNE uses the Stack Exchange API to retrieve the `<id>` of the associated question. Then, AIRBORNE retrieves the body and metadata of both the question

and all its answers. The body of each post includes any embedded code snippets. The metadata includes information such as the post's creation and modification dates, the author of the post, and any associated licenses. From the body of each post, all code snippets are extracted using the regular expressions: `<pre><code.*?>(.*?)</code></pre>` and `` `([^`\n]+)` ``.

4. Each extracted code snippet is compared against the file where the reference was found, using clone detection. AIRBORNE employs CCFinderSW [164]—an evolution of CCFinder [94]—as the clone detection tool. If a clone is found, the cosine similarity between the Stack Overflow code snippet and the corresponding file is computed. Conversely, if no clone is found, the self-admitted Stack Overflow reference is still retained.

5. AIRBORNE generates the fine-grained SBOM by mapping the identified self-admitted Stack Overflow references to CycloneDX data fields [38]. Each file containing a reference is treated as a CycloneDX component, and the corresponding Stack Overflow references are listed for that component, along with details about any code clones. If a cosine similarity threshold is provided, only references associated with a code clone having a cosine similarity equal to or higher than that threshold are included; in this case, references without a detected clone are excluded as well.

6.1.2 *AIRBORNE in Action*

AIRBORNE can be run via the command line using the syntax below, providing as input one mandatory and three optional parameters:

```
python script/main/airborne.py <project_path>
--sbom_path=<sbom_path>
--stack_exchange_api_key=<stack_exchange_api_key>
--cosine_similarity_threshold=<cosine_similarity_threshold>
```

- `<project_path>` (mandatory), a string specifying the local path to the software system for which the fine-grained CycloneDX SBOM should be generated.
- `<sbom_path>` (optional), a string specifying the local path to an existing CycloneDX SBOM in JSON format to be augmented with the fine-grained information.
- `<stack_exchange_api_key>` (optional), a Stack Exchange API key to increase the request limits when retrieving data from Stack Overflow.
- `<cosine_similarity_threshold>` (optional), a float in the range [0..1] specifying the minimum cosine similarity between a file and referenced Stack Overflow code snippets.

The output of AIRBORNE consists of both the execution log and the fine-grained SBOM. The log provides information about the execution of AIRBORNE, including the analyzed files, HTTP requests made to the Stack Exchange API, and any encountered errors (*e.g.,* Stack Overflow URLs that are neither questions nor answers).

Figure 6.2 shows the example of an SBOM generated through AIRBORNE for the software system hosted on the GitHub repository `https://github.com/PabloCastellano/bormeparser`. This SBOM is formatted according to the "`CycloneDX`" standard, version "`1.6`", and conforms to the schema located at "`http://cyclonedx.org/schema/bom-1.6.schema.json`". It describes one software component of type "`file`" containing one self-admitted Stack Overflow reference. This is reported via an external reference entry of type "`website`": "`http://stackoverflow.com/a/22281062`". Embedded within this file component is a nested component of type "`data`", which indicates that the reference is a Stack Overflow answer and contains a cloned code. The author of the answer—reported as the manufacturer—is "`Martijn Pieters`" and their Stack Overflow profile is available at "`https://stackoverflow.com/users/100297/martijn-pieters`". Being a code clone, the component's data type is set to "`source-code`", and the HTML-formatted body of the corresponding Stack Overflow post is included as an attachment. Additional metadata is provided as properties and includes the "`cosine similarity`"

```
{
  "$schema": "http://cyclonedx.org/schema/bom-1.6.schema.json",
  "bomFormat": "CycloneDX",
  "specVersion": "1.6",
  "components": [
    {
      "type": "file",
      "name": "bormeparser/borme.py",
      "externalReferences": [
        {
          "type": "website",
          "comment": "Self-admitted Stack Overflow Usage",
          "url": "http://stackoverflow.com/a/22281062"
        }
      ],
      "components": [
        {
          "type": "data",
          "name": "https://stackoverflow.com/questions/22281062",
          "description": "A Stack Overflow post (answer) containing code similar to code in the
          ↪  file.",
          "manufacturer": {
            "name": "Martijn Pieters",
            "url": [
              "https://stackoverflow.com/users/100297/martijn-pieters"
            ]
          },
          "data": [
            {
              "type": "source-code",
              "contents": {
                "attachment": {
                  "content": "<p>Turn sets into lists before serializing, or use a custom
                  ↪  <code>default</code> handler to do so:</p>\n\n<pre><code>def
                  ↪  set_default(obj):\n    if isinstance(obj, set):\n        return list(obj)\n
                  ↪  raise TypeError\n\nresult = json.dumps(yourdata,
                  ↪  default=set_default)\n</code></pre>\n"
                },
                "properties": [
                  {
                    "name": "Cosine similarity",
                    "value": "0.4281103162176793"
                  },
                  {
                    "name": "Last activity date of the post",
                    "value": "2014-03-09 11:26:26"
                  },
                  {
                    "name": "Last edit date of the post",
                    "value": "2025-09-19 21:48:18"
                  },
                  {
                    "name": "Creation date of the post",
                    "value": "2014-03-09 11:26:26"
                  }
                ]
              }
            }
          ],
          "licenses": [ {
              "license": {
                "id": "CC-BY-SA-3.0",
                "url": "https://stackoverflow.com/help/licensing"
              }
            }
          ]
        }
      ]
    }
  ]
}
```


Figure 6.2: Example of an SBOM generated through AIRBORNE.

and the "`last activity`, `last edit`, and `creation date of the post`". The only license associated with the Stack Overflow post is "`CC-BY-SA-3.0`".

6.1.3 *AIRBORNE Limitations*

AIRBORNE deals only with explicit URLs to Stack Overflow posts, leaving implicit code reuse undetected. Similarly, it does not capture implicit or explicit code reuse from other (*e.g.*, GitHub) software repositories. While explicit code reuse has been implemented for Stack Overflow, it can be extended to support other Q&A forums or software repositories.

The clone detection works only with Type-1 and Type-2 clones and for the programming languages supported by CCFinderSW. However, CCFinderSW provides an extension mechanism for adding new languages [164]. When no existing SBOM is provided as input, AIRBORNE generates a minimal CycloneDX SBOM. AIRBORNE currently supports SBOMs in JSON format that adhere to the CycloneDX standard. However, these SBOMs are among the most adopted [158, 166].

6.2 PRELIMINARY EMPIRICAL EVALUATION

This section presents a preliminary empirical evaluation of AIRBORNE.

6.2.1 *Study Design*

The aim is to determine whether the generated SBOMs accurately report all the expected self-admitted Stack Overflow references—and, if not, to investigate the reasons behind any omissions. To identify systems with known self-admitted Stack Overflow references, the SOTorrent dataset [13] was leveraged. This dataset maps GitHub files to Stack Overflow posts that have been explicitly referenced within those files. The total number of GitHub repositories reported in the dataset is 438,602. Three programming languages were chosen: C, Java, and Python, as they are both popular and natively supported by CCFinderSW, the clone

detection tool used by AIRBORNE. Consequently, any mappings related to other languages were filtered out, resulting in a dataset comprising 79,129 GitHub repositories, each containing at least one file written in C, Java, and/or Python with a known Stack Overflow reference. The breakdown of these 79,129 repositories is as follows: 45,734 repositories contained only Python files referencing Stack Overflow; 26,004 only Java files; 5,266 only C files, 876 contained both Python and Java files, 750 both C and Java files, 387 both Java and C files, and 112 repositories included files in all three languages (C, Java, and Python). Given the large size of the population, a statistically significant stratified random sample was constructed, using a confidence level of 95% and a margin of error of 5%. The resulting sample size was 383 repositories. The stratified random sampling was based on the combinations of programming languages present in the repositories, ensuring that each stratum was proportionally represented in the final sample.

Across the 383 repositories, the SOTorrent dataset reported 1,140 mappings between files and self-admitted Stack Overflow references. For each repository, AIRBORNE was used to generate a fine-grained SBOM augmented with self-admitted Stack Overflow references. Each repository was examined to understand whether its fine-grained SBOM included the expected self-admitted Stack Overflow references from the dataset. In cases where expected references were missing, these were manually analyzed to determine the root cause. This involved inspecting the content of the corresponding file and reviewing the Stack Overflow post linked in the reference. Finally, the code clones detected between repository files and self-admitted Stack Overflow references were analyzed.

6.2.2 *Evaluation Results*

Out of 383 GitHub repositories, all the self-admitted Stack Overflow references from SOTorrent were found in 368 of them. Not all the self-admitted Stack Overflow references were found for the other 15 repositories because: (i) seven repositories deleted the files where the self-admitted Stack Overflow reference was added; (ii) six repositories removed the self-admitted Stack Overflow ref-

erence from the file where these should have been; and (iii) two repositories both deleted the files where the self-admitted Stack Overflow reference was added and removed the self-admitted Stack Overflow reference from the file where these should have been (of course, for different self-admitted Stack Overflow references). This means that the problem in retrieving the self-admitted Stack Overflow references was not due to AIRBORNE, but rather to the changes that the GitHub repositories underwent from the release of SOTorrent to the date on which this preliminary empirical evaluation was conducted. We also found that, in 43 repositories, all self-admitted Stack Overflow references were associated with a code clone. This number increases to 112 when considering repositories where at least some of the self-admitted Stack Overflow references from the SOTorrent dataset were associated with a code clone, while others were not. This outcome was expected because the referenced Stack Overflow discussion may not include any code snippet, or the snippet may be too short to be detected by the clone detection.

A total of 227 code clones associated with self-admitted Stack Overflow references were identified. The cosine similarity scores for these clones had a mean of 0.66 and a standard deviation of 0.21. The minimum cosine similarity was 0.08, while the maximum reached 1. The median was 0.7, with the 25^{th} percentile (Q1) at 0.49 and the 75^{th} percentile (Q3) at 0.83. These results indicate a generally high level of similarity among the matched code snippets from self-admitted Stack Overflow references, though with some variation across cases. The identification of 227 code clones across 112 software systems—representing 29% of the 383 sampled instances—suggests that reusing code from Stack Overflow is a relatively common practice among developers. The high cosine similarity scores, with values ranging from 0.49 at Q1 to 0.83 at Q3, indicate that much of this reused code undergoes minimal modification. From a legal perspective, this high degree of code similarity raises concerns about code ownership and licensing. When developers reuse code from Stack Overflow without significant modification or proper attribution, they may violate copyright and licensing terms. Stack Overflow content is typically published under the CC BY-SA 3.0 license, which requires attribution regardless of the extent of modification and applies

"share-alike" obligations if the code is redistributed. Failure to comply with these licensing terms could expose developers or organizations to legal risks.

6.2.3 *Threats to Validity*

As for threats to *conclusion validity*, we built a statistically significant stratified random sample of SOTorrent repositories having C, Java, and/or Python files with self-admitted Stack Overflow references, using a 95% confidence level and a 5% margin of error. This might pose a threat to conclusion validity because the randomness of the sample selection process does not rule out selection bias, *i.e.,* the selected repositories may not fully represent the population of repositories described in the SOTorrent dataset.

As for threats to *external validity*, the evaluation may not generalize beyond the GitHub repositories in SOTorrent or to those on other platforms

6.3 DISCUSSION

The results of a preliminary empirical evaluation outline that AIRBORNE can successfully generate fine-grained SBOMs by incorporating provenance information from self-admitted Stack Overflow references in source code. This can be of relevance to software suppliers, choosers, and operators. However, AIRBORNE currently depends on explicit reuse declarations and cannot detect implicit or unattributed reuse, which is increasingly relevant with the rise of AI-generated code. Future research could extend AIRBORNE to identify AI-generated content and include it in SBOMs, further enhancing software transparency and accountability. Finally, extending AIRBORNE to support explicit code reuse from other Q&A forum or software repositories (*e.g.,* GitHub) represents a further future research direction.

TOOL AVAILABILITY

AIRBORNE is available open-source under the Mozilla Public License 2.0 License on GitHub [124].

Part III

STUDIES ON ARTIFICIAL INTELLIGENCE BILL OF MATERIALS

7

MULTIVOCAL LITERATURE REVIEW ON AIBOMS

This chapter is based on the paper entitled "What We Know about AIBOMs: Results from a Multivocal Literature Review on Artificial Intelligence Bill of Materials" [155], submitted to the ACM Transactions on Software Engineering and Methodology (currently at the second round of revision).

The increasing pervasiveness of AI is accelerating the integration of AI components into software supply chains, as well as the development of regulatory frameworks to govern their use. Consequently, traditional conceptions of the software supply chain must be revisited and adapted to accommodate the distinct characteristics of AI components. In this context, the concept of the AIBOM has recently emerged as a means to document and manage AI components. Despite growing attention to supply chains of AI and AI-enabled systems, there remains a significant gap between theory and practice. In particular, the roles and positions of stakeholders with respect to AIBOM are not yet well understood. A structured synthesis of existing knowledge is therefore essential to support researchers with the contextual foundation needed for further studies, and to provide software suppliers, choosers, and operators with guidance rooted in both academic and practical experience. This chapter presents the findings of an MLR on AIBOMs, which synthesizes insights from both Formal Literature (FL) and Gray Literature (GL). Unlike traditional systematic reviews, which primarily draw on peer-reviewed FL sources, MLRs explicitly incorporate practical knowledge from GL (*e.g.*, technical blogs, white papers), thus helping to bridge the gap between academic research and industry practice.

CHAPTER STRUCTURE. Section 7.1 presents the study design. The results are reported in Section 7.2 while their discussion concludes the chapter in Section 7.3.

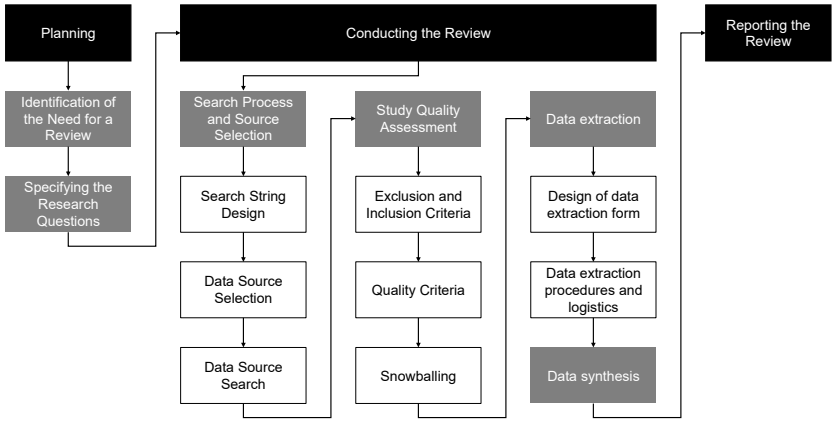


Figure 7.1: Overview of the phases/steps of the MLR.

7.1 STUDY DESIGN

In this section, the study design is described following the guidelines for including GL and conducting MLRs in Software Engineering (SE) by Garousi *et al.* [69]. These are based on the guidelines for performing SLRs in SE by Kitchenham and Charters [98] and Kitchenham *et al.* [97]. An overview of the steps conducted to perform the MLR is shown in Figure 7.1.

7.1.1 *Planning*

In the following, the planning phase of the MLR is presented, which includes establishing the need for an MLR on AIBOMs and defining its goal and RQs.

7.1.1.1 *Identification of the Need for a Review*

Before planning/executing the MLR, an extensive search was performed to ensure that a systematic review of AIBOMs was necessary. To this end, Google Scholar was queried to identify any existing secondary studies on the topic by using the search string below.

Table 7.1: Items to decide whether to include the gray literature in software engineering reviews [69].

#	Item	Response
1	Is the subject "complex" and not solvable by considering only the formal literature?	Yes
2	Is there a lack of volume or quality of evidence, or a lack of consensus on outcome measurement in the formal literature?	Yes
3	Is the contextual information important to the subject under study?	Yes
4	Is it the goal to validate or corroborate scientific outcomes with practical experiences?	Yes
5	Is it the goal to challenge assumptions or falsify results from practice using academic research or vice versa?	Yes
6	Would a synthesis of insights and evidence from the industrial and academic community be useful to one or even both communities?	Yes
7	Is there a large volume of practitioner sources indicating high practitioner interest in a topic?	Yes

Search String for Identifying the Need for a Review

```
("artificial intelligence BOM" OR "ai BOM" OR
"artificial intelligence bill of materials" OR
"ai bill of materials" OR "AIBOM" OR "AI-BOM" OR
"machine learning BOM" OR "ml BOM" OR "machine
learning bill of materials" OR "ml bill of
materials" OR "MLBOM" OR "ML-BOM" OR "data BOM"
OR "data bill of materials" OR "DataBOM" OR
"Data-BOM") AND ("review")
```

The search procedure, inspired by common practices in exploratory literature reviews, entailed reviewing the first ten pages of results [83]. Diminishing returns were monitored, and upon reaching the tenth page without identifying any additional pertinent contributions, the search was concluded. Across the analyzed results, no review of AIBOMs was identified from either FL or GL. This outcome justifies the systematic review on AIBOM presented in this study.

The choice of including GL and, therefore, conducting an MLR was made systematically using a well-defined checklist provided by Garousi *et al.* [69]. Such a checklist is reported, along with the responses, in Table 7.1. Each response can be either "yes" or

"no": one or more "yes" responses justify the inclusion of GL in the systematic review and then the execution of an MLR. Below, there is the motivation for the response, *i.e.,* "yes" response, to all the items shown in such a table.

#1. AIBOM is a rather novel topic, as well as "complex", since it captures various and interconnected aspects of SE and AI (*e.g.,* AI-enabled software systems). Moreover, these aspects can be approached from different perspectives (*e.g.,* practice and research).

#2. FL about AIBOMs is scarce, probably due to the topic's novelty. A lack of FL about AIBOMs was identified during the search for prior systematic reviews. Nevertheless, such a novelty further motivates the need to investigate "current" perspectives about AIBOMs emerging from GL.

#3. Contextual information—which includes the approaches enabling a given technology [25, 50, 141, 142]—is crucial to investigate AIBOM. Indeed, this topic has a high relevance for software suppliers, choosers, and operators who need to manage the supply chain of AI and AI-enabled systems, complying with regulations such as the EU AI Act [58] or President Biden's Executive Order 14028 on Improving the Nation's Cybersecurity [186]. Moreover, AIBOMs are also important in understanding the AI components on which systems rely. This enables, for example, software choosers to make informed decisions, build trust, and assess potential risks, biases, and ethical concerns associated with any AI solution.

#4. The goal is to corroborate formal research on AIBOMs with insights provided by practice. Including GL in the systematic review may lead to findings useful for software suppliers, choosers, and operators, as well as researchers.

#5. The study of FL and GL could shed light on and question common beliefs about AIBOMs. An MLR would provide software suppliers, choosers, and operators, as well as researchers, with more reliable knowledge about AIBOMs.

#6. Reliable knowledge about AIBOMs, emerging from an analysis of both FL and GL, would be useful to software suppliers, choosers, operators, and researchers.

#7. A large volume of evidence from software suppliers, choosers, and operators indicates a strong interest in AIBOM.

7.1.1.2 *Specifying the Research Questions*

Following the guidelines by Garousi *et al.* [69], the Goal-Question-Metric (GQM) methodology [17] was leveraged to formalize the goal and RQs of the MLR. Specifically, the *goal* was to analyze FL and GL about AIBOMs for the *purpose* of improving their understanding *with respect to* (i) motivations/benefits, (ii) laws/regulations, (iii) elements, (iv) use cases, (v) approaches/-tool categories/techniques, and (vi) limitations/challenges. The *perspective* is that of software suppliers, choosers, operators, and researchers. The *context* consists of research papers retrieved from specialized databases and digital libraries (for the FL) and resources retrieved from general web search engines (for the GL).

Based on the goal mentioned above and the classification scheme proposed by Easterbrook *et al.* [52], six *exploratory* RQs of type *descriptive-classification* were formulated. This type of RQ is appropriate when attempting to understand a given topic. The RQs are described below.

RQ1. *What are the motivations and benefits of using AIBOMs?*

This RQ aims to study the motivations and benefits of using AIBOMs. Understanding these motivations and benefits—even those that might be deemed clear or self-evident—is essential to assessing the value proposition of AIBOMs in practical settings. Answering this RQ sheds light on the practical impact that AIBOMs could have. Indeed, the findings of this RQ are expected to provide software suppliers, choosers, operators, and researchers with insights into the drivers and practical advantages of integrating AIBOMs into software development.

RQ2. *What are the laws and regulations related to using AIBOMs?*

Software suppliers, choosers, and operators need to comply with regulations when distributing, acquiring, and managing software [156]. As AI and AI-enabled systems become increasingly pervasive, software suppliers, choosers, and operators must navigate a growing body of regulations related, for example, to transparency, accountability, data provenance, and ethical AI. RQ2 seeks to identify which laws, standards, and compliance requirements can be addressed or supported by adopting AIBOMs. Examination of these aspects aims to clarify the role AIBOMs can play in helping organizations meet regulatory obligations and reduce the risk of non-compliance throughout the AI lifecycle.

RQ3. *What elements are included or worth including in AIBOMs?*

This RQ investigates the essential elements that constitute an AIBOM. As a structured artifact, an AIBOM is expected to capture comprehensive information about the AI and AI-enabled system it documents—including, but not limited to, ML models, training and test datasets, software dependencies, configuration parameters, and metadata related to performance, fairness, and provenance. Through RQ3, the aim is to identify which elements are currently included in existing AIBOMs and to propose additional elements that may be worth including. Answering this RQ would deepen the understanding of the data fields that should constitute AIBOMs and their differences with the data fields expected for traditional SBOMs. The findings would drive the design of AIBOM standards and the development of tools for AIBOM automated generation and consumption.

RQ4. *What are the use cases where AIBOMs are used?*

This RQ aims to study the use cases associated with AIBOMs consumption. This includes activities related to the development and management of software systems supported using AIBOMs. Answering this RQ sheds light on best practices that software suppliers, choosers, and operators can adopt to exploit AIBOMs.

RQ5. *What are the approaches, tool categories, and techniques to deal with AIBOMs?*

This RQ aims to unveil the approaches, tool categories, and techniques to deal with AIBOMs. That is, the aim is to understand the approaches used by software suppliers, choosers, and operators to generate and consume AIBOMs, as well as the tools that could be employed for this purpose.

RQ6. *What are the limitations and challenges of AIBOMs?*

The need to manage the supply chain of AI and AI-enabled systems has been growing in the past few years, and AIBOM provides a way to support this. However, the adoption of AIBOMs could be hindered by limitations and challenges associated with such a novel technology. This RQ aims to unveil such limitations and challenges associated with AIBOM, along with possible solutions.

7.1.2 *Conducting the Review*

In the following, the conduction phase of the MLR is presented, which includes the search process and source selection, study quality assessment, data extraction, and data synthesis.

7.1.2.1 *Search Process and Source Selection*

The search process and source selection involve designing the search string, selecting the data sources, and conducting the searches within those sources.

SEARCH STRING DESIGN A search string was constructed to retrieve both FL and GL, guided by the key terms derived from the RQs and supplemented by relevant synonyms and alternative expressions. An iterative refinement process was employed to obtain the final search string—*e.g.*, evaluating the top results for relevance to assess how well the search string meets the RQs, and balancing breadth and specificity of the search string to enhance coverage and relevance—during which it was progressively adjusted to reach the final used version. The final search string used is the following:

Search String for the Multivocal Literature Review

```
("artificial intelligence BOM" OR "ai BOM" OR
"artificial intelligence bill of materials" OR
"ai bill of materials" OR "AIBOM" OR "AI-BOM" OR
"machine learning BOM" OR "ml BOM" OR "machine
learning bill of materials" OR "ml bill of
materials" OR "MLBOM" OR "ML-BOM" OR "data BOM"
OR "data bill of materials" OR "DataBOM" OR
"Data-BOM") AND ("motivation*" OR "benefit*"
OR "law*" OR "regulation*" OR "element*" OR
"dataset*" OR "model*" OR "data field*" OR "use
case*" OR "approach*" OR "tool*" OR "technique*"
OR "limitation*" OR "challeng*")
```

DATA SOURCE SELECTION Below, the data sources chosen to search for FL and GL are presented:

FL. The following three data sources were used: Scopus [162], IEEE Xplore [85], and ACM Digital Library [1]. Scopus is a broad-coverage abstract database, while IEEE Xplore and ACM Digital Library are full-text databases with more limited coverage. These three are considered among the top computer science FL data sources [135], have been used in many other literature reviews [68, 71, 88, 97], and are recommended by the guidelines [69, 98].

GL. The Google search engine [80] was leveraged. It is a general web search engine that has been used by other MLRs [3, 110], including in the field of SE [68, 71, 88]. The Google search engine is also recommended by the guidelines for including GL in systematic reviews [4, 69].

The plan was to collect all the literature resulting from the search on Scopus, IEEE Xplore, and ACM Digital Library. However, for the GL, this could not be accomplished due to the large number of results that the Google search would have produced [69, 131]. Therefore, it was necessary to decide on *data exhaustion stopping criteria* [69] for the GL search. Following the suggestions by Garousi *et al.* [69, 71], an *effort bounded* stopping

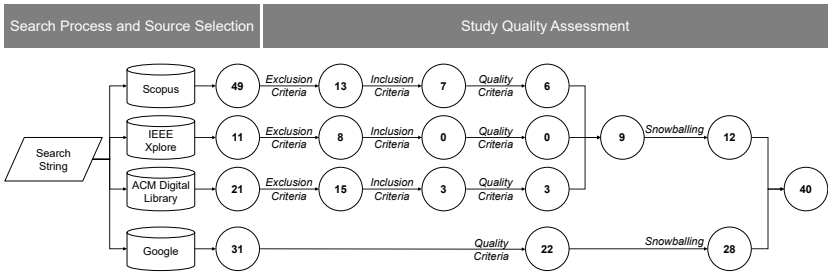


Figure 7.2: Source selection process.

criterion augmented with an *evidence exhaustion* stopping criterion was employed. First, a limited number of pages produced by the Google search were analyzed (effort-bound). The first ten pages of results were chosen for analysis because this threshold has been used by other authors [101, 103]. The search was continued further if the results of the last page still provided additional evidence (evidence exhaustion). Note that the effort-bound criterion was applied to prevent the impractical task of reviewing hundreds/thousands of Google search results. Nonetheless, in accordance with the evidence exhaustion criterion, the search would have continued beyond the initial ten pages whenever potentially relevant sources appeared. In practice, diminishing returns were observed after the first few pages, with a substantial drop in relevance; by the tenth page, no additional valuable sources were identified.

DATA SOURCE SEARCH Figure 7.2 illustrates the source selection process, including the search conducted across the selected data sources. The data sources were queried on February 3, 2025. As for FL, a total of 81 papers were retrieved: 49 from Scopus, 11 from IEEE Xplore, and 21 from the ACM Digital Library. As for the GL, the Google search returned over 250,000 results. Following the defined stopping criteria, the first ten pages of results were reviewed and 31 relevant sources were identified.

7.1.2.2 *Study Quality Assessment*

Once the potentially relevant primary sources were collected, relevance was assessed. This involved defining detailed selection

Table 7.2: Exclusion, inclusion, and quality criteria for FL.

Exclusion Criteria	Inclusion Criteria	Quality Criteria
EC1 - The paper is not written in English EC2 - The paper is not in the field of software engineering or computer science EC3 - The paper is a secondary study EC4 - The paper is a duplicate	IC1 - The paper deals with (Software) Bill of Materials for artificial intelligence IC2 - The paper has the full text available	Q1 - The paper clearly reports valuable information for the review Q2 - The paper clearly reports its RQs and/or objectives Q3 - The paper clearly reports the results/findings

criteria and then systematically applying them (see Figure 7.2). The following three types of selection criteria were distinguished:

- *Exclusion Criteria*. They are applied only to the title, abstract, and keywords. The source is excluded if any of the criteria are met.
- *Inclusion Criteria*. They are applied only to a given source's full text. The source is included if all criteria are met.
- *Quality Criteria*. They are applied only to a given source's full text. These criteria are used to evaluate the quality of the sources. The source is included if deemed of enough quality, *i.e.*, its evaluation meets or exceeds a certain threshold.

Following the guidelines for including GL and conducting MLRs in SE by Garousi *et al.* [69], distinct criteria were applied for FL and GL sources. In particular, only quality criteria were applied to the GL sources, which were distinct from those used for FL. The rationale is that, in the case of GL, inclusion and exclusion criteria often overlap with, and are sometimes integrated into, quality criteria [69].

EXCLUSION AND INCLUSION CRITERIA Table 7.2 shows the exclusion and inclusion criteria for including the FL. Each criterion was evaluated as either *True* or *False*. Applying these criteria enabled the inclusion of only sources related to the topic of interest, namely AIBOMs. It is worth mentioning that BOMs have been extensively used and studied in other fields, such as manufacturing, so it was necessary to ensure that the selected sources

explored the use of BOMs for AI. The application of the exclusion criteria reduced the initial set of 81 papers to 36, and the subsequent application of the inclusion criteria further narrowed this down to nine scientific papers (*i.e.,* FL sources).

QUALITY CRITERIA Table 7.2 shows the quality criteria applied to the FL sources. In line with established guidelines [69, 97, 98], each quality criterion could be answered using a 3-point Likert scale (yes = 1, partly = 0.5, and no = 0). The sum of these values reflects the quality score for a given source. Then, the minimum acceptable quality score was defined as the maximum possible quality score divided by two [69]. FL sources with a quality score lower than 1.5 (3/2) were excluded. Applying the quality criteria resulted in the exclusion of only one paper, leading to the inclusion of nine papers for the FL in the MLR.

Table 7.3 shows the quality criteria applied to the GL sources. These criteria were not arbitrarily defined but were adopted directly from the established guidelines by Garousi *et al.* [69]. Each criterion was associated with one or more questions, and each question could be answered according to a 3-point Likert scale (yes = 1, partly = 0.5, and no = 0) [69, 97, 98]. GL sources with a quality score lower than 9.5 (19/2) were excluded [69]. The application of the quality criteria resulted in the exclusion of nine sources, leading to the selection of 22 sources for the GL.

The FL sources were imported into one spreadsheet and the GL sources into another. Each spreadsheet reported the criteria for the corresponding sources, *i.e.,* FL or GL. The exclusion, inclusion, and quality criteria for each source were applied independently by two researchers involved in the study. In case of disagreement, the final decision on including a given source was made through a discussion between the two researchers who applied the inclusion, exclusion, and quality criteria. All disagreements were successfully resolved between the two researchers.

SNOWBALLING After applying the inclusion, exclusion, and quality criteria to the initial set of sources, snowballing was performed following Wohlin's guidelines [193] for systematic literature studies. Snowballing involves using reference lists (backward snowballing) and citation lists (forward snowballing) to identify

Table 7.3: Quality assessment checklist of gray literature for software engineering [69].

Criteria	Question	
Authority of the producer	QGL1	Is the publishing organization reputable? *e.g.*, the Software Engineering Institute (SEI)
	QGL2	Is an individual author associated with a reputable organization?
	QGL3	Has the author published other work in the field?
	QGL4	Does the author have expertise in the area? (*e.g.*, job title principal software engineer)
Methodology	QGL5	Does the source have a clearly stated aim?
	QGL6	Does the source have a stated methodology?
	QGL7	Is the source supported by authoritative, contemporary references?
	QGL8	Are any limits clearly stated?
	QGL9	Does the work cover a specific question?
	QGL10	Does the work refer to a particular population or case?
Objectivity	QGL11	Does the work seem to be balanced in presentation?
	QGL12	Is the statement in the sources as objective as possible? Or, is the statement a subjective opinion?
	QGL13	Is there a vested interest? *e.g.*, a tool comparison by authors that are working for a particular tool vendor
	QGL14	Are the conclusions supported by the data?
Date	QGL15	Does the item have a clearly stated date?
Position	QGL16	Have key related GL or formal sources been linked to / discussed?
Novelty	QGL17	Does it enrich or add something unique to the research?
	QGL18	Does it strengthen or refute a current position?
Outlet type	QGL19	1st tier GL: High outlet control/ High credibility: Books, magazines, theses, government reports, white papers; 2nd tier GL: Moderate outlet control/ Moderate credibility: Annual reports, news articles, presentations, videos, Q/A sites (such as StackOverflow), Wiki articles; 3rd tier GL: Low outlet control/ Low credibility: Blogs, emails, tweets

additional relevant papers. This process helped mitigate the risk of omitting important literature from this MLR [193].

Snowballing was applied to the FL sources that passed the inclusion and exclusion criteria, and to the GL sources that passed the quality criteria. Each source identified through snowballing was evaluated using the same criteria previously defined for FL and GL, depending on its type. Backward and forward snowballing were performed iteratively until no new sources were

found in the reference or citation lists—thus, the process was repeated multiple times for both FL and GL.

For backward snowballing of FL sources, the reference lists of the selected papers were examined; for forward snowballing, Google Scholar was used to identify citing sources. Regarding GL sources, and in line with the guidelines by Garousi *et al.* [69], backward snowballing was performed by extracting links and references from each source using *Link Gopher*[1], *Extract Link PDF*[2], and *Link Grabber*[3]; to identify forward links to GL sources, *Ahrefs Link Grabber*[4], a web-based service that tracks backlinks, was used.

Through snowballing, three additional papers were identified for the FL and six additional sources for the GL. As a result, the final selection comprised 12 FL papers and 28 GL sources, totaling 40 sources used for data extraction and synthesis.

7.1.2.3 *Data extraction*

For the data extraction phase, the design of the data extraction form and the procedures and logistics used for data extraction are presented.

DESIGN OF DATA EXTRACTION FORM In designing the data extraction form, traceability to the RQs of the MLR was ensured. This approach was necessary to identify relevant elements for addressing each RQ. Each RQ was systematically mapped to specific attributes or aspects to clarify the data to be extracted. For example, in the case of RQ1 (What are the motivations and benefits of using AIBOMs?), "motivations" and "benefits" were identified as key aspects. The data extraction form was defined through brainstorming and discussion among all researchers contributing to the study. In addition to RQ-specific data, more generic data such as publication year, contribution type, and

1 https://chromewebstore.google.com/detail/bpjdkodgnbfalgghnbeggfbfjpcfamkf?utm_source=item-share-cb
2 https://chromewebstore.google.com/detail/fchghbfimckdiapkacfaljimmkojgfhi?utm_source=item-share-cb
3 https://chromewebstore.google.com/detail/caodelkhipncidmoebgbbeemedohcdma?utm_source=item-share-cb
4 https://ahrefs.com/backlink-checker

research type were also extracted, following recommendations from existing guidelines [69, 98]. Similar to other works [68, 71, 88], a pilot data extraction was conducted on a random sample of sources to validate and refine the data extraction form.

DATA EXTRACTION PROCEDURES AND LOGISTICS All sources were imported into two spreadsheets hosted on Google Sheets, one for each researcher involved in the data extraction. A new column was added for each field in the data extraction form. This structure ensured explicit traceability links between the sources and the extracted data. Two researchers independently extracted data from the sources in random order, recording all information necessary to answer each RQ. In cases where different pieces of information were extracted for a given source, a discussion was planned between the two authors to deal with this discrepancy. This collaborative process helped ensure the quality and validity of the extracted data, in line with established guidelines for literature reviews [69]. All extracted data are available in the online supplementary material [9].

7.1.2.4 *Data synthesis*

During the data synthesis phase, the previously extracted data were synthesized using qualitative coding techniques [111], specifically applying open coding followed by axial coding. This approach is considered to be particularly appropriate for the data synthesis of GL sources, which often contain rich narrative content suitable for qualitative synthesis [69].

During the open coding phase, the extracted data for each field in the data extraction form were systematically examined, and descriptive codes capturing the underlying concepts were assigned. Whenever a new concept emerged (*i.e.*, it was not yet represented by an existing code), a new code was created and a corresponding column was added to the spreadsheet. The relevant data segment was then placed into that column. If a concept had already been coded previously, the new data segment was added to the existing column for that code. This iterative process involved constant comparison and refinement of codes to ensure they accurately reflected the meaning of the data. A balance be-

tween granularity and abstraction was retained, following best practices for qualitative analysis [111].

Lastly, axial coding [173, 187] was conducted. This was necessary to group codes identified during the open coding process into higher-level categories/themes [69]. Together, open and axial coding enabled the synthesis of the extracted data and the answering of the RQs.

7.1.3 *Reporting the Review*

The final phase of the MLR was the reporting of findings. In line with the guidelines for systematic reviews [69, 98], this phase was carefully planned to ensure the results are valuable to software suppliers, choosers, operators, as well as researchers. For each RQ, the identified codes are presented, organized under their corresponding broader themes (axial codes). For each code, the number of sources reporting it is indicated in parentheses. In addition to listing each code and the number of sources that mentioned it, information from sources that elaborated on the code beyond a mere mention is also included.

A single source may contribute to multiple codes (also for the same RQ). Although codes reported by one or two sources might not be meaningful, these were reported because it is common in MLRs (*e.g.,* [19, 101, 138, 174]). They could still offer valuable insights in specific contexts or highlight unique perspectives worth further exploration. This is especially relevant in the case of AIBOMs, a new and rapidly evolving area where the literature is still taking shape. For example, novel approaches or early-stage practices may not yet have gained widespread attention. Including low-frequency codes allows for a more comprehensive and inclusive synthesis that acknowledges the diversity of viewpoints and experiences present in the current body of knowledge. While these should be interpreted with caution, they help ensure that potentially important but underrepresented aspects are not overlooked and may inform future research directions or practical discussions.

Implications and actionable insights for software suppliers, choosers, and operators are also highlighted, as well as possible future research directions for researchers.

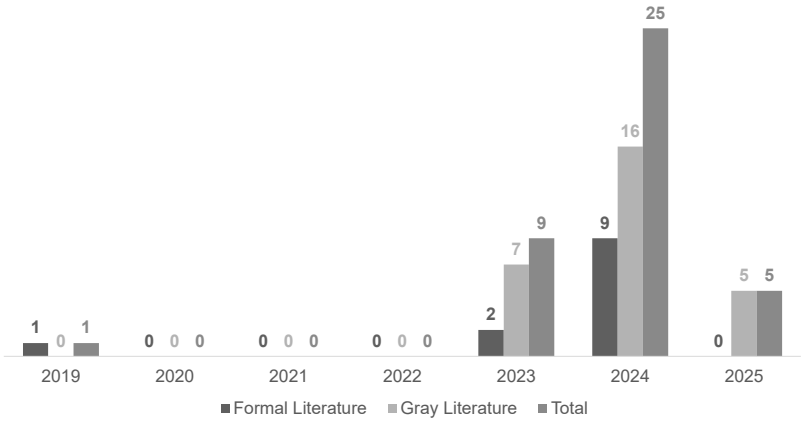


Figure 7.3: Number of sources published per year (up to February 2025).

Finally, the review dataset was openly shared as supplementary material [9]. By making the extracted data and synthesis outcomes available in a structured and publicly accessible format, transparency is promoted, replication is supported, and reuse or extension in future studies is enabled.

7.2 RESULTS

This section first reports descriptive statistics from the analyzed sources, followed by the results for each studied RQ.

7.2.1 *Statistics on the Analyzed Sources*

Figure 7.3 presents the number of sources published yearly. The first source was published in 2019, followed by three years of publication inactivity (2020-2022). Publication activity resumed in 2023 with nine sources and peaked in 2024 with 25 sources (more than double the previous year). By February 2025, five additional sources will have been published, suggesting that the upward trend will likely continue.

Figure 7.4 shows the distribution of the types of sources for FL and GL. Blog post is the largest single type of sources, ac-

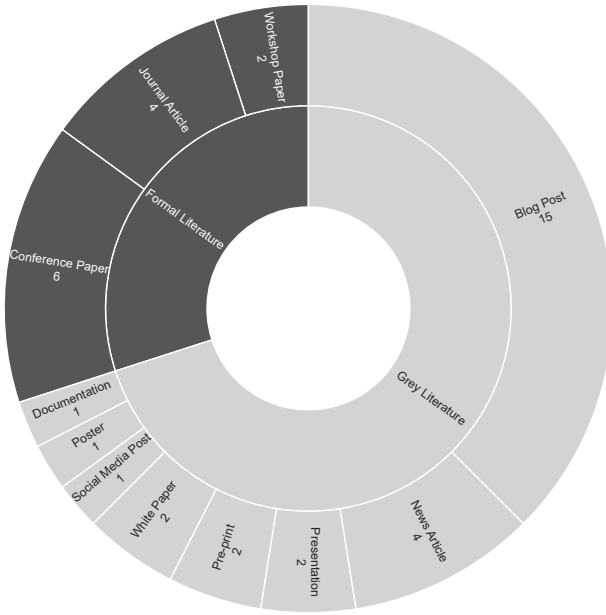


Figure 7.4: Overview of literature source types using a sunburst chart.

counting for 15 of the 40 sources (37.5%), followed by conference papers, accounting for 6 of the sources (15%). Both journal and news articles account for 4 of the sources (10%), while each of the workshop paper, white paper, pre-print, and presentation accounts for 2 of the sources (5%). Documentation, poster, and social media post were found, each accounting for one of the sources (2.5%).

Finally, Table 7.4 presents the venues—both conference proceedings and journals—where the FL sources were published, along with the corresponding publication counts. The FL sources are published across different venues, with no single venue standing out as prevailing. The venues with the highest number of publications—each featuring two papers—are: the IEEE/ACM International Conference on Software Engineering, the IEEE/ACM International Conference on AI Engineering–Software Engineering for AI, and the ACM Transactions on Software Engineering and Methodology.

Table 7.4: Number of FL sources per venue.

Venue	#Total
IEEE/ACM International Conference on Software Engineering	2
IEEE/ACM International Conference on AI Engineering—Software Engineering for AI	2
ACM Transactions on Software Engineering and Methodology	2
Concurrency and Computation: Practice and Experience	1
IEEE International Conference on Blockchain	1
IEEE/ACM International Conference on Program Comprehension	1
IEEE/ACM International Workshop on Engineering and Cybersecurity of Critical Systems & IEEE/ACM International Workshop on Software Vulnerability	1
International Workshop on Science Gateways	1
The Journal of Defense Modeling and Simulation	1

7.2.2 *RQ1. What are the motivations and benefits of using AIBOMs?*

Seventeen distinct motivations and benefits for using AIBOMs were identified, each targeting improvements in either software systems or processes (see Table 7.5). The following elaborates on each motivation/benefit, categorizing them into those that improve software systems and those that improve software processes.

7.2.2.1 *Improvement of Software Systems*

Transparency (35) is the most frequently mentioned motivation and benefit for using AIBOMs. As AI and AI-enabled systems become increasingly widespread, it is vital to ensure the transparency of both traditional and AI components [113, 129]. Due to their nature, AI components are characterized by greater complexity and opacity than traditional software components [92]. AIBOMs achieve transparency by providing a clear and detailed inventory of all components constituting AI and AI-enabled systems [81, 105, 160, 170]. This inventory includes models, data sources, synthetic data, hardware, tools, and methodologies employed in the development and delivery of the systems [81, 105, 160, 170].

Table 7.5: Summary of codes for RQ1.

Axial Codes	Codes	#Total	#FL	#GL	FL Sources	GL Sources
Software System Improvement	Transparency	35	10	25	[15, 21, 107, 139, 146, 170, 189, 195, 196, 201]	[16, 20, 30, 48, 54, 72, 81, 92, 95, 102, 105, 113, 122, 129, 130, 159, 160, 165, 175, 176, 183, 208–210, 212]
	Security	24	4	20	[14, 21, 146, 189]	[16, 20, 30, 48, 72, 92, 95, 102, 105, 113, 129, 130, 169, 175, 176, 183, 184, 208, 210, 212]
	Responsibility	21	5	16	[106, 107, 146, 196, 201]	[30, 48, 72, 81, 105, 122, 130, 159, 160, 165, 169, 175, 176, 184, 209, 210]
	Trust	15	4	11	[21, 106, 146, 170]	[16, 54, 81, 105, 130, 159, 175, 176, 184, 209, 212]
	Accountability	14	6	8	[15, 21, 106, 146, 195, 196]	[30, 81, 105, 122, 129, 176, 208, 212]
	Traceability	12	6	6	[14, 15, 21, 106, 196, 201]	[20, 54, 95, 105, 130, 159]
	Safety	10	1	9	[196]	[16, 20, 48, 72, 81, 95, 113, 160, 183]
	Reproducibility	6	2	4	[106, 170]	[72, 81, 105, 176]
	Reliability	2	1	1	[106]	[159]
	Explainability	1	1	0	[21]	
Software Process Improvement	Regulatory Compliance	21	4	17	[106, 107, 146, 196]	[16, 20, 54, 72, 81, 95, 102, 105, 122, 129, 130, 159, 160, 175, 209, 210, 212]
	Risk Management	19	3	16	[15, 146, 196]	[16, 20, 30, 48, 72, 95, 102, 105, 122, 130, 160, 175, 176, 184, 210, 212]
	License Compliance	10	3	7	[106, 146, 201]	[16, 20, 113, 122, 130, 175, 210]
	Procurement	6	1	5	[107]	[72, 92, 175, 184, 209]
	Collaboration	4	0	4		[72, 102, 175, 210]
	Maintenance and Evolution	4	1	3	[196]	[130, 160, 208]
	Automation	3	0	3		[16, 30, 54]

AIBOMs can also be used for supporting **traceability** (12), that is, to track the provenance of the components constituting AI and AI-enabled systems [21, 95, 201]. For example, AIBOMs can track the lineage of AI models, their intended use, where they came from, and how they were trained [54]. Regarding the various datasets that might have been employed, traceability enables the tracking of data flow from its origin to its final storage or use [106].

Transparency and traceability enable several other benefits/motivations for using AIBOMs [160, 183]. For example, an in-depth understanding of the AI components constituting a given system could help assess how these components affect **reliability** (2), **safety** (10), or **security** (24). As for security, AIBOMs facilitate managing risks originating from vulnerabilities and supply chain attacks [16, 175]. Since AI models may require software libraries to run, process inputs and outputs, and use hardware, these libraries constitute vectors for potential vulnerabilities and supply chain attacks [16]. A further security risk is represented by source code generated through AI models trained on compromised datasets [175]. AIBOMs could be leveraged to identify and track the provenance of code generated by AI models.

Responsibility (21) represents another important motivation/benefit. AIBOMs help ensure that AI and AI-enabled systems are developed and deployed responsibly, thus meeting ethical standards [146, 159, 160]. This is possible because AIBOMs can also document considerations about possible biases or limitations of AI models [81, 105]. Eventually, AIBOMs can also help determine liability when AI and AI-enabled systems cause harm or make erroneous decisions [196]. Responsibility is closely associated with **accountability** (14). AIBOMs can offer a tangible record against which stakeholders can be identified and held accountable for their decisions and operations [106, 146]. AIBOMs ensure accountability as stakeholders would be aware of the origins, components, and performance metrics of AI and AI-enabled systems [81, 105]. Such accountability is crucial as society demands increasing accountability for such systems, which are increasingly used commercially and in the public sector [15].

A further motivation for using AIBOMs is building **trust** (15). This is achieved by documenting the components constituting AI and AI-enabled systems, how they work, and how their stakeholders are safeguarded from potential risks [146, 175, 176]. An AIBOM (like any other kind of BOM) can also offer enough information for **reproducibility** (6). That is, AIBOMs can provide enough information to recreate/reproduce the AI and AI-enabled system it describes, including its AI components and their behaviors/results [81, 105]. Reproducibility can also involve data,

e.g., operations to transform a dataset from one version to another [106]. Lastly, an AIBOM could support **explainability** (1).

7.2.2.2 *Improvement of Software Processes*

Regulatory compliance (21) was the most mentioned motivation and benefit for using AIBOMs to improve software processes. As AI becomes increasingly pervasive, new rules, laws, directives, and regulations are likely to be introduced and followed [72, 105, 129, 159, 160]. AIBOMs can be pivotal in ensuring and demonstrating regulatory compliance of AI and AI-enabled systems [72, 105, 129, 159, 160]. In particular, industries subjected to strict regulations (*e.g.*, healthcare or manufacturing) might benefit from AIBOMs to safeguard user data privacy and copyright [81, 105]. Other aspects that might be regulated, and for which AIBOMs can be useful, relate to provenance. For example, regulations might impose restrictions on where the training data is sourced, where the infrastructure is based, or where the AI model is deployed and available for use [105, 106]. Another aspect to which AI needs to comply is licensing. AIBOMs also document licensing information and, thus, support **license compliance** (10). Just like traditional software components, AI components have (and may inherit) licenses, which might be very permissive (*e.g.*, Apache 2.0 license) or restrictive (*e.g.*, copyleft licenses) [16]. In addition, the different parts of an AI model (*e.g.*, training data or architecture) can each be separately licensed [16]. Licensing also involves AI-generated code [175, 201].

AIBOMs support the **automation** (3) of software processes. AIBOMs are typically distributed as JSON or HTML files [30] and follow a unified standard like SPDX or CycloneDX [54]. Therefore, the list of components provided by AIBOMs is structured and machine-readable [16, 54]. This enables the automat a variety of activities, such as responding to security incidents [16, 30]. AIBOMs also support **collaboration** (4) within organizations [72] and the **maintenance and evolution** (4) of AI and AI-enabled systems. The latter motivation/benefit is particularly important when dealing with AI components because, differently from traditional software that once deployed remains static in its behavior, AI models may evolve as they process new data [196].

AIBOMs can be leveraged for the **procurement** (6) of AI components. For example, a primary concern of the US Army is adopting the best AI components from industry into their systems [92, 184]. In doing so, the US Army must be assured about the safety of the AI models it relies on, especially because it has to re-train such models for military purposes using sensitive or classified data [92]. In general, AIBOMs provide support to **risk management** (19) by mitigating potential threats emerging from AI [30, 146, 160, 175, 176]. For example, an AIBOM can help detect and then address issues like outdated libraries, insecure dependencies, or issues with datasets or models [176].

Answer to RQ1

The most mentioned benefits for the *improvement of software systems* are enhanced **transparency** and **traceability**. Both have important impact for **security** and **safety**. In terms of *software process improvement*, AIBOMs are particularly important for **regulatory compliance**. They are considered to help organizations meet legal and industry standards, but also help to increase **trust**, and aid **risk management**.

7.2.3 *RQ2. What are the laws and regulations related to using AIBOMs?*

Twenty-one laws and regulations related to the use of AIBOMs were identified. These laws and regulations were organized into five broad categories based on the regulatory bodies that proposed them (see Table 7.6). The regulatory bodies are: (i) international organizations, (ii) European Union, (iii) United States, (iv) Canada, (v) Singapore, and (vi) China.

7.2.3.1 *International Organizations*

The **ISO/IEC 42001 - AI Management System Standard** (3) was introduced by the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC). It establishes governance across the AI lifecycle [176]. AIBOMs facilitate compliance with ISO/IEC 42001 by providing structured documentation of model provenance, training data sources, and

Table 7.6: Summary of codes for RQ2.

Axial Codes	Codes	#Total	#FL	#GL	FL Sources	GL Sources
International Organizations	IEEE Ethical Technology Standards - P70xx series	1	0	1		[20]
	ISO/IEC 42001 - AI Management System Standard	3	0	3		[20, 54, 176]
	National Institute of Standards and Technology (NIST) AI Risk Management Framework	3	0	3		[54, 130, 176]
Canada	Principles for responsible, trustworthy and privacy-protective generative AI technologies	1	0	1		[165]
China	Basic security requirements for generative artificial intelligence service	1	0	1		[165]
	Measures for the Management of Generative Artificial Intelligence Services	2	0	2		[81, 165]
	Personal Information Protection Law of the People's Republic of China	1	0	1		[165]
European Union	EU AI Act	7	0	7		[20, 54, 81, 122, 129, 165, 176]
	EU General Data Protection Regulation - GDPR	4	2	2	[106, 201]	[81, 165]
	EU Product Liability Directive	1	1	0	[196]	
	European Medicines Agency (EMA) Requirements for Medical Devices	1	0	1		[20]
Singapore	Model ai governance framework for generative ai	1	0	1		[165]
United States	California Consumer Privacy Act	2	0	2		[129, 165]
	Cyber Supply Chain Management and Transparency Act	1	1	0	[146]	
	Internet of Things Cybersecurity Improvement Act	1	1	0	[146]	
	US Copyright Law	1	1	0	[201]	
	US Executive Order 13960 - Promoting the Use of Trustworthy Artificial Intelligence in the Federal Government	1	0	1		[122]
	US Executive Order 14028 - Improving the Nation's Cybersecurity	6	3	3	[21, 146, 170]	[30, 113, 129]
	US Executive Order 14110 - Safe, Secure, and Trustworthy Development and Use of Artificial Intelligence	3	0	3		[105, 129, 175]
	US Food and Drug Administration (FDA) Requirements for Medical Devices	1	0	1		[20]
	US Health Insurance Portability and Accountability Act	1	0	1		[81]

security controls [176]. AIBOMs can also support the **IEEE Ethical Technology Standards - P70xx series** (1)—developed by the Institute of Electrical and Electronics Engineers (IEEE)—which specify the essential documentation required to support ethical AI handling [20]. Bennet *et al.* [20] examined the compliance of the SPDX 3.0 standard with these IEEE and ISO standards. Their analysis assessed whether SPDX 3.0 includes data fields that capture the information indicated by these IEEE and ISO standards. SPDX 3.0 provides data fields for documenting such information, although those fields are not mandatory. AIBOMs also align with the **National Institute of Standards and Technology (NIST) AI Risk Management Framework** (3). The governance, transparency, and continuous monitoring led by AIBOMs can help organizations document risks across the AI lifecycle, ensuring AI deployments adhere to trustworthy AI principles [176].

7.2.3.2 *European Union*

The EU is advancing significant regulatory requirements for AI models as part of the **EU AI Act** (7). The latter, effective from 1 August 2024, aims to regulate AI by establishing rules for AI and AI-enabled systems to design and deploy them in compliance with existing laws and fundamental rights [20]. The EU AI Act requires that all systems with embedded AI must be assessed before being put on the market and throughout their lifecycle [81]. This legislation categorizes AI and AI-enabled systems based on their risk levels and imposes more stringent requirements on high-risk applications to enhance transparency and accountability [20]. Such risks include safety, security, discrimination, privacy, and licensing [122, 176]. The systems subjected to the EU AI Act are not limited to software systems, but they may also include, for example, toys, aircraft, cars, medical devices, and elevators [81]. The EU AI Act also regulates generative AI [165]. For instance, it mandates that AI-generated deepfake content must disclose a machine-readable and detectable watermark indicating that such content has been artificially manipulated [165]. AIBOMs enables compliance with regulations like the EU AI Act by tracking the risks of AI, *e.g.*, by identifying components and

assuring proper documentation for audits [176]. The SPDX 3.0 standard was designed to comply with the EU AI Act [20].

AIBOMs can also facilitate compliance with pre-existing regulations that were not conceived for dealing with AI and AI-enabled systems. For example, in the case of the **EU General Data Protection Regulation - GDPR** (4), AIBOMs can help comply with general privacy requirements such as the right to be informed, the right of access, right to rectification, or right to erasure [165, 201]. Further examples of EU regulations that could be facilitated by AIBOMs are the **European Medicines Agency (EMA) Requirements for Medical Devices** (1) and the **EU Product Liability Directive** (1).

7.2.3.3 *United States*

The **US Executive Order 13960 - Promoting the Use of Trustworthy Artificial Intelligence in the Federal Government** (1) introduced principles required for understanding AI technologies used in software systems, including accountability, transparency, responsibility, traceability, and regulatory monitoring [122]. The **US Executive Order 14028 - Improving the Nation's Cybersecurity** (6) paved the way for the large-scale adoption of SBOMs. It mandated software vendors to distribute the SBOMs of the software systems commercialized to the US government [30, 170]. Software systems that incorporate AI components must still comply with this regulation and, thus, use AIBOMs. The Executive Order 14028 was anticipated by previous regulations [146], such as the **Cyber Supply Chain Management and Transparency Act** (1) and the **Internet of Things Cybersecurity Improvement Act** (1). More recently, the **US Executive Order 14110 - Safe, Secure, and Trustworthy Development and Use of Artificial Intelligence** (3) was released. It aims to shape the priorities for future laws concerning AI [105]. Similarly to the EU AI Act, the Executive Order 14110 may require the disclosure of material created with generative AI, such as via an electronic watermark [175]. AIBOMs following SPDX 3.0 proved to be valid also with respect to the **US Food and Drug Administration (FDA) Requirements for Medical Devices** (1) [20]. AIBOMs could also help to deal with copyrighted content in the training of AI models and copy-

right infringements [201], thus facilitating the **US Copyright Law** (1). Other regulations that have been introduced in the US to regulate AI are the **California Consumer Privacy Act** (2) and the **US Health Insurance Portability and Accountability Act** (1).

7.2.3.4 *Canada*

Canadian regulators released the guidance **Principles for responsible, trustworthy and privacy-protective generative AI technologies** (1), which enforces requirements related to AI, such as AI-generated watermarking [165]. This guidance could be facilitated by AIBOMs.

7.2.3.5 *Singapore*

Singapore regulators released the **Model AI Governance Framework for Generative AI** (1), which enforces requirements related to generative AI, such as AI-generated watermarking [165]. This framework could be supported by AIBOMs.

7.2.3.6 *China*

China's **Measures for the Management of Generative Artificial Intelligence Services** (2) resembles an AIBOM-like requirement [81]. Accordingly, foundation models embedded in public-facing AI and AI-enabled systems must be registered with governments [81, 165]. China also has other regulations that could be facilitated through AIBOMs. These are **Personal Information Protection Law of the People's Republic of China** (1) and the **Basic Security Requirements for Generative Artificial Intelligence Services** (1).

Answer to RQ2

Standards like **ISO/IEC 42001**, **IEEE P70xx**, and the **NIST AI Risk Management Framework** promote responsible AI governance, ethics, and risk management—areas where AIBOMs contribute by documenting information related to AI components. In the *European Union*, the **EU AI Act** requires transparency, risk classification, and documentation for AI and AI-enabled systems. AIBOMs help meet these demands, as well as those under **GDPR**, **EMA Requirements for Medical Device**, and **Product Liability Directive**. In the *United States*, several executive orders (including **EO 13960**, **EO 14028**, and **EO 14110**) and laws (including **Copyright Law**, **Cyber Supply Chain Management and Transparency Act**, and **Internet of Things Cybersecurity Improvement Act**) emphasize AI transparency, security, responsibility, and accountability. AIBOMs are believed to guarantee these attributes.

7.2.4 *RQ3. What elements are included or worth including in AIBOMs?*

Based on the open coding, twenty-three elements that are included (or worth including) in AIBOMs were identified (see Table 7.7). These elements were organized into three broad categories based on whether they pertain to the models, datasets, or both. The following section elaborates on each of them.

7.2.4.1 *Models and Datasets*

Identifiers (19) stand out as the most common elements worth including in AIBOMs for both models and datasets. These elements can be used to represent both name and versions of models and datasets [14–16, 20, 48, 95, 105, 106, 113, 146, 159, 170, 176, 195, 196, 210–212]. Identifiers can be also used to indicate the name and path of the files with AI-generated code [175].

Other elements worth including in AIBOMs for both models and datasets are **licenses** (17). These elements provide a standardized way of communicating licensing information about

Table 7.7: Summary of codes for RQ3.

Axial Codes	Codes	#Total	#FL	#GL	FL Sources	GL Sources
Datasets	Characteristics	3	1	2	[170]	[20, 183]
	Processing	8	3	5	[14, 146, 170]	[16, 20, 113, 176, 183]
	Sensors	3	0	3		[20, 72, 183]
Models	Architecture	9	2	7	[146, 170]	[16, 20, 105, 113, 159, 208, 211]
	Authenticity	9	2	7	[106, 146]	[16, 72, 105, 113, 159, 175, 176]
	Autonomy	2	0	2		[20, 160]
	Explainability	2	0	2		[20, 160]
	Performance	4	0	4		[20, 130, 160, 211]
	Safety	2	0	2		[20, 160]
	Security	6	1	5	[146]	[48, 176, 183, 208, 210]
	Sustainability	9	1	8	[146]	[16, 20, 72, 105, 113, 159, 160, 211]
	Training	20	4	16	[146, 170, 195, 196]	[16, 20, 30, 92, 95, 105, 113, 129, 130, 160, 176, 183, 184, 208, 211, 212]
	Usage	13	2	11	[15, 146]	[16, 20, 30, 95, 105, 113, 159, 160, 176, 183, 211]
Models and Datasets	Authorship	13	3	10	[14, 15, 146]	[16, 20, 48, 95, 105, 113, 159, 175, 176, 211]
	Dependencies	13	1	12	[146]	[16, 48, 105, 113, 129, 130, 159, 176, 183, 208, 210, 212]
	Ethical Considerations	12	2	10	[146, 170]	[16, 20, 72, 105, 113, 159, 176, 210–212]
	Identifiers	19	7	12	[14, 15, 106, 146, 170, 195, 196]	[16, 20, 48, 95, 105, 113, 159, 175, 176, 210–212]
	Licenses	17	4	13	[15, 106, 146, 201]	[16, 20, 30, 48, 95, 113, 130, 159, 175, 176, 208, 210, 211]
	Limitations	3	0	3		[20, 160, 211]
	Privacy	2	0	2		[20, 160]
	Provenance	12	4	8	[14, 146, 170, 195]	[16, 20, 95, 105, 113, 130, 159, 212]
	Standards	2	0	2		[20, 160]
	Type	10	1	9	[146]	[16, 20, 105, 113, 130, 159, 160, 176, 183]

models and datasets, making it easier to comply with license obligations for AI and AI-enabled systems [16, 20]. Licenses can also be associated with AI-generated code [175, 201].

Authorship (13) facilitates tracking the origin of components by identifying their suppliers [20]. Examples of these elements

are the names of the individuals and/or organizations owning, curating, managing, or distributing a given model or dataset [16, 20, 113, 211]. In AI-generated code, authorship may relate to the developer who added such code [175].

Dependencies (13) may indicate any libraries that a given component relies upon [16, 146]. In the case of AIBOMs, dependencies may indicate the functions, tools, and datasets used to facilitate the development, training, and deployment of ML models [208].

Ethical considerations (12) outline how potential ethical concerns are addressed and mitigated, including privacy violations or racial bias [16, 20, 113]. These concerns can introduce bias in model predictions [20]. For datasets, ethical concerns may arise due to: (i) selection bias, where the collected data fails to represent the target population accurately; (ii) measurement bias, which occurs when the accuracy of the measurements varies; and (iii) label bias, when the labels used to categorize the data are biased in some way [20].

Other elements that help track the origin of components relate to **provenance** (12). They can link to the source of the training datasets or the place where the model can be accessed/downloaded, such as Hugging Face [16, 20, 113]. Provenance may also indicate if datasets pertain to customer, open-source, or synthetic data [130], as well as the data collection approach (*e.g.*, mining repositories or interviews) [20].

For any given model or dataset, AIBOMs can support the inclusion of elements describing its **type** (10). For models, this includes indicating whether the asset is, for example, a neural network, decision tree, text generation system, image processing model, or a learning paradigm such as supervised, unsupervised, or reinforcement learning—or a combination thereof [16, 20, 113, 160]. Additionally, the application domain (*e.g.*, computer vision, healthcare, finance, or automotive) can be specified to provide further information [20, 160]. For datasets, the type can denote whether the data is personal or sensitive [130], and may also include the variety of data formats involved, such as textual or images [20]. Furthermore, datasets may be characterized by their structure, distinguishing between unstructured content and

structured data typically organized in tabular formats or sourced from relational databases [20].

Limitations (3) can be explicitly documented in AIBOMs to highlight areas where a model or dataset may fall short [160]. For models, limitations may pertain to factors that impact performance negatively [211] or limited generalization to new/unseen data [20]. In the case of datasets, limitations might include issues such as class imbalance or overfitting due to insufficient data diversity [20].

AIBOMs could also report information related to **privacy** (2). This would refer to any sensitive personal data processed by a model or included in datasets [20, 160]. AIBOMs could describe the methods used to anonymize the dataset or fields in the dataset (*e.g.,* pseudonymization, k-anonymity, l-diversity, or t-closeness), the confidentiality level, and the presence of sensitive personal information [20].

AIBOMs could also report the **standards** (2) that a system adheres to [20, 160]. This could prove that the system meets published or unpublished standards, such as those developed by ISO, IEEE, and ETSI [20]. The standards may, but are not necessarily required to, satisfy legal or regulatory requirements [20].

7.2.4.2 *Models*

Training (20) stands out as the most common element worth including in AIBOMs to document the training phase of the model. Training information includes the techniques, algorithms, configurations, hardware, and software used to develop models, which are essential for understanding and reproducing results [16, 113, 160, 183]. Key among these are hyperparameters—predefined settings like learning rate, batch size, and number of layers—that significantly influence model behavior and performance [20, 30, 130, 160, 170, 176, 183, 184, 211]. Proper selection and tuning of these hyperparameters are crucial for effective learning, ultimately leading to better predictive performance and more reliable outcomes [20, 160]. Links to the datasets used to train, validate, and test the models can also be made explicit [20, 146, 170, 176, 196, 208, 211].

Other elements worth including in AIBOMs for models relate to the **usage** (13) and **architecture** (9). As for usage, the elements include the hardware and software required to run the model, the expected input and output, as well as intended, improper, or malicious uses [15, 16, 20, 30, 95, 105, 113, 146, 159, 176, 183]. As for architecture, the information of the elements are arranged into the following four groups:

(i) Model architecture [16, 105, 113, 159]. It provides details on the specific architecture of the model, including its structure, layers, nodes, activation functions, and any specialized components [20, 146, 211]. If applicable, it may also indicate the name of such architecture, *e.g.,* GPT-J, GPT-1, ResNet-50, or YOLOv3 [16, 211].

(ii) Architecture family [16, 105, 113, 159, 211]. It details the model architecture family, such as transformer network, convolutional neural network, residual neural network, or LSTM neural network [211]. If applicable, it may also indicate the name of the architecture family or class, *e.g.,* Llama 2 [16, 146].

(iii) Base model [16, 113, 146, 159, 170, 208]. It provides details on the name, version, and source of the base (or "foundation") model on which the current model is based, if any [113, 146]. A base model can act as a starting point for fine-tuning or transfer learning on specific tasks, thus it serves as a foundation for building more complex models [208].

(iv) Parent model [16, 105, 113, 146, 159]. It provides details on the name, version, and source of the parent model that the current model is based on, if any [113, 146]

Elements related to the **authenticity** (9) of a model encompass certifications, attestations, and verifications—digitally signed by the model creator—that establish the model's credibility and demonstrate compliance with relevant standards and regulations [146]. Such information can also help ensure the integrity of the model [16, 113, 176] and may be delivered through blockchain-based mechanisms, such as smart contracts [106]. In the case of

AI-generated code, authenticity may be expressed as a confidence level indicating the likelihood that the code was generated by an AI model [175].

Elements related to **sustainability** (9) include environmental impact [16, 20, 113, 159, 160] and, in particular, the energy consumption required to train, fine-tune, and operate AI models [16, 20, 72, 105, 113, 211]. For instance, large language models often require substantial computational resources; estimates suggest that a single large language model can release as many carbon emissions as five cars over their entire life-cycle [20]. This can lead to significant environmental impact, especially when training occurs in data centers powered by non-renewable energy sources [20]. This contributes to carbon footprint and carbon emissions (*e.g.,* release of carbon dioxide, also known as CO_2), with potential impacts on climate change [16, 20, 113]. Another relevant concern is using physical resources such as water for cooling data centers [20]. Energy consumption estimates can be based on various factors, including details about the computational resources used (*e.g.,* number of floating-point operations), training duration, type and quantity of processing units, and other relevant aspects of the training process [20]. Energy units may be specified as kilowatt-hours (kWh) or megajoules (MJ), as well as other alternative units [20]. For example, the training of a deep learning neural network for image classification has been reported to consume 980 kWh of energy; fine-tuning consumed an additional 15 kWh; and inference, when the model is deployed, was reported to consume approximately 80 watt-hours of energy [20].

Security (6) elements that can be reported in AIBOMs encompass vulnerabilities and other security requirements (*e.g.,* encryption methods, access controls, and governance measures) [146, 176, 208]. Information about vulnerabilities relevant to the AI and AI-enabled system (*e.g.,* LLM Vulnerabilities & Exposures) can be taken from databases such as the NVD and CVE, along with their severity ratings and potential impact [146, 208]. Metrics related to **performance** (4) can be reported in AIBOMs to document how the model was evaluated and to guide future improvements [20, 160]. Common metrics include accuracy, precision, recall, F1-score, ROC curve, AUC, MSE, MAE, RMSE,

R-squared, and relevant benchmarks [20]. AIBOMs may also document any trade-offs in accuracy or performance made during the model's construction, training, and application [160]. **Safety** (2) elements could cover the comprehensive assessment of safety risks associated with a given system [20, 160]. This includes identifying and integrating various risk levels and considerations to better understand and mitigate potential threats posed by each model [20, 160]. As in SPDX, these risk levels should align with established regulations, such as the EU General Risk Assessment Methodology, which supports the implementation of Article 20 of Regulation (EC) No 765/2008 and aids authorities in evaluating general system safety compliance [20, 160]. The standardized safety risk levels are defined as follows:

(i) Serious [20, 160]. Critical risk requiring immediate attention (*e.g.,* self-driving vehicles);

(ii) High [20, 160]. Significant risk necessitating stringent controls (*e.g.,* adaptive education systems);

(iii) Medium [20, 160]. Moderate risk that warrants balanced mitigation strategies (*e.g.,* bank loan approvals);

(iv) Low [20, 160]. Minimal risk with limited safety concerns (*e.g.,* shopping recommendation).

Software suppliers are expected to classify their application's safety risk level based on the intended use and target audience [20]. AIBOMs could also describe the level of decision-making **autonomy** (2). It relates to whether, and to what extent, the system can perform a decision or action without human involvement or guidance [20, 160]. In other words, it should indicate whether humans are involved "in the loop" to validate the decision-making process of the system with embedded AI [20]. Examples of systems where humans are not in the loop include self-driving cars, autonomous robots, and virtual assistants [20]. In contrast, systems that typically involve human oversight include mortgage loan approvals, financial investment decisions, and applications within the justice system [20]. **Explainability** (2) elements could indicate how well it is possible to understand how the model arrived at its predictions, decisions, or recommendations, and why it made those choices [20, 160]. These

elements may also report the different explainability mechanisms that can be used to explain the results from the AI model—*e.g.,* SHapley Additive exPlanations (SHAP) and Local Interpretable Model-Agnostic Explanations (LIME) [20]. The goal of model explainability is to make AI and AI-enabled systems more transparent, trustworthy, and accountable [20, 160].

7.2.4.3 *Datasets*

Processing (8) stands out as the most common element worth including in AIBOMs for datasets. It encompasses all procedures that led to creating a given dataset before being used [16, 20, 113]. These procedures include data collection, curation, cleaning, transformation, augmentation, sampling processes, and update [20, 146, 183]. Detailed information may include methods for tokenization, normalization, standardization, or deduplication [20, 183]. As for the update procedures, these can be batch, real-time (*e.g.,* stock prices), incremental (*e.g.,* social media or news feeds), or manual (*i.e.,* requiring human verification) [20].

For systems operating in physical domains, **sensors** (3) used for data collection can be reported in AIBOMs [20, 72, 183]. This might also involve calibration information and how the values are stored (*e.g.,* in a key-value format and the names of such keys) [20]. Other elements related to datasets that can be included in AIBOMs are the **characteristics** (3), such as the size (*e.g.,* in terms of bytes) and labeling [20, 170, 183].

Answer to RQ3

For both *models and datasets*, the analyzed sources recommend including **provenance** information, including **authorship**, **licensing**, or **dependencies**, as well as **limitations**, such as bias or privacy issues. For *datasets*, it is particularly important to detail information describing how the dataset has been **processed**. For *models*, relevant elements to include pertain to their **architecture**, **training**, and **usage**, as well as **authenticity**, **security**, and **sustainability** considerations.

Table 7.8: Summary of codes for RQ4.

Axial Codes	Codes	#Total	#FL	#GL	FL Sources	GL Sources
Compliance	Enforcing Intellectual Property	5	1	4	[196]	[20, 130, 176, 210]
	Enforcing Regulations	10	1	9	[146]	[16, 20, 95, 102, 122, 130, 176, 210, 212]
Dependability	Security-by-Design	9	3	6	[21, 146, 170]	[16, 20, 102, 113, 130, 176]
	Responding to Vulnerabilities	12	3	9	[21, 146, 189]	[20, 72, 95, 102, 113, 129, 130, 176, 210]
	Supporting Troubleshooting	3	1	2	[146]	[16, 159]
Management	Managing AIOps Pipelines	10	2	8	[146, 195]	[16, 72, 95, 130, 159, 183, 210, 212]
	Managing Communication	1	1	0	[107]	
	Managing Models	9	4	5	[15, 139, 146, 196]	[20, 72, 130, 176, 210]
	Managing Third-party AI Components	12	2	10	[139, 146]	[20, 30, 92, 113, 129, 130, 175, 176, 210, 212]

7.2.5 *RQ4. What are the use cases where AIBOMs are used?*

The open coding resulted in the identification of nine use cases for AIBOMs (see Table 7.8), which were organized into three broad categories based on whether they pertain to management, compliance, or dependability. The following section elaborates on each of them.

7.2.5.1 *Management*

Managing third-party AI components (12) is one of the most prominent and well-documented use cases for AIBOMs. The supply chain of AI and AI-enabled systems is inherently complex and involves several third-party components [146]. As more organizations seek to embed AI into their systems [92], the risks associated with third-party components become increasingly significant [130]. To mitigate these risks, it is essential to properly evaluate third-party open-source models and datasets before their adoption [130]. AIBOMs enhance transparency by offering visibility into the provenance and characteristics of these components, enabling proactive assessment and management of the associated risks [146, 212]. AIBOMs can be used in evaluation

and procurement processes to identify and select trustworthy assets [30, 129, 210]. For example, the US Army has planned the use of AIBOMs to evaluate AI models under consideration for military applications, to test, retrain, and further evolve these models as needed [92].

AIBOMs are also expected to play a pivotal role in **managing AIOps pipelines** (10) [159, 210]. For instance, they can help ensure that AI models are developed, deployed, and maintained in a responsible and transparent manner [159]. Similar to SBOMs, AIBOMs should be automatically generated and integrated into pipelines for auditing and compliance purposes [16, 183]. However, AIOps pipelines present unique characteristics, such as the need for continuous training to address data or concept drift [195].

In general, the use of AIBOMs supports **managing models** (9), helping to ensure the quality of the AI and AI-enabled systems in which such models are embedded [146]. For instance, AIBOMs facilitate model traceability, lifecycle management, and streamlined rollback or retraining processes [130]. By documenting specific versions, dependencies, and configurations of each component, AIBOMs also enable the identification and mitigation of potential risks [210].

AIBOMs could also facilitate **managing communication** (1) between stakeholders [107]. For example, the standardized reporting offered by AIBOMs could be used to inform stakeholders about the development and design process of AI and AI-enabled systems [107].

7.2.5.2 *Compliance*

AIBOMs can be used for **enforcing regulations** (9). They help ensure that AI models meet external regulatory requirements, comply with internal policies and standards, and address customer-specific compliance needs [16, 95, 130, 146, 210]. Relevant policies may pertain to privacy [210], ethics [146, 212], and legal frameworks [146]. AIBOMs can also support **enforcing intellectual property** (5). By documenting and scanning license information, AIBOMs help ensure that AI models comply with intellectual property rights and usage restrictions [130, 210]. This is valuable

for various stakeholders. For instance, compliance officers and auditors can verify that AI and AI-enabled systems adhere to licensing requirements, while legal counsels can review contracts, terms of service, and licensing agreements to ensure legal compliance and reduce potential legal risks related to third-party software components [210]. AIBOMs can also assist in managing intellectual property concerns related to AI-generated code by helping detect potential exposure of proprietary or sensitive information in generative AI inputs and outputs [176].

7.2.5.3 *Dependability*

AIBOMs can play a critical role in **responding to vulnerabilities** (12) by enabling their rapid identification and mitigation in case of cybersecurity incidents [21, 130]. By supporting continuous monitoring, AIBOMs help detect AI-specific vulnerabilities and adversarial threats—such as backdoors, model drift, or data poisoning attempts—that may compromise the integrity of AI and AI-enabled systems [129, 176]. For example, if a business analyst or data scientist discovers that a dataset has been poisoned, the AIBOM makes it easier to identify all models that interacted with the compromised data [95]. Another threat type, namely model evasion attacks, involves manipulating training data or exploiting weaknesses in the model to bypass security controls and generate false outputs [210]. With detailed insight into each component and dependency, development teams can quickly apply patches and update compromised components [189]. This proactive identification and response process is essential for preserving model dependability and maintaining the security of the broader systems in which AI is deployed [189].

In general, integrating AIBOMs into the software development lifecycle supports the implementation of **security-by-design** principles (9) [130]. As with traditional software systems, building AI and AI-enabled systems often involves assembling open-source and commercial components, which may introduce security flaws [21]. AIBOMs can function similarly to SBOMs by helping in managing security risks throughout the development process [21]. By providing detailed visibility into the components used, AIBOMs enable the selection of models that are better

tested, more secure, and pose lower risks [16]. They also support tracking external APIs and libraries, alerting stakeholders to known vulnerabilities [176]. Additionally, AIBOMs can flag whether a model was trained on potentially poisoned or compromised datasets [170].

Besides security, AIBOMs can help in **supporting troubleshooting** (3). AIBOMs facilitate the rapid identification of problematic components when model failure or bias occurs, enabling more effective troubleshooting and enhancing overall system dependability [146, 159]. For instance, AIBOMs can assist in debugging issues that arise from upstream model changes [16].

Answer to RQ4

AIBOMs are considered to play a pivotal role in ***managing* third-party AI components** by increasing transparency and facilitating risk evaluation during adoption and procurement. They are also relevant to help **managing AIOps pipelines** by enabling responsible, automated, and auditable model deployment. More generally, AIBOMs are relevant to help in **managing models** across their lifecycle. In terms of *compliance*, AIBOMs contribute to **enforcing regulations**, including internal policies and ethical/legal standards, and **intellectual property**, by tracking licensing terms and preventing unauthorized use. As for *dependability*, AIBOMs enhance **responding to vulnerabilities**, including adversarial attacks such as backdoors, model drift, or data poisoning. Using AIBOMs is also important to support **security-by-design** practices, enabling better-informed model choosing and vulnerability tracking throughout development.

7.2.6 *RQ5. What are the approaches, tool categories, and techniques to deal with AIBOMs?*

Sixteen approaches, tool categories, and techniques to deal with AIBOMs were identified and organized into six broad categories based on whether they pertain to AIBOM information sources,

Table 7.9: Summary of codes for RQ5.

Axial Codes	Codes	#Total	#FL	#GL	FL Sources	GL Sources
Standard	CycloneDX	10	1	9	[196]	[30, 48, 54, 95, 105, 122, 130, 159, 211]
	SPDX	10	1	9	[196]	[20, 30, 54, 105, 130, 159, 160, 176, 210]
AIBOM Information Sources	Model Cards	11	3	8	[139, 170, 196]	[16, 20, 54, 122, 159, 169, 208, 211]
	Datasheets	2	1	1	[196]	[20]
	Factsheets	1	0	1		[20]
	Interactive catalog	1	0	1		[176]
Generation	Automated	5	1	4	[170]	[16, 105, 176, 183]
	Manual	3	0	3		[16, 176, 183]
	AIBOM generator	1	0	1		[176]
	Quasi-AIBOM generator	1	1	0	[170]	
Format	JSON	3	0	3		[48, 95, 159]
	XML	1	0	1		[48]
Consumpion	AIBOM visualizer	2	1	1	[146]	[159]
	Query	1	1	0	[15]	
Framework	Blockchain	2	2	0	[106, 196]	
	Enterprise	1	0	1		[102]

standards, generation, format, consumption, or framework (see Table 7.9). The following section elaborates on each of them.

7.2.6.1 *Standards*

Two standards have emerged for structuring AIBOMs: **CycloneDX** (10) and **SPDX** (10). CycloneDX was extended in version 1.5 to support the documentation of AI-related components [95, 122, 159]. Similarly, SPDX introduced support for AI-related components in version 3.0 [20, 160]. Notably, the SPDX Working Group—operating under the Linux Foundation—is actively working to complement the existing SPDX ISO Standard (ISO/IEC 5962:2021), which defines the use of SBOMs to document the components involved in building a software system [159, 210].

7.2.6.2 *AIBOM Information Sources*

Model cards (11), introduced by Ezi Ozoani, Marissa Gerchick, and Margaret Mitchell in a 2022 blog post [159], may offer a promising foundation for constructing AIBOMs. Today, major AI ecosystems use model cards to describe models and datasets [16, 139, 170, 208]. In this respect, Hugging Face presents model cards that closely align with the goals of SBOMs [16, 211]. Model cards are considered a valuable documentation framework as they support understanding, sharing, and improving models and datasets. However, despite their potential, model cards have several critical limitations that prevent them from serving as full replacements for AIBOMs [16, 54]:

- Lack of standardization. There is no widely adopted schema defining the structure or minimum content of model cards, leading to inconsistency across models.
- Not machine-readable. Most model cards are written as unstructured text intended for human readers, rather than for automated processing.
- Limited automation. Due to their informal and inconsistent structure, model cards cannot be reliably generated, parsed, or integrated into automated pipelines.

In addition, model cards often omit critical metadata—such as the datasets used for training—undermining key AIBOM use cases like verifying licensing compliance, identifying security risks, and ensuring ethical sourcing [169]. Access restrictions further complicate the use of model cards for AIBOM generation. For example, accessing the full contents of Hugging Face repositories may require user authentication, acceptance of Terms of Service, or other legal agreements [169]. In practice, most model cards offer limited actionable information, which reduces their overall utility for AIBOM generation [169]. Nonetheless, existing tools for model card generation—such as those provided by Hugging Face and Google's Model Card Toolkit—could be extended or integrated to support AIBOM generation [16].

Other information sources that could be a starting point for generating AIBOMs could be **datasheets** (2), **factsheets** (1), and

interactive catalog (1). The latter may offer a more practical and user-friendly option to design AIBOMs, making it easy to navigate and manage components [176].

7.2.6.3 *Generation*

Ideally, a mature ecosystem of **automated** (5) AIBOM generators would enable software suppliers to generate AIBOMs at negligible cost [183]. However, achieving this level of capability would require substantial investment in research and development—estimates suggest that an investment of $2–4 million would be necessary to define, design, and prototype high-quality AIBOM tooling [183]. To date, tool support for generating AIBOMs remains scarce, if not nearly nonexistent [170]. In the short term, some data is likely to be populated in a **manual** (3) manner until AIBOM generators become more mature [16]. The cost of generating an AIBOM manually varies significantly depending on system complexity [183]. This cost is usually prohibitive for all but the simplest systems, due to the extensive expertise required to document all relevant components [183]. This challenge is further compounded by the fact that much of the necessary information may lie outside the deploying organization, particularly when third-party providers—such as pre-trained model vendors or dataset suppliers—are involved [183].

Only one **AIBOM generator** (1) was reported [176]. It is a commercial solution that claims to automatically scan AI ecosystems across cloud, on-premises, and third-party services, identifying and cataloging all AI assets, to generate AIBOMs as interactive catalogs. It is worth mentioning the possibility of leveraging also **quasi-AIBOM generators** (1), such as DVC and ML-Flow [170].

7.2.6.4 *Format*

AIBOMs are structured files that can be provided in various file formats. The file formats found to be reported for AIBOMs are **JSON** (3) and **XML** (1).

7.2.6.5 *Consumption*

Among the possible consumption tools—software that reads, interprets, and/or uses AIBOM—a **visualizer** (2) would enable exploring the content of an AIBOM through an interactive visual interface (*e.g.*, `https://aibomviz.aisecurityresearch.org`) [146]. Another way to consume AIBOMs would be to **query** (1) AIBOM data using technologies such as GraphQL [15]. Consumption tools would facilitate efficient access, management, and modification of the information contained in an AIBOM.

7.2.6.6 *Framework*

AIBOMs have been envisioned within frameworks. Considering the decentralized nature of the supply chain, **blockchain** (2) could serve as an infrastructure for managing AIBOMs among diverse stakeholders [106, 196]. For example, a three-tier architecture for blockchain-based AIBOM could comprise three layers: the service layer, which includes an AIBOM generator for automatically capturing metadata from artifacts; the on-chain AIBOM layer, which processes business logic via smart contracts; and the off-chain data layer, where data repositories are maintained by stakeholders themselves instead of being centralized in the platform [106].

AIBOMs could also be integrated in **enterprise** (1) architectures. For example, the Unified Architecture Framework [102], which is based on work in the defense domain, can support the systematic definition and organization of the components in AIBOMs, ensuring that all relevant elements are covered and properly documented through a common language and set of concepts.

Table 7.10: Summary of codes for RQ6.

Axial Codes	Codes	#Total	#FL	#GL	FL Sources	GL Sources
Consumption	Intellectual Property	4	1	3	[170]	[30, 105, 209]
	Security Auditing	2	0	2		[16, 183]
Generation	Accuracy	3	1	2	[170]	[30, 176]
	Available data	3	1	2	[201]	[16, 20]
	Capturing Dynamic Aspects	4	2	2	[195, 196]	[122, 183]
	Generators	7	3	4	[139, 170, 195]	[16, 20, 30, 122]
Human Aspects	Education	3	0	3		[105, 122, 183]
	Information Visualization	2	1	1	[14]	[16]
Interoperability	Interoperability with other BOMs	4	2	2	[170, 196]	[72, 183]
	Interoperability with other existing infrastructure	3	1	2	[170]	[72, 183]
Traceability	Predictions	2	1	1	[14]	[183]

Answer to RQ5

AIBOM files have generally **JSON** *formats*. Regarding *AIBOM information sources*, **model cards** are the most commonly referenced starting point for AIBOMs, as they help document AI models and datasets. However, model cards face critical limitations such as a lack of standardization, limited machine readability, and minimal automation. As for *standards*, **SPDX** and **CycloneDX** are emerging as crucial in supporting AI-specific extensions of SBOMs. For *generation*, there is currently limited tool support.

7.2.7 *RQ6. What are the limitations and challenges of AIBOMs?*

Twelve challenges and limitations related to AIBOMs were identified and organized into five broad categories based on whether they pertain to consumption, generation, human aspects, interoperability, or traceability (see Table 7.10). The following section elaborates on each of them.

7.2.7.1 *Generation*

The main limitation of current AIBOMs pertains to the lack of mature **generators** (7) [16, 30, 170, 195]. This highlights a significant market opportunity for developing AIBOM generators [16]. However, there are specific challenges related to the generation of AIBOMs. One is **capturing dynamic aspects** (4). While traditional systems remain static in their behavior once deployed, AI and AI-enabled systems might behave differently as they process new data [195, 196]. For example, an AI model could learn from such new data and make different predictions [195, 196]. In general, AIBOMs must deal with the co-evolution of data, models, code, and configurations [195]. Another challenge in AIBOM generation concerns the **available data** (3) used to generate AIBOMs. Data sources and repositories (*e.g.,* Hugging Face) often provide data that vary widely in both content and structure [16, 201]. When such data is not adequately structured, it must be parsed from unstructured text, an approach that makes the generation process difficult and can reduce the reliability of the extracted information [16]. In other words, a lack of standardization and the use of multiple sources can lead to **accuracy** (3) concerns when generating AIBOMs like false positives and false negatives [30].

7.2.7.2 *Consumption*

A key challenge in consuming AIBOMs involves concerns around **intellectual property** (4). Software suppliers may hesitate to adopt or fully populate AIBOMs due to fears of exposing information or losing intellectual property [30, 105, 170]. For instance, experts could reverse-engineer data from AIBOMs to compromise the vendor's intellectual property [209]. However, AIBOMs do not need to be public as they can be kept internal and shared selectively with verified stakeholders as needed [30]. Another possible challenge in AIBOM consumption relates to **security auditing** (2). The latter is fundamentally different for AI components compared to traditional software components, such as libraries [16]. Effective auditing would require a security database (similar to MITRE's CVE) tailored to AI-specific components [183]. Existing databases of AI vulnerabilities are inadequate for AIBOMs, as they often lack rigorous definitions

and function more as general collections of security and ethics incidents [183]. This new security database should define clear abstractions of AI vulnerabilities and enforce a machine-readable format, enabling such a database to serve as a reliable data source for AIBOM-based security audits [183].

7.2.7.3 *Interoperability*

AIBOMs represent a specialized type of BOM designed to describe the AI components of the software supply chain; consequently, challenges may arise around **interoperability with other BOMs** (4). Different types of software require distinct information to be adequately described [170]. For instance, AIBOMs might not encompass specialized hardware components, such as graphics processing units (GPUs), necessitating complementary documentation like Hardware Bills of Materials (HBOM) [183]. Furthermore, with the emergence of quantum machine learning, future AIBOMs may need to incorporate quantum computing resources and algorithms as this technology matures and gains traction in AI applications [72]. The distinction between AIBOMs and other BOMs extends beyond including additional fields: it also reflects the unique nature of the information they represent [196]. AIBOMs also face challenges related to **interoperability with existing infrastructure** (3). This encompasses integration with established AIOps pipelines across various stages, including data collection, data transformation, model creation, model configuration, and model transformation [170, 183].

7.2.7.4 *Human aspects*

Human-related challenges of AIBOMs are primarily associated with **education** (3). As AI remains an emerging technology, many in the industry lack a clear understanding of the purpose and requirements of AIBOMs, highlighting the need for targeted education and awareness [105, 122]. Even with the availability of mature, automated AIBOM tools, effective integration into AIOps pipelines would require training IT operations and cybersecurity personnel to use them appropriately [183]. Another possible human-related challenge involves **information visualization** (2). While visual representations such as tables and graphs can effec-

tively convey insights (*e.g.,* model performance or audit results), they are difficult to express in machine-readable formats like JSON [16]. Thus, providing a usable interface alongside structured data could significantly improve stakeholder engagement in AIBOMs [14].

7.2.7.5 *Traceability*

A key limitation of AIBOMs would be tracing **predictions** (2), particularly when attempting to attribute specific outputs to models within complex downstream components [183]. Despite this challenge, AIBOMs can support inspection and validation by enabling verifiable claims about contributions within the system [14]. Another possible challenge concerns tracking the **provenance** (1) of AI components throughout the supply chain. This includes model lineage, *i.e.,* the entire lineage from the current model to the foundational model to the data [16]. For example, the GPT4All-13B-Snoozy-SuperHOT-8K-GPTQ model is derived from Meta's 13B Llama, fine-tuned with GPT4All's Snoozy dataset using Low Rank Adaptation (LoRA), enhanced with SuperHot adapters for improved context reading, and then quantized for efficient storage and inference [16]. Provenance tracking is further complicated by inconsistent naming conventions [16]. The absence of standardized, unique identifiers remains a longstanding issue in traditional software and extends to AI components because it hinders the identification of components affected by a given issue [16].

Answer to RQ6

Despite the market opportunity, challenges and limitations related to *generation* primarily stem from the lack of mature AIBOM **generators**. Key issues include the difficulty of **capturing the dynamic aspects** of AI and AI-enabled systems (*e.g.,* evolution of models) and having **available data**. Even when available, data could be unstructured or inconsistent across various sources. *Consumption* challenges and limitations involve **intellectual property** concerns (*e.g.,* disclosing sensitive information). *Interoperability* presents challenges and limitations both in aligning AIBOMs **with other BOMs** (*e.g.,* HBOM) and **with other existing infrastructure** (whose integrating might add further complexity). Challenges and limitations related to *human aspects* include the need for better **education** around AIBOM.

7.3 DISCUSSION

A prominent theme across both FL and GL is the consensus on AIBOMs as a critical mechanism for documenting the components involved in the supply chain of AI and AI-enabled systems—ranging from models and datasets to training procedures and associated risks. The MLR revealed that AIBOMs are not merely conceptual artifacts but are increasingly supported by concrete use cases, particularly in high-stakes domains such as healthcare, defense, and critical infrastructure. Furthermore, the alignment of AIBOMs with emerging regulations (*e.g.,* EU AI Act, US Executive Orders, ISO/IEC 42001) underscores their regulatory utility. For example, AIBOMs help address accountability in AI-enabled systems, where responsibility is often diffused across multiple stakeholders in the AI supply chain (*e.g.,* data scientists, model trainers, infrastructure providers). By providing a structured and traceable record of data, models, and decisions—including AI-specific concerns like data lineage, model provenance, and ethical or environmental considerations—AIBOMs enable clearer attribution of responsibility, particularly in cases where systems cause harm or make erroneous decisions.

However, while motivations and benefits are well-articulated, practical adoption is hindered by substantial challenges. Chief among these are the insufficient and fragmented automation support for AIBOMs generation and consumption. Despite growing interest in AIBOMs, the MLR reveals a significant lack of mature, publicly accessible tools. Currently, only one proprietary tool claims to support AIBOM generation. Overall, AIBOM generation remains largely manual, fragmented, and prone to errors, which limits its scalability in practice. On the consumption side, there are no dedicated tools to support key use cases such as AI component management, compliance verification, or security assessment, which further hinders practical AIBOM adoption. Compounding these challenges is the lack of adequate underlying data needed to generate AIBOMs in the first place. While information sources like model cards and data sheets are discussed as starting points, they are not yet widely adapted or integrated into AI development life-cycles. For example, these information sources only offer partial support for AIBOM generation and lack both machine-readability and standardization, which hinders the extraction and synthesis of relevant data to include in AIBOMs.

Another important insight is the socio-technical nature of AIBOMs. Beyond technical traceability and security, they are positioned as enablers of ethical AI practices and responsible innovation. This multidimensionality introduces both opportunities and tensions. For example, the same documentation needed for reproducibility can raise concerns about intellectual property.

Below, the implications of this MLR for software suppliers, choosers, operators, and researchers are summarized.

SOFTWARE SUPPLIERS AIBOMs offer software suppliers several key advantages in managing the complexity of AI and AI-enabled systems. One of the most significant benefits is the improvement of transparency and traceability, which are critical for addressing security and safety concerns. These attributes enable software suppliers to identify vulnerabilities, manage risks, and ensure that AI components are adequately documented throughout the development lifecycle.

From a regulatory compliance perspective, AIBOMs play a crucial role in helping software suppliers meet legal and industry standards. Standards such as ISO/IEC 42001, IEEE P70xx, and the NIST AI Risk Management Framework emphasize responsible AI governance, and AIBOMs assist in documenting AI components to meet these requirements.

SOFTWARE CHOOSERS AIBOMs can effectively support software choosers in selecting AI components that align not only with technical requirements but also with broader organizational, ethical, and regulatory priorities. By offering structured documentation on key attributes—such as ethical trade-offs, model limitations, licensing, and sustainability—AIBOMs make it possible to evaluate AI components based on transparency, traceability, and trustworthiness. This level of visibility is particularly important in contexts where security, safety, and compliance are critical. For example, provenance information such as authorship, dependencies, and known bias or privacy concerns enables software choosers to avoid problematic components early in the selection process. Similarly, for datasets, details on processing history, and for models, insights into architecture, training, and intended use, allow for more comprehensive evaluation and better-informed decisions.

As global and regional regulatory frameworks evolve—such as the EU AI Act—AIBOMs provide a mechanism to ensure selected components meet increasing demands for accountability, risk classification, and documentation. This helps organizations reduce the risk of non-compliance and positions them to respond more effectively to internal audits and external disclosure requirements.

AIBOMs are especially valuable when evaluating third-party AI components, where visibility is often limited. By increasing transparency during procurement, AIBOMs help assess security risks, intellectual property constraints, and potential integration issues—key considerations when choosing components for regulated or mission-critical systems.

Software choosers should select systems with AIBOMs to ensure clear accountability across the supply chain, enabling trace-

ability and responsibility attribution in the event of failures, harm, or ethical concerns.

SOFTWARE OPERATORS Integrating AIBOMs into development pipelines can significantly enhance resilience and reduce incident response time, particularly in regulated sectors. By providing greater transparency and traceability, AIBOMs empower software operators to better identify and mitigate risks such as vulnerabilities, dataset contamination, and licensing issues in AI components. These attributes are crucial for ensuring security and safety, especially in high-stakes environments where rapid response to incidents is essential.

For software operators, AIBOMs also play a critical role in ensuring regulatory compliance. As regulations like the EU AI Act push for greater accountability and transparency, AIBOMs provide an effective mechanism for managing the risks associated with AI components. This includes support for internal audits and ensuring that systems comply with both legal and industry standards.

More generally, AIBOMs help operators manage third-party AI components and AIOps pipelines more effectively, enabling responsible, automated, and auditable model deployment—a crucial capability for maintaining long-term system performance and compliance. AIBOMs also contribute to security-by-design practices, supporting better-informed decisions when monitoring vulnerabilities throughout their lifecycle. This includes improving the ability to respond to adversarial attacks, model drift, and data poisoning.

RESEARCHERS AIBOM research presents both clear gaps and emerging opportunities that warrant further investigation. One promising direction is the empirical evaluation of AIBOM solutions. Currently, there is no concrete evidence regarding their effectiveness in real-world software development scenarios. Empirical studies should assess whether and how AIBOMs enhance trust, traceability, and regulatory compliance in practice. Comparative analyses across domains—such as healthcare and defense—could reveal domain-specific requirements, constraints, and adoption barriers.

AIBOMs hold potential for operationalizing AI ethics principles, including fairness, privacy, bias mitigation, transparency, and responsibility. Further research should investigate how AIBOMs can support accountability through both design-time and run-time documentation. In this regard, AIBOMs may serve as transparency artifacts that provide stakeholders with interpretable and verifiable evidence of responsible AI practices.

A critical challenge is the lack of automation in the generation and consumption of AIBOMs. There is a pressing need for tools and methods that can automatically extract relevant metadata from AI development artifacts—such as model training logs, dataset lineage, pipeline configurations, and source code repositories. Future work should explore techniques like information retrieval, provenance tracking, and static/dynamic analysis to facilitate automated AIBOM generation.

Existing documentation formats—such as model cards (*e.g.*, Hugging Face) and datasheets—offer only partial coverage. They often lack machine readability, semantic standardization, and integration into broader development workflows. Researchers should investigate how these artifacts can be extended or harmonized within the AIBOM framework to improve their utility and interoperability.

Finally, a promising direction for future work would be to replicate and extend this systematic review with a dedicated focus on FL. As the body of peer-reviewed research on AIBOMs continues to grow, a follow-up review centered exclusively on FL could provide additional insights and strengthen the evidence base. The search string used in this MLR and publicly available replication package [9] offer a solid foundation and aid for such an effort, enabling researchers to build on the presented methodology. This would also capture developments that may not have been present or prominent at the time of the original review.

7.3.1 *Threats to Validity*

This section discusses potential threats to the validity of the results. According to the validity schema by Wohlin *et al.* [194], these threats can be categorized as construct, reliability, external, and internal validity. Threats to internal validity are not discussed

because such threats concern causal relationships [194], and causality was not investigated in this MLR.

Threats to *construct validity* could arise from the formulation of the RQs and the design of the data extraction. To mitigate these threats, the GQM methodology [17] was followed, as recommended by Garousi *et al.* [69], to ensure alignment and traceability of the study's objectives, RQs, and the data extraction form. Each author independently reviewed the RQs and data extraction form, with discrepancies resolved through collaborative discussions. Other potential threats to construct validity may relate to inaccurate data extraction, incomplete coverage of relevant studies due to limitations in search string, the use of search engines, and potential researcher bias in applying inclusion, exclusion, or quality criteria (*e.g.,* [68, 70, 71]). In constructing the search string, both the most common terms (*e.g.,* AIBOM) and similar terms used by different authors to refer to related concepts (*e.g.,* DataBOM) were included. For the choice of search engines, Scopus, IEEE Xplore, and the ACM Digital Library were selected for the FL, as they are considered among the top computer science data sources [135] and are recommended by the guidelines [69, 98]. Google Search Engine was chosen for the GL, in line with guidelines recommending its use for including GL in systematic reviews [4, 69]. To further reduce the risk of missing relevant FL and GL sources, backward and forward snowballing were employed following the related guidelines [69, 193]. Another potential construct validity threat might arise from the lack of empirical rigor in information based on practical experience or subjective opinion. While this aligns with the broader aims of MLRs, which intentionally integrate both FL and GL, it is worth acknowledging potential biases introduced by the heavy reliance on GL sources. These may include the risk of overstating benefits, presenting overly optimistic claims without empirical support, promotional- and marketing-driven content, and a lack of negative views on AIBOMs. Although the guidelines by Garousi *et al.* [69] were followed—which include quality assessment criteria targeting subjectivity and vested interests—some bias may persist despite these mitigation efforts. Indeed, no sources that explicitly presented negative views on AIBOMs were found, such as dissenting or critical perspectives regarding their feasibility.

Potential threats to *reliability validity* regard the application of inclusion, exclusion, and quality criteria. These were mitigated by having two individuals independently apply the criteria across all sources. To evaluate the inter-rater agreement, the Cohen's κ coefficient was calculated, obtaining an almost perfect agreement. In detail, Cohen's κ was 0.82 for the selection of FL, 0.89 for GL, and 0.87 overall. These values indicate strong consistency between raters and support the reliability of the source selection process. In case of disagreement, the final decision on including a given source was made through a discussion between the two authors. All disagreements were successfully resolved between the two authors who applied the inclusion, exclusion, and quality criteria. Finally, having data synthesis conducted by a single researcher may introduce a risk of subjective interpretation.

As for the threats to *external validity*, the MLR was conducted using the most popular and recommended data sources for both FL [69, 98, 135] and GL [4, 69]. Additionally, snowballing was performed following the related guidelines [69, 193] to identify further relevant studies. Nevertheless, we cannot claim that all possible FL and GL sources were examined. For example, some potentially relevant sources may not have been properly indexed or may not be freely available. Another potential threat to generalizability stems from the inclusion and exclusion criteria. For instance, only sources written in English were included; therefore, studies written in other languages were excluded. In this respect, it is argued that the final set of included sources contained a sufficiently comprehensive and diverse representation of both academic and practical perspectives. Finally, the findings based on very few sources (*e.g.*, explainability for RQ1) may have limited generalizability beyond the specific contexts in which they were reported (*e.g.*, they may reflect isolated perspectives, early-stage developments, or context-specific practices rather than broadly established trends).

DATA AVAILABILITY

The replication package of the study is available online [9].

8

ALOHA: A(IBOM) TOOL GENERATOR FOR HUGGING FACE

This chapter is based on the paper entitled "ALOHA: A(IBoM) tooL generatOr for Hugging fAce" [51], published in the proceedings of the 29th International Conference on Evaluation and Assessment in Software Engineering.

While there exist SBOM generators, AIBOM generation is not supported by suitable, automated tools. To address this gap, this chapter presents ALOHA (AIBOM tooL generatOr for Hugging fAce), a tool that supports the automated generation of AIBOMs for AI models hosted on Hugging Face. ALOHA mines and analyzes the content of Hugging Face model cards, extracting key pieces of information, including the license, the AI model's dependencies, the availability of a training dataset, and documentation on bias/fairness. A preliminary evaluation of 312 Hugging Face models indicates that the quality of the generated AIBOMs is inherently dependent on the information provided by the authors of AI models. Since ALOHA extracts data directly from model cards, well-documented and comprehensive AI model and dataset descriptions enable the creation of more accurate and informative AIBOMs, underscoring the value of thorough and consistent documentation.

CHAPTER STRUCTURE. Section 8.1 presents ALOHA. The preliminary empirical evaluation is reported in Section 8.2, while an overall discussion concludes the chapter in Section 8.3.

8.1 ALOHA

This section outlines the overall workflow of ALOHA, demonstrates its usage, and discusses its limitations.

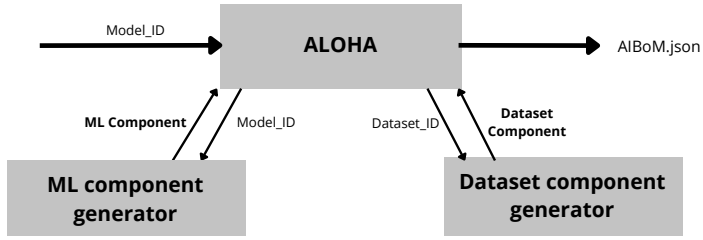


Figure 8.1: Overview of the ALOHA tool workflow

8.1.1 *Workflow*

ALOHA—developed in Python 3.10.2—takes as input a string containing the model ID from Hugging Face in the format *author/model*, producing an AIBOM in JSON format compliant with the CycloneDX standard.

The overall workflow of ALOHA is illustrated in Figure 8.1. ALOHA first invokes the *ML Component Generator*, which retrieves model-related information and generates the Hugging Face component following the CycloneDX standard. Specifically, ALOHA performs the following steps to generate AIBOMs from Hugging Face:

- It performs a request to the Hugging Face API to obtain the model metadata and the `README.md` file, which contains the *Model Description* and *Uses* sections from the text description of the model card.
- The extracted information is then mapped to the corresponding CycloneDX fields, forming the model component.

Next, ALOHA checks the model metadata to determine whether any datasets used for training or testing are specified. If datasets are present, the *Dataset Component Generator* is invoked for each dataset. This component:

- Performs a request to the Hugging Face API to obtain dataset metadata.
- Extracts relevant information and maps it to the CycloneDX data fields, generating the dataset component.

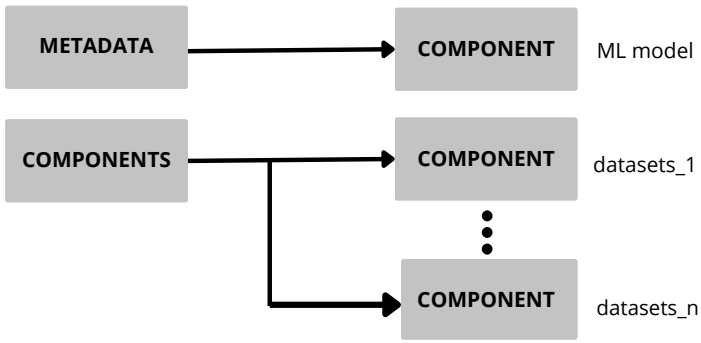


Figure 8.2: AIBOM Structure

Finally, the model component and all dataset components are added to the base BOM structure, producing the AIBOM in JSON format, fully compliant with the CycloneDX standard. The final structure of the generated AIBOM is shown in Figure 8.2.

8.1.1.1 *Mapping Hugging Face Model Card Sections to CycloneDX*

A key challenge in creating AIBOMs for models hosted on Hugging Face is the mapping of model card fields to their respective CycloneDX fields. The model cards are saved in the `README.md` file of each model repository and have two key parts, with overlapping information: *metadata* and *text description*. The metadata section is represented by a YAML block located at the top of the model card. It contains structured metadata about the model, such as tags, tasks, and licenses. The text description section is written in markdown and provides a more detailed textual description of the model, including its purpose, performance, and limitations. Hugging Face provides templates for both metadata and textual descriptions. These templates allow straightforward mapping of model card fields to CycloneDX data fields. However, few models adhere to the text description template, making data extraction challenging. Some models introduce minor modifications to section titles, while others adopt a completely different structure. To address this issue, we analyzed the model cards of the 10,000 most downloaded models to identify the possible aliases of the section titles of interest. Since this analysis required

Table 8.1: Mapping between CycloneDX and Hugging Face for the component data field in metadata.

CycloneDX	Hugging Face
authors	author
tags	tags
description	# Model Description
name	modelId
id	cardData-> license
name	cardData-> license_name
task	pipeline_tag
architectureFamily	config-> model_type
modelArchitecture	config-> architectures
datasets	datasets
slice	dataset-> type, config, split
type	metric-> type
value	metric-> value
useCases	# Uses
environmentalConsiderations	emissions
environmentalConsiderations	source
environmentalConsiderations	training_type
environmentalConsiderations	geographical_location
environmentalConsiderations	hardware_used
properties	library_name
properties	base_model
properties	base_model_relation

human intervention, we considered only the titles with at least 20 occurrences.

Once the model information is retrieved, if datasets are specified, their metadata is extracted, and the relevant details are mapped into the AIBOM. The model's information is stored within the component field of metadata and the dataset information within the component field of components. Figure 8.2 shows the final structure of the AIBOM. The final mapping is presented through Table 8.1 and Table 8.2, where one column lists the CycloneDX fields and the other shows the corresponding fields extracted from Hugging Face.

Table 8.2: Mapping between CycloneDX and Hugging Face for the component data field in components.

CycloneDX	Hugging Face
contents-> URL	URL of the Model
contents-> properties	task_categories
contents-> properties	task_ids
contents-> properties	language
contents-> properties	language_details
contents-> properties	size_categories
contents-> properties	annotations_creators
contents-> properties	language_creators
contents-> properties	pretty_name
contents-> properties	source_datasets
contents-> properties	paperswithcode_id
contents-> properties	config_name
contents-> properties	data_files-> split
contents-> properties	data_files->path
contents-> properties	license
contents-> properties	license_name
contents-> properties	license_link
contents-> properties	license_details
description	description
owners-> organization-> name	author
owners-> organization-> url	URL to author page

8.1.2 *ALOHA in Action*

ALOHA can be executed via the command line using the following general syntax:

```
python ALOHA.py <model_ID> -o <output_dir_path>
```

- `<model_ID>`: It indicates the model ID from Hugging Face in the format *author/model*. This is a required parameter and must correspond to a valid model identifier.
- `<output_dir_path>`: It indicates the destination directory where the output files will be saved. This is an optional

```
{
  "bomFormat": "CycloneDX",
  "specVersion": "1.6",
  "serialNumber": "urn:uuid:47a09005-e737-471b-8ab0-d65461fbd675",
  "version": 1,
  "metadata": {
    "timestamp": "2025-02-28T12:11:23.832568+00:00",
    "component": {
      "type": "machine-learning-model",
      "bom-ref": "1bitLLM/bitnet_b1_58-3B-9f518091-90ea-528e-9efe-15e2f9373a8",
      "name": "1bitLLM/bitnet_b1_58-3B",
      "licenses": [
        {
          "license": {
            "id": "MIT",
            "url": "https://spdx.org/licenses/MIT.html"
          }
        }
      ],
      "externalReferences": [
        {
          "url": "https://huggingface.co/1bitLLM/bitnet_b1_58-3B",
          "type": "documentation"
        }
      ],
      "modelCard": {
        "modelParameters": {
          "task": "text-generation",
          "architectureFamily": "llama",
          "modelArchitecture": "BitnetForCausalLM",
          "datasets": []
        }
      },
      "properties": [
        {
          "name": "library_name",
          "value": "transformers"
        }
      ]
    },
    "authors": [
      {
        "name": "1bitLLM"
      }
    ]
  }
}
```


Figure 8.3: Example of an AIBOM for `bitnet_b1_58-3B` generated using ALOHA.

parameter; if not provided, the tool will use a default directory.

Figure 8.3 shows an example of an AIBOM for the Hugging Face model `bitnet_b1_58-3B` generated by ALOHA. The AIBOM follows the CycloneDX standard, starting with the fields `bomFormat` and `specVersion`, which indicate the format and the specific version of the CycloneDX. The BOM element has two key properties: `serialNumber` and `version`, which together define the unique identity of the AIBOM. While the serial number acts as a unique identifier, the version helps keep track of potential updates or changes to the BOM. Within the `metadata` field, there

is the `timestamp` representing the exact moment of the AIBOM's creation, along with a `component` object that provides essential information about the model described. Each component includes a field called `bom-ref`, a unique reference to connect the component to other elements in the BOM. The `external references` field provides additional relevant information not directly included in the BOM itself. In this case, it links to the official documentation of the model on Hugging Face, offering a direct source for further details. The `modelcard` section aims to describe specific aspects of the model, such as its `task`, `architecture`, and `datasets` used for training and testing. However, in this example, the model's author did not provide these details. Lastly, the `properties` section includes additional metadata that does not directly match the CycloneDX standard (*e.g.,* information about the used library).

8.1.3 *ALOHA Limitations*

The "quality" of the AIBOMs generated by ALOHA is inherently dependent on the information provided by the authors of Hugging Face models. Since ALOHA extracts data directly from model cards, any inconsistencies, omissions, or inaccuracies in the original documentation will be reflected in the resulting AIBOMs. For example, if authors do not provide detailed descriptions of datasets, architectures, or performance metrics, the AIBOM will lack this critical information. This reliance on user-generated content introduces variability in the quality of the AIBOMs, as not all authors follow the same documentation standards or provide the same level of detail. Consequently, the tool's output is only as reliable as the input it receives, which may limit its effectiveness in generating comprehensive and accurate AIBOMs.

Another limitation is that ALOHA was developed specifically for AI models hosted on Hugging Face, which means its applicability to other platforms or ecosystems may be limited. Other platforms may have different standards or templates for model documentation, which could make it difficult to directly apply the tool without significant modifications.

8.2 PRELIMINARY EMPIRICAL EVALUATION

This section presents a preliminary empirical evaluation of ALOHA.

8.2.1 *Study Design*

The *goal* of this preliminary empirical evaluation is to study ALOHA with respect to the generated AIBOMs for the *purpose* of assessing their completeness in terms of data fields. The *perspective* is that of researchers interested in the supply chain of AI and AI-enabled systems. The *context* consists of AI models publicly hosted on Hugging Face.

Based on the aforementioned goal, the following high-level RQ was formulated and studied:

RQ. *To what extent the AIBOMs generated by ALOHA report all the expected data fields?*

This RQ aims at assessing whether the AIBOMs generated by ALOHA report all the expected data fields and, if not, why. This is because AIBOMs generated through ALOHA may not report information for each data field in case such information is lacking on Hugging Face.

The Hugging Face API Endpoints [61] and the associated Python wrapper, namely `huggingface_hub` [62], were leveraged to search for the models for which to generate the AIBOMs. Hugging Face was queried to retrieve the list of all hosted models, along with information about their number of downloads and likes. To avoid the risk of selecting personal or inadequate models, models with at least 100 downloads and at least 100 likes on Hugging Face were retrieved. These inclusion criteria are commonly used to mitigate the perils of mining software repositories [93, 139]. Eventually, a total of 1,643 models was retrieved, which represent the study population. Due to the large size of this population, a statistically significant random sample with a confidence level equal to 95% and a margin of error equal to 5% was built. The size of this sample was equal to 312.

Table 8.3: Number and percentage of documented and missing data fields in the generated AIBOMs.

Case	Documented Field		Missing Field	
	Sum	**%**	**Sum**	**%**
authors	312	100	0	0
tags	312	100	0	0
description	134	42.95	178	57.05
license	284	91.03	28	8.97
task	296	94.87	16	5.13
architectureFamily	217	69.55	95	30.45
modelArchitecture	213	68.27	99	31.73
datasets	88	28.21	224	71.79
performanceMetrics	41	13.14	271	86.86
useCases	77	24.68	235	75.32
environmentalConsiderations	2	0.64	310	99.36
library_name	298	95.51	14	4.49
Total	*2,274*	*60.74*	*1,470*	*39.26*

8.2.2 *Evaluation Results*

Table 8.3 shows the number and percentage of data fields that were reported (documented) and not reported (missing) in the AIBOMs generated from the models of the sample. The data fields `authors` and `tags` are always documented, while `license`, `task`, and `library_name` appear in over 90% of the AIBOMs. Following, `architectureFamily` and `modelArchitecture` appear almost in 70% of the AIBOMs, while `description` in about 40% of them. The data fields `datasets`, `performanceMetrics`, and `useCases` appear in between 10% and 30% of the AIBOMs. Lastly, `environmentalConsiderations` appears in less than 1% of AIBOMs.

These results indicate that the data fields that are mostly reported in the generated AIBOMs refer to the information of the model card that can be (i) reported by the author of the model or (ii) inferred directly by Hugging Face based on the information available in manifest/configuration files. The data fields that can be inferred by Hugging Face are `authors` [61], `tags` [63], `library_name` [64],

`task` [65], `architectureFamily` [66], `modelArchitecture` [66], and `environmentalConsiderations` [60]. These data fields are among the most documented in the generated AIBOMs, except for `environmentalConsiderations`, which is the least documented data field. In fact, it might be inferred only for the models using the AutoTrain feature [59], but only one model in the sample used it (although there is no `environmentalConsiderations` for that model).

A qualitative analysis of the missing data fields in the generated AIBOMs was conducted to better understand the reasons why they were not reported. This analysis involved a manual inspection of the content of the model cards on Hugging Face and open coding to categorize the causes of missing data fields. That is, for each missing data field, a "code" explaining the reason why that data field was missing was assigned. The causes of missing data fields were eventually classified with three codes:

- *Code 01 - Missing Information*. Information for the data field is not available anywhere in the model card.
- *Code 02 - Misplaced Information*. Information for the data field is available in a section of the model card that is not mapped to that data field.
- *Code 03 - Restricted Access to Information*. Direct access to the model card on Hugging Face, which is required to extract the information for generating the AIBOMs, is not allowed, as there are conditions to be accepted.

Table 8.4 shows the classification of missing data fields based on the aforementioned codes. The results indicate that the data field `library_name` is always missing due to Code 01, while `task`, `architectureFamily`, `environmentalConsiderations` are missing in over 90% of the AIBOMs due to the same code. Following, `modelArchitecture`, `datasets`, and `useCases` are missing between 80% and 90% of the AIBOMs due to Code 01. Due to the same code, the data fields `license` and `performanceMetrics` are missing between 60% and 70% of the AIBOMs, while `description` is missing in about 26% of them. As for Code 02, the data field `description` is missing in about 70% of the AIBOMs, while `license` and `performanceMetrics` are missing between

Table 8.4: Number and percentage of missing data fields in the generated AIBOMs grouped by codes.

Case	Code 01		Code 02		Code 03	
	Sum	%	Sum	%	Sum	%
description	46	25.84	122	68.54	10	5.62
license	19	67.86	9	32.14	0	0
task	15	93.75	1	6.25	0	0
architectureFamily	89	93.68	6	6.32	0	0
modelArchitecture	89	89.9	10	10.1	0	0
datasets	194	86.61	30	13.39	0	0
performanceMetrics	170	62.73	101	37.27	0	0
useCases	200	85.11	33	14.04	2	0.85
environmentalConsiderations	305	98.39	5	1.61	0	0
library_name	14	100	0	0	0	0
Total	*1,141*	*77.62*	*317*	*21.56*	*12*	*0.82*

30% and 40% of them. Following, `modelArchitecture`, `datasets`, and `useCases` are missing between 10% and 20% of the AIBOMs due to Code 02. Due to the same code, all other data fields are missing in less than 10% of the generated AIBOMs. Only two data fields are missing due to Code 03, namely `description` and `useCases`, which happened in about 6% and 1% of the generated AIBOMs, respectively. Summing up, about 77.62% of the data fields are missing due to Code 01, 21.56% due to Code 02, and 0.82% due to Code 03. ALOHA was not able to report the expected data fields of all the generated AIBOMs because the needed information was not reported in the corresponding model cards.

The results of the investigation indicate that the missing data fields in the generated AIBOMs are mainly due to a lack of the associated information in the model cards. On the other hand, a portion of the data fields is missing because the associated information is documented in unexpected sections of the model cards; therefore, their model cards do not follow the template recommended by Hugging Face, and no mapping with those sections has been identified. Lastly, some data fields in the generated AIBOMs were missing due to restrictions in accessing the associated information of the model cards; in these cases, the

authors of the models limited access upon acceptance of certain conditions (*e.g.,* sharing of contact information by those who want to access the information).

8.2.3 *Threats to Validity*

As for threats to *construct validity*, ALOHA was not evaluated with respect to generating subfields of AIBOMs. That is, given a data field (*e.g.,* `environmentalConsiderations`) consisting of several subfields (*e.g.,* `emissions`, `source`, `training_type`, `geographical_location`, and `hardware_used`), ALOHA was considered to have generated that data field whether it contained information for at least one of its subfields. This threat to validity is not a concern given the nature of the investigation, *i.e.,* a preliminary empirical investigation.

As for threats to *conclusion validity*, a statistically significant random sample of Hugging Face models with at least 100 downloads and at least 100 likes—using a 95% confidence level and a 5% margin of error—was built. The analysis of samples is common in empirical research to avoid analyzing the entire population of interest (*e.g.,* [44]). However, it might pose a threat to conclusion validity because the randomness of the sample selection process does not rule out selection bias, *i.e.,* the selected AI models may not fully represent the population under investigation.

As for threats to *external validity*, the results might not be generalizable to the generation of AIBOMs from AI models hosted on platforms other than Hugging Face. However, this is beyond the current features of ALOHA. Also, the results reflect the state of documentation of the model cards on Hugging Face at the time the investigation was conducted (*i.e.,* February 2025). Replications of this investigation at a later time might lead to observing different results, for example, due to changes to the model cards or the hosting of new AI models on Hugging Face. Lastly, considering AI models that are popular on Hugging Face might not fully represent documentation practices across all of Hugging Face.

8.3 DISCUSSION

The increasing complexity of AI-driven software necessitates robust mechanisms for ensuring transparency, traceability, and security. ALOHA represents a considerable step forward by enabling the automated generation of AIBOM, leveraging the widely adopted CycloneDX standard.

The preliminary evaluation demonstrated that ALOHA is capable of extracting and organizing relevant data fields from Hugging Face model cards, including both metadata and free-text descriptions. However, inconsistencies in documentation and missing metadata remain significant challenges. In other words, the "quality" of the AIBOMs generated by ALOHA is directly influenced by the information provided by the owners of the AI models. Since ALOHA pulls data directly from the model cards, any discrepancies, omissions, or inaccuracies in the original documentation are reflected in the resulting AIBOMs. This underscores the critical role of high-quality, detailed inputs from AI model authors, as they directly impact the completeness and accuracy of the resulting AIBOMs. As documentation practices continue to improve and collaboration with the community strengthens, ALOHA is well-positioned to produce increasingly reliable and informative AIBOMs.

DATA AND TOOL AVAILABILITY

ALOHA is available open-source under the Mozilla Public License 2.0 License on Zenodo [40] and GitHub [41]. The replication package of the preliminary empirical evaluation is also available online [39].

Part IV

FINAL REMARKS

9 OVERALL DISCUSSION

This chapter links the results from the single studies to the objectives of the thesis and subsequently outlines the implications for researchers, software suppliers, software consumers, and software operators.

9.1 LINKING RESULTS AND OBJECTIVES

In the following, for each objective, the main results are summarized.

9.1.1 *O1. Understanding the current state of practice of SBOM and AIBOM to support the management of the supply chain of software and AI systems*

The results for O1 pertain to knowledge-seeking empirical research [172]. These are presented as key findings in framed boxes, with SBOM discussed first, followed by AIBOMs.

9.1.1.1 *Current state of practice of SBOM to support the management of the supply chain of software and AI systems*

Finding #1

While the adoption of SBOMs remains limited, it is exhibiting a consistent upward trajectory.

Although SBOM adoption remains low, interest from both the software industry and the open-source communities is increasing [156–158], particularly as a means to improve supply chain management. One likely explanation for the current low adoption is the lack of clear incentives for organizations to adopt SBOMs [99, 170, 195]. For example, open-source systems are still not required to provide SBOMs [170]. Another contributing fac-

tor may be limited awareness and understanding of SBOMs and software supply chain regulations, especially in regions such as Italy [156]. In contrast to the United States, where awareness is higher—partly due to regulatory drivers such as Executive Order 14028 on Improving the Nation's Cybersecurity [186]—Italy currently lacks similarly strict requirements. The European Union has been moving in this direction only recently, for example, with the Cyber Resilience Act [56] (entered into force in December 2024), which aims to enhance software supply chain transparency, security, and accountability.

Finding #2

Multiple SBOMs can be generated for the same software system.

It is not uncommon to find multiple SBOMs generated for the same software system [157, 158]. This can occur for several reasons. SBOMs may be made available in different file formats, such as JSON or XML, to support compatibility with a variety of consumption tools. Moreover, these file formats can be manipulated with a variety of tools, enabling stakeholders without specialized technical skills to engage with the software component inventory [156]. Another reason is that multiple SBOMs may be generated to account for different deployment contexts, capturing variations in hardware and software configurations—such as targeting specific operating systems or processor architectures. Additionally, variations in software components may arise to meet regulatory compliance requirements [156]. For instance, certain countries may enforce specific legal or security regulations that constrain which features or components can be included in deployed software. As a result, a particular functionality may be available in one country but omitted in another, leading to different SBOMs for each localized version of the software. Lastly, SBOMs can exist in both development and release phases: they may be maintained under version control during the development lifecycle and also published as part of the official release artifacts. These factors contribute to the coexistence of multiple SBOMs for a single system, each serving a distinct purpose or audience.

Finding #3

SBOMs may not conform to their declared standard version.

Not all SBOMs fully comply with their declared SBOM standard versions [157, 158]. For example, integrating SBOMs within software supply chain frameworks may introduce deviations from the standard SBOM schema. When SBOMs are not valid with the expected standard version, tools that consume SBOMs—such as vulnerability scanners, license compliance analyzers, and other automated supply chain tools—may fail to process or accurately interpret the data—*e.g.,* in case such tools expect strict adherence to the SBOM standard. This would undermine the capabilities of tools consuming SBOMs, reducing the practical benefits that SBOM adoption is intended to provide.

Finding #4

Guidelines on how SBOMs should be distributed are overlooked.

SBOMs should accompany every official release of software systems and be updated with each code change to ensure accuracy and relevance [117, 171]. Regular and timely distribution of SBOMs is critical for maintaining transparency throughout the software supply chain, enabling key use cases such as effective vulnerability management, compliance verification, and risk assessment. Despite these recommendations, evidence suggests that they are frequently overlooked or not widely understood [157, 158]. This lack of adherence significantly undermines the potential benefits of adopting SBOMs, limiting their effectiveness in supporting the software supply chain.

Finding #5

Some SBOMs omit the primary component that identifies the software system they represent, and others exclude component data entirely.

Some SBOMs fail to describe the primary component [157, 158], despite NTIA guidance stating that an SBOM must include at least one primary component—*i.e.,* the software system the

SBOM represents [114]. Declaring the primary component is essential for accurately documenting direct and transitive dependencies. Furthermore, SBOMs should list all direct dependencies of the primary component and either include transitive dependencies or provide enough detail to resolve them recursively [185]. Without this information, it becomes difficult to distinguish between direct and transitive dependencies, undermining SBOM use cases.

More concerning is the presence of SBOMs that omit all component-level information entirely [157, 158]. In other words, some SBOMs may not report any information about components whatsoever. This represents a fundamental misuse of the SBOM concept, severely undermining its value and rendering such artifacts ineffective for key use cases.

Finding #6

Guidelines on the minimum data fields that SBOMs should contain are overlooked, with supplier names rarely included.

According to the NTIA's guidelines [185] issued by the United States Department of Commerce, an SBOM should include, for each component, the component name, version, supplier name, and unique identifiers. Additionally, metadata about the SBOM itself must be provided, such as the identity of the SBOM authoring entity, the date and time of generation, and unique identifiers for the SBOM document. These fields, referred to as NTIA's minimum data fields [185], represent baseline information necessary to uniquely and unambiguously describe software components. This foundational data enables effective tracking across the software supply chain and supports integration with other data sources (*e.g.,* vulnerability databases or license information).

However, most SBOMs appear not to report all NTIA's minimum data fields [157, 158]. Among these fields, unique identifiers are consistently present—likely due to their ease of generation and native support in SBOM generators. Most SBOMs rely on purl and, to a lesser extent, CPE for component identification, while SWID tags are virtually unused. On the other hand, the supplier name seems to be the least frequently reported data field.

According to Xia *et al.* [195], one of the reasons for not reporting all NTIA's minimum data fields is the unavailability of data to feed SBOMs. Nevertheless, NTIA guidance recommends that, in such cases, the absence of data should be explicitly asserted by the SBOM author [185]. Despite this, explicit non-assertion of missing data fields appears infrequent [157, 158].

Finding #7

Recommended data fields are scarcely reported in SBOMs, including component hashes and licenses.

According to the NTIA's guidelines [185], a comprehensive SBOM should, for each component, include the component hash, the life cycle phase indicating when and from where the data was collected, non-default component relationships, and license information. However, current SBOMs largely fail to report all of these recommended data fields [157, 158]. In particular, information on the life cycle phase is virtually absent [157, 158].

Only a few SBOMs include component hashes [157, 158], and even fewer report multiple hashes (generated using different hashing algorithms) for the same component. However, hashes are critical for identifying and linking components to external data sources (*e.g.*, vulnerability databases) [185], helping address challenges related to component renaming and whitelisting. Furthermore, hashes serve as a security mechanism for verifying the integrity of components by detecting unauthorized modifications. Including multiple hashes enhances interoperability by increasing compatibility with third-party tools and external systems that rely on specific hashing algorithms [12].

Similarly, the inclusion of license information in SBOMs is limited [157, 158]. One likely reason is the limited capability of SBOM generators to accurately retrieve licensing data. Indeed, the management of software license remains largely manual, even in the absence of SBOMs [156]. While a systematic comparison between SPDX and CycloneDX generators has not been conducted, the higher incidence of licensing information in SPDX SBOMs suggests that SPDX SBOM generators may be more effective at extracting and incorporating license metadata.

Finding #8

Vulnerabilities are virtually never reported in SBOMs.

Although prior studies have highlighted the value of enriching SBOMs with vulnerability information [67, 170, 195, 199], such information is virtually never reported in current SBOMs [157, 158]. While best practices recommend storing vulnerability information—including exploitability—in dedicated, linked artifacts like VEX documents, the latter appears to be unused. Several factors may contribute to this lack of vulnerability information. First, software suppliers may be reluctant to publicly disclose vulnerabilities due to reputational concerns or fears that such disclosures could serve as roadmaps for attackers. Second, SBOM generators may lack support for automatically retrieving and integrating up-to-date vulnerability data from relevant sources. Lastly, vulnerability data may already be reported by security tools [67, 156], but integration into SBOMs is hindered by the absence of standardized formats or interoperability mechanisms.

Finding #9

SBOM generators are seeing growing adoption, especially through seamless integration with build processes and CI/CD pipelines.

Standard creators (*i.e.,* SPDX and CycloneDX) and forges (*e.g.,* GitHub) are working to make available to software suppliers convenient tools for SBOM generation [44, 45, 157, 158]. In the open-source communities, in particular, a wider adoption of SBOMs via build-integration tools (*e.g.,* Maven) and CI/CD tools (*i.e.,* GitHub Action) was observed compared to the adoption of SBOMs via libraries [157, 158]. This is probably due to the ease of use of both build-integration and CI/CD tools: the former allows SBOMs to be generated when building a software system, while the latter when a certain event occurs (*e.g.,* push) [156]. Moreover, it can be expected an increase in the adoption trend of SBOMs since, on March 28, 2023, GitHub has introduced a new feature to export SBOMs that allows anyone with read access to a GitHub

repository to generate an SPDX SBOM by clicking on a button on the dependency graph page of the repository [75]. This self-service feature not only enables on-demand SBOM generation but could also be integrated into CI/CD pipelines and build-automation processes. Therefore, it can be expected that more open-source software systems will adopt SBOMs in the future.

Finding #10

SBOM generators yield inaccuracies.

One of the primary limitations of current SBOM generators is the inaccuracies in the SBOMs they generate [156]. This concern aligns with Xia *et al.* [195], who identify the limited availability and capabilities of SBOM tools as a key barrier to broader adoption. These inaccuracies negatively affect the fidelity of component representations.

For instance, GitHub's SBOM generator exhibits significant—and worsening—accuracy issues due to the limitations of the underlying SCA mechanisms [44, 45]. Such inaccuracies are not merely technical inconveniences, as they increase the risk of undetected vulnerabilities and license compliance violations, potentially leading to security breaches or legal consequences. The misrepresentation of dependencies arises from various error types [45], mostly caused by limitations in dealing with the information from metadata and package managers of software systems. These errors include unindexed dependencies, renamed components or suppliers, inconsistent component naming conventions (*e.g.*, hyphen "−" vs. underscore "_"), unexpected packaging structures, and archived components that have been removed from indexing.

Finding #11

Mitigating SBOM inaccuracies may involve using multiple generators although validating SBOMs remain a critical step.

The use of multiple SBOM generators can improve the completeness of component information provided in SBOMs [156]. This is because multiple tools may produce different SBOMs, each

identifying different sets of components. By leveraging multiple generators, suppliers can increase the likelihood of detecting all relevant components and their associated information. However, this approach necessitates merging the outputs from different tools into a unified SBOM, which can be challenging due to potential inconsistencies or contradictions between individual SBOMs. Therefore, validation is a critical step [156].

Finding #12

SBOMs enable fine-grained tracking and management of software components.

Traditionally, software documentation is used to produce and distribute the component inventory of software systems [156]. This documentation can take the form of spreadsheets that manually record metadata such as component versions and licensing information [156]. However, manual entry introduces the risk of inaccuracies, as the documented information may not reflect the actual components present in the source code. Documentation may also include UML diagrams—such as component or class diagrams—which provide structural overviews but lack the granularity needed to describe lower-level components, such as libraries [156]. In contrast, SBOMs enable a finer-grained representation of software composition, capable of capturing information at the level of libraries and even individual files—though file-level documentation is virtually absent in practice [157, 158]. SBOMs hold significant potential for managing software component inventories at fine-grained levels (*e.g.*, reused code snippets).

9.1.1.2 *Current state of practice of AIBOM to support the management of the supply chain of software and AI systems*

Finding #13

AIBOMs are experiencing growing interest.

AIBOMs are experiencing growing and sustained interest across both research and industry domains [155]. While initially a niche topic, recent years have witnessed a noticeable surge in attention, reflecting a broader recognition of their relevance in

addressing the challenges associated with AI. The upward trend suggests that AIBOMs are evolving from a conceptual idea into a practical and widely adopted technology, and this momentum is expected to continue.

Finding #14

AIBOMs can enhance the quality, security, and trustworthiness of AI and AI-enabled systems and processes by increasing transparency, supporting regulatory compliance, enabling effective risk management, and clarifying accountability among stakeholders.

The improvement of software systems and processes through the use of AIBOMs is increasingly recognized as a critical enabler for building secure, reliable, and trustworthy systems [155]. Transparency and traceability play a foundational role in enhancing the security, safety, and reliability of such systems [155]. By offering a detailed and structured inventory of all components involved—ranging from datasets and AI models to hardware and development tools—AIBOMs make the inner workings of AI and AI-enabled systems more visible and understandable. This also helps addressing accountability in AI and AI-enabled systems, where responsibility is often diffused across multiple stakeholders in the supply chain (*e.g.,* data scientists, model trainers, infrastructure providers). By providing a structured and traceable record of data, models, and decisions—including AI-specific concerns like data lineage, model provenance, and ethical or environmental considerations—AIBOMs enable clearer attribution of responsibility, particularly in cases where systems cause harm or make erroneous decisions. This level of visibility directly supports provenance tracking, risk mitigation, and the identification of potential issues across the supply chain.

In terms of software process improvement, AIBOMs are particularly valuable for ensuring and demonstrating regulatory compliance, especially as AI and AI-enabled systems increasingly fall under the scope of emerging laws, directives, and industry-specific standards [155]. AIBOMs help organizations align with both legal obligations and ethical expectations. Furthermore, AIBOMs contribute to broader process goals such as trust-building,

responsible AI development, and risk management, while also enabling automation, collaboration, and maintenance throughout the software lifecycle [155]. In this way, AIBOMs not only enhance the technical quality of AI and AI-enabled systems but also promote organizational readiness and accountability in an evolving regulatory and technological landscape.

Finding #15

AIBOMs can help ensure compliance with AI regulations and governance standards.

As AI becomes increasingly embedded into software systems, regulatory bodies and standards organizations worldwide are establishing frameworks to ensure the responsible development and deployment of AI. AIBOMs can play a crucial role in supporting compliance with these evolving legal and technical requirements [155]. For example, AIBOMs align with international standards such as ISO/IEC 42001 and the IEEE P70xx series, which emphasize principles of AI governance including transparency, accountability, and risk mitigation.

In the European Union, the EU AI Act introduces binding requirements for risk classification, transparency, documentation, and post-market monitoring, particularly for systems using AI, particularly those deemed high-risk. These requirements extend to both software and physical systems such as medical devices, vehicles, and industrial equipment. AIBOMs facilitate compliance by offering a verifiable inventory of system components and associated risk factors, thus supporting conformity assessments and audits. Moreover, AIBOMs can facilitate compliance with pre-existing EU laws that intersect with AI (*e.g.,* GDPR, EMA Requirements for Medical Device).

In the United States, a series of executive orders—*e.g.,* Executive Order 13960 on Promoting the Use of Trustworthy Artificial Intelligence in the Federal Government, Executive Order 14028 on Improving the Nation's Cybersecurity, and Executive Order 14110 on Safe, Secure, and Trustworthy Development and Use of Artificial Intelligence—underscore the federal government's commitment to AI accountability, transparency, and risk management. Other jurisdictions are also advancing AI-specific gover-

nance. For instance, Canada and Singapore have released policy frameworks addressing generative AI, including requirements for content transparency and watermarking. China mandates the registration of foundation models in public-facing systems and imposes strict data protection and safety regulations.

Finding #16

For both AI models and datasets, AIBOM should include provenance information, including authorship, licensing, and dependencies, as well as limitations, such as bias or privacy considerations.

Detailed provenance information for both AI models and datasets in AIBOMs serve to trace the lineage and origin of artifacts by capturing aspects such as authorship (*e.g.,* suppliers), licensing terms, and dependencies on other AI models, datasets, or libraries [155]. They may also reference the data source (*e.g.,* synthetic, open-source, or customer-provided), the data acquisition method (*e.g.,* mining, scraping, manual annotation), or platforms hosting the artifacts (*e.g.,* Hugging Face).

Complementing provenance, the explicit documentation of limitations is equally critical [155]. Limitations pertain to known shortcomings of AI models or datasets, such as generalization failures, dataset imbalance, biases (*e.g.,* label or selection bias), or issues arising from restricted data diversity. Moreover, AIBOMs can include specific disclosures related to privacy risks, such as the presence of sensitive data or the methods applied to ensure data anonymization. By documenting provenance and limitations, AIBOMs contribute to the transparency, accountability, and trustworthiness of AI-enabled systems.

Finding #17

For AI models, AIBOM should include information about their architecture, training, and usage, as well as authenticity, security, and sustainability considerations.

AIBOMs should comprehensively describe the architecture of AI models [155]. This encompass the specific model architecture (*e.g.,* structure, layers, activation functions), the broader architec-

ture family (*e.g.*, transformers, CNNs), and the identification of any base or parent models used as foundations for development or fine-tuning. Training information should document the full range of technical configurations used during model development, including algorithms, hyperparameters (*e.g.*, learning rate, batch size), software and hardware environments, and links to the datasets used in training, validation, and testing [155].

Usage-related information should clarify both the operational context and constraints of the AI models, such as the expected inputs and outputs, supported hardware/software, and any known or anticipated cases of misuse or adversarial application [155]. Authenticity can be established through cryptographic mechanisms like digital signatures, certifications, or attestations, which verify the origin, integrity, and trustworthiness of a model, and may be supported by blockchain-based infrastructures [155]. Security information should document known vulnerabilities, associated mitigations, and applied security mechanisms (*e.g.*, encryption, access control), referencing public vulnerability databases (*e.g.*, CVE, NVD) where applicable [155].

Sustainability concerns relate to the environmental footprint of model training and deployment, including energy consumption, carbon emissions, and resource usage such as water for data center cooling [155]. AIBOMs may quantify these using standardized units (*e.g.*, kWh, MJ) and report the contributing computational factors, such as GPU hours or training duration. Incorporating these dimensions into AIBOMs enhances the transparency, reproducibility, and ethical stewardship of AI models, ultimately fostering greater accountability and responsible AI development.

Finding #18

For datasets, AIBOM should document all procedures applied to data prior to its use in training or evaluation of AI models.

The procedures applied to datasets encompasses the entire data preparation pipeline, starting from initial acquisition to the final state in which the data is used for the training or evaluation of AI models [155]. These procedures typically include data collection strategies, curation processes to select or filter relevant sam-

ples, cleaning to remove noise or inconsistencies, transformation techniques for restructuring or reformatting data, augmentation to synthetically expand the dataset, and sampling methods for selecting representative subsets. In addition, mechanisms describing how the dataset evolves over time—whether through batch updates, real-time feeds (*e.g.*, stock market data), incremental additions (*e.g.*, social media or news streams), or manual interventions that require human oversight and verification—should be described as well.

In systems that interface with the physical world, AIBOMs may also document the sensors used to collect data [155]. This includes not only the type and function of the sensors but also calibration parameters and the structure in which sensor data is stored—such as key-value pairs and the naming conventions for those keys. Additionally, AIBOMs can capture key dataset characteristics, such as overall size (*e.g.*, in bytes or number of records) and labeling information (*e.g.*, indicating whether and how data points are annotated for supervised learning tasks).

Finding #19

AIBOMs can be used to manage AI components, ensuring compliance, and enhance dependability.

AIBOMs are increasingly recognized as critical means for the management of AI and AI-enabled systems [155]. For example, AIBOMs can play a pivotal role by enhancing transparency into the provenance, configuration, and licensing of external models and datasets of third-party AI component. This visibility supports risk-aware decision-making during the evaluation, procurement, and integration of AI assets, helping organizations mitigate legal, ethical, and operational risks associated with external dependencies. Beyond procurement, AIBOMs can contribute to AIOps pipeline management [155], where they facilitate the automated, auditable, and secure deployment of AI models. Integrated into CI/CD workflows, AIBOMs have the potential to support continuous model validation and retraining, particularly in dynamic environments subject to data or concept drift. This ensures that AI and AI-enabled systems are not only deployed responsibly but are also continuously monitored and updated in line with evolv-

ing performance and compliance requirements. More broadly, AIBOMs can support AI model lifecycle management by enabling traceability, rollback, and version control.

From a compliance standpoint, AIBOMs serve as mechanisms to enforce regulatory, ethical, and organizational policies [155]. Legal and compliance teams can use AIBOMs to review contractual terms, track adherence to internal guidelines, and prevent regulatory violations—especially in regulated industries such as healthcare, defense, or finance. AIBOMs can enable structured governance and facilitate cross-stakeholder communication, providing standardized and interpretable documentation that bridges the gap between technical teams, auditors, legal departments, and end-users.

In terms of dependability, AIBOMs can bolster security-by-design practices by enabling early identification of vulnerabilities such as model poisoning, backdoors, or adversarial manipulation [155]. They have the potential to allow for fine-grained auditing in the event of a breach or model failure, helping identify compromised components, trace the root causes of anomalous behavior, and support fast remediation. AIBOMs can also alert stakeholders to the use of insecure third-party libraries or outdated dependencies, fostering a proactive approach to AI and AI-enabled system hardening. Furthermore, they can support troubleshooting and debugging by linking model behavior to upstream changes in components, configurations, or datasets, enabling more efficient fault diagnosis and system recovery.

Finding #20

AIBOMs adopt structured, machine-readable formats like JSON and XML and are supported by SPDX and CycloneDX standards.

Similar to SBOMs [156–158], AIBOMs are typically represented in structured file formats such as JSON and, to a lesser extent, XML [155]. These formats facilitate the handling of component in a machine-readable manner, which is essential for downstream processing and integration with other tools in the supply chain. As with SBOMs [157, 158], SPDX and CycloneDX have emerged as the leading standards for structuring AIBOMs. CycloneDX

introduced support for AI-specific metadata in version 1.5, while SPDX incorporated similar extensions in its 3.0 version.

Finding #21

AIBOM generators are lacking, and their development is hindered by challenges in capturing the dynamic behavior of AI components and ensuring access adequate data.

Despite growing market interest, tool support for AIBOM generation remains sparse [155]. Current AIBOMs appear being manually generated, which is costly, time-intensive, and difficult to scale—particularly when third-party models or datasets are involved. As a result, AIBOM generation remains fragmented, error-prone, and impractical.

Several challenges hinder the development of AIBOM generators [155]. One major issue is the need to capture the dynamic nature of AI and AI-enabled systems—especially their capacity to evolve over time [155]. Unlike traditional software systems, which typically exhibit static behavior post-deployment, AI and AI-enabled systems can adapt as they are exposed to new data. For instance, a model might update its internal state or predictions based on this new data, complicating traceability and documentation.

Another critical challenge lies in data availability and quality [155]. Resources such as model cards, datasheets, factsheets, and interactive catalogs are emerging as important sources for AIBOM generation [155]. However, these artifacts vary significantly in content, structure, and format. For example, while model cards offer a promising foundation, they are often non-standardized, written in unstructured formats, and not readily machine-readable, making automated parsing difficult. In cases where such data is not adequately structured, the generation process relies heavily on text extraction techniques that may introduce inaccuracies. This approach that makes AIBOM generation process difficult and error-prone.

Finding #22

AIBOM consumption tools are lacking, and their development is hindered by challenges in the handling and availability of data.

On the consumption side, there are no dedicated tools to support key use cases such as AI component management, compliance verification, or security assessment, which further hinders practical AIBOM adoption [155].

Several challenges hinder the development of AIBOM consumption tools [155]. One major issue involves concerns around intellectual property. Since AIBOMs can expose detailed information about AI components, there is a risk that proprietary technologies could be reverse-engineered.

Security auditing poses another challenge for AIBOM consumption [155]. Unlike traditional software components, AI components may require fundamentally different approaches to security evaluation. Existing vulnerability databases are inadequate for this purpose, as they lack clear definitions of AI-specific security issues.

Compounding these challenges is the lack of adequate underlying data needed to generate AIBOMs in the first place [155].

Finding #23

Interoperability challenges arise in aligning AIBOMs with other BOMs and existing infrastructures.

Interoperability challenges pertain aligning AIBOMs with other types of BOMs (*e.g.,* HBOMs) and in integrating AIBOMs with existing software and operational infrastructures [155]. Ensuring compatibility between AIBOMs and other BOM formats demands more than schema alignment; it necessitates reconciling fundamentally distinct informational domains. Likewise, integrating AIBOMs into operational pipelines—such as AIOps workflows encompassing data preparation, model training, configuration, and deployment—may necessitate revisiting such pipelines to accommodate AIBOMs.

Finding #24

Human-related challenges of AIBOMs center on the lack of education and awareness.

Challenges and limitations related to human aspects and AIBOMs include the need for better education and awareness [155]. This educational gap extends beyond basic technical literacy; it encompasses a broader understanding of the socio-technical implications, ethical responsibilities, and regulatory expectations tied to AIBOM adoption. Many practitioners, even those familiar with traditional SBOMs, may struggle to grasp the unique role of AIBOMs in ensuring AI transparency, accountability, and compliance. Consequently, the absence of standardized training programs or educational resources hinders consistent implementation across organizations.

9.1.2 *O2. Improving the tooling of SBOM and AIBOM to support the management of the supply chain of software and AI systems*

To better support the management of the software supply chain, SBOMs and AIBOMs should evolve beyond their current limitations by capturing more granular, diverse, and trustworthy information about software and AI components. Traditional SBOMs tend to operate at a coarse level of granularity, typically listing third-party packages, libraries, or modules used within a system. While this provides a basic overview of dependencies, it overlooks a wide range of critical artifacts—such as reused code snippets, models, and datasets—that may introduce a variety of risks (*e.g.,* legal, security, environmental, or ethical). Improving SBOMs/AIBOMs involves enhancing their ability to capture and represent the different types of components that may constitute software systems.

One key area for improvement lies in incorporating traceable information about external code reuse, particularly from informal sources such as Q&A forums. Developers frequently copy and adapt code snippets from platforms like Stack Overflow, but these small fragments often go undocumented in current SBOMs. To address this, new tools and techniques must be developed

to identify, attribute, and document such reuse instances in a standardized and machine-readable format. For example, AIRBORNE [127] is a SBOM generator capable to combine static code analysis, repository mining, and clone detection to recognize self-admitted Stack Overflow references and embed them into SBOMs. Capturing this level of detail adds transparency to development practices and supports better legal compliance, especially in relation to licensing and intellectual property concerns.

AIBOMs are still in their infancy, and the tool support is even worse than SBOMs. ALOHA [51] marks an important initial step toward the development of practical AIBOM generators, demonstrating the ability to extract and structure key information from Hugging Face model cards (and related metadata). It effectively processes both structured metadata and unstructured free-text descriptions, organizing them into a machine-readable format that supports greater transparency and traceability of AI components within the software supply chain. However, model cards are inconsistently structured and frequently lack critical information such as the training datasets or known limitations. To enhance the effectiveness of AIBOMs in software supply chain management, it is essential to ensure the availability of high-quality data sources and to develop robust tools capable of systematically extracting, validating, and structuring this information. Equally important is the promotion of consistent and comprehensive documentation practices among model developers, as the accuracy and completeness of AIBOMs heavily depend on the quality of the underlying model metadata. Moreover, standardizing the metadata fields and aligning them with widely adopted formats like CycloneDX can improve interoperability and allow AIBOMs to be easily integrated into broader supply chain management workflows.

9.2 IMPLICATIONS AND FUTURE DIRECTIONS

The obtained results have implications for several stakeholders, in particular, software suppliers, choosers, operators, and researchers. The remainder of this section delineates the implica-

tions for each of them, as well as for groups of them where the same implications apply to more than one stakeholder.

9.2.1 *Software Suppliers, Choosers, and Operators*

ADOPT SBOMS AND AIBOMS Software suppliers, choosers, and operators can take advantage of SBOMs and AIBOMs to document component information at different levels of granularity, including at the library level, which cannot be effectively represented by common software documentation (*e.g.,* UML diagrams), as well as AI-specific components (*e.g.,* models, datasets) [155, 156].

MONITOR SOFTWARE SUPPLY CHAIN REGULATIONS As SBOM and AIBOM adoption becomes increasingly critical—driven by emerging regulations such as the EU Cyber Resilience Act [56] or the EU AI Act [58]—software suppliers, choosers, and operators will need to stay ahead of new software supply chain requirements. Therefore, they must remain vigilant and well-informed about software supply chain regulations. To support this, organizations should leverage dedicated communication channels—such as internal newsletters—to disseminate timely and relevant regulatory updates [155, 156].

PAY ATTENTION TO THE CONTENT OF SBOMS AND AIBOMS Software suppliers, choosers, and operators must account for the possibility that SBOMs and AIBOMs may contain inaccuracies or omissions in the reported components, potentially undermining critical use cases such as vulnerability assessment and license compliance analysis [44, 45, 155].

9.2.2 *Software Suppliers*

GENERATE SBOMS AND AIBOMS Software suppliers are likely to face increasing pressure to generate and distribute SBOMs, including AIBOMs, driven by growing demand from major stakeholders and regulatory requirements—particularly those introduced by government agencies [155, 157, 158]. To stay ahead of these, suppliers should proactively integrate SBOM and AI-

BOM generation into their development pipelines in anticipation of potential compliance mandates. Tools such as the cyclonedx-maven-plugin and spdx-maven-plugin can be easily integrated into projects that already use Maven as a package manager, requiring minimal effort [157, 158]. Software suppliers should also consider to augment their SBOMs with fine-grained information like explicit code reuse from Q&A and software repository platforms. In this regard, AIRBORNE [127] may come in handy, although it currently supports only explicit code reuse from Stack Overflow. As for AIBOMs, software suppliers may consider ALOHA [51]. We made both AIRBORNE and ALHOA open-source available on GitHub, so that anyone may contribute to their growth and diffusion.

GENERATE SBOMS AND AIBOMS USING WELL-ESTABLISHED FILE FORMATS Software suppliers should generate SBOMs and AIBOMs using widely adopted and well-supported file formats, such as JSON or XML [155–158]. Since no universally accepted file format exists, they could also use multiple file formats to improve interoperability across the SBOM toolchain.

GENERATE SBOMS AND AIBOMS TAILORED TO SPECIFIC DEPLOYMENTS Software suppliers should generate and distribute SBOMs tailored to each supported hardware or software configuration to accurately represent the components required to deploy/run that software [157, 158]. Generating distinct SBOMs for different deployment targets enhances the precision of the reported components. A similar consideration can be done for AIBOMs [155].

USE MULTIPLE SBOM AND AIBOM GENERATORS Software suppliers can improve the completeness and correctness of their software component inventories by leveraging multiple SBOM and AIBOM generators, although validating the resulting SBOMs and AIBOMs still requires manual effort [155, 156].

MAINTAIN AND DELIVER UP-TO-DATE SBOMS AND AIBOMS Software suppliers should distribute SBOMs, including AIBOMs, following the NTIA's recommendations [117], including deliver-

ing updated SBOMs at each code change and release version [157, 158].

VALIDATE GENERATED SBOMS AND AIBOMS Software suppliers should check the compliance of the SBOMs and AIBOMs they generate with respect to their declared standard and version [155, 157, 158]. To that end, convenient tools like sbom-utility [36] might come in handy to them.

DATA FIELDS THAT ARE WORTH INCLUDING IN SBOMS AND AIBOMS Software suppliers should list the primary components and their direct dependencies in SBOMs [157, 158]. The NTIA's minimum data fields should be considered as non-negotiable and always include them [157, 158]. In case a data field cannot be provided (*e.g.*, the supplier name is unknown), software suppliers should explicitly assert the missing/unavailable information as per NTIA's guidelines [185]. Additionally, software suppliers should include NTIA's recommended data fields in their SBOMs [157, 158], ensuring that each component includes license information. If a component's license is not available, it should be manually added. Software suppliers should also provide more than one hash (each generated with a different hashing algorithm) for each component reported in their SBOMs.

Similar considerations apply to AIBOMs, as they are SBOMs specialized in describing AI components. While specific guidelines for AIBOM data fields have yet to be established, software suppliers should consider including information about model performance and limitations, model architecture (encompassing architecture family, base model, and parent model), training and testing datasets and procedures, as well as ethical and sustainability considerations (*e.g.*, privacy risks, racial biases, environmental impact, or energy consumption) [155].

INCLUDE VULNERABILITY INFORMATION IN SBOMS/AIBOMS OR VEX Following NTIA recommendations [116], software suppliers should integrate information on vulnerabilities in SBOMs/AIBOMs or by using data structures linked to them like VEX [155, 157, 158].

9.2.3 *Software Choosers*

LEVERAGE SBOMS AND AIBOMS DURING PROCUREMENT Software choosers should leverage SBOMs and AIBOMs during the procurement of software components to evaluate the security, compliance, and traceability of such components [155, 156]. They should also require that SBOMs and AIBOMs include all essential information—such as the NTIA's minimum and/or recommended data fields—to support a broad range of use cases [157, 158]. As regulatory demands increase, especially in highly regulated domains, SBOMs and AIBOMs are likely to become mandatory procurement requirements [157, 158].

9.2.4 *Software Operators*

LEVERAGE SBOMS AND AIBOMS IN CI/CD PIPELINES TO MITIGATE RISKS Software operators should integrate SBOM and AIBOMs into their CI/CD pipelines to monitor eventual risks [155–158]. For example, to identify components that need to be updated due to being deprecated or affected by vulnerabilities, or for software license compliance [156]. In case of AIBOMs, risk management involves AI-specific issues, including dataset contamination, adversarial attacks, model drift, and data poisoning [155].

9.2.5 *Researchers*

IMPROVE THE TOOLING FOR GENERATING SBOMS AND AIBOMS SBOM and AIBOM generators remain underdeveloped. Most SBOM generators focus on representing coarse-grained components, such as software libraries and packages. Researchers need to make a concerted effort to include fine-grained components in SBOMs, such as AI-generated code and code reused from Q&A or software repository platforms. In this regard, AIRBORNE represents an important first step. It demonstrates that augmenting SBOMs with fine-grained provenance data—such as explicit references to Stack Overflow and the files that include them—can be automated, resulting in richer traceability. On the other hand, tools for generating AIBOMs are nearly nonexistent.

Our AIBOM generator, ALOHA, shows that automated extraction of data from collaborative AI platforms like Hugging Face can produce structured AIBOMs that capture models, datasets, and their relationships. However, this approach alone is insufficient for widespread AIBOM generation. Further research is needed to develop more comprehensive tools for generating AIBOMs on a broader scale.

IMPROVE THE ACCURACY OF SBOM AND AIBOM GENERATORS Research should aim to develop reliable, automated SBOM and AIBOM generators [155, 156]. This may require improving the underlying SCA mechanisms used to identify and track software components [44, 45]. Researchers should also improve and make available convenient, high-quality data sources for SBOM and AIBOM generation [44, 45, 155]. However, existing data sources may contain incorrect or incomplete information which, if used to populate an SBOM/AIBOM, could result in a non-compliant or inaccurate SBOM/AIBOM [156–158]. For example, this issue is particularly critical when the SBOM/AIBOM should include vulnerability information, where accuracy is essential [157, 158]. In the specific case of AIBOMs, the need for approaches that can automatically extract relevant data from AI artifacts—such as model training logs or dataset lineage—is even more critical because existing information sources—such as model cards [51]—provide only partial coverage, lack machine readability, and lack standardization [155].

TRACK CODE REUSE Researchers should enhance SBOM/AIBOM generators to detect a broader range of code reuse, including implicit or unattributed reuse of code from sources like Q&A and software-repository platforms or AI-generated code [51, 127, 156].

UNIQUE IDENTIFIERS FOR SUPPLIERS Researchers should establish mechanisms for globally identifying suppliers. This would facilitate the retrieval of supplier information from external databases when generating SBOMs and AIBOMs [155, 157, 158]. Currently, suppliers are often poorly reported [157, 158],

likely due to the lack of global identification standards similar to those used for components (*e.g.,* CPE or purl).

10

CONCLUSION

This thesis investigated how Software Bill of Materials (SBOMs) and Artificial Intelligence Bill of Materials (AIBOMs) are generated and used to support software supply chain management, with the twin goals of understanding current state of practice and developing supporting tools. To this end, qualitative and quantitative methods were employed: semi-structured interviews with practitioners, mining of open-source repositories, an empirical evaluation of a widely used SBOM generator (GitHub's SBOM generator), and a Multivocal Literature Review (MLR) on AIBOMs. Two supporting tools were also designed, implemented, and empirically evaluated: AIRBORNE for fine-grained SBOMs, which links source files to Stack Overflow provenance, and ALOHA for AIBOMs derived from Hugging Face model cards.

Empirical evidence from industry interviews shows that awareness of SBOM principles and relevant regulations is limited among practitioners, and that adoption is still nascent even where software supply chain risks are recognized. The mining study of public GitHub repositories corroborates this: SBOM generation is increasing but remains rare. Many produced SBOMs lack the minimum required or recommended data fields, including vulnerability information. The empirical assessment of GitHub's SBOM generator revealed concrete technical causes for inaccurate SBOM content: oversimplified processing by the underlying Software Composition Analysis (SCA) mechanisms leads to misidentification and misprofiling of components. The MLR on AIBOMs highlights parallel dynamics in the AI ecosystem: AIBOMs promise improved transparency and regulatory alignment for models and datasets, but tooling is immature, interoperability is poor, and stakeholder understanding is uneven.

AIRBORNE and ALOHA instantiate two practical paths forward. AIRBORNE demonstrates that augmenting SBOMs with fine-grained provenance data—explicit references to Stack Overflow and the files that include them—can be automated and

yields richer traceability. ALOHA shows that automated extraction of data from collaborative AI platforms like Hugging Face can produce structured AIBOMs that capture models, datasets, and their relationships. Both tools succeeded in their targeted evaluations, but their deployment also surfaced challenges.

Taken together, the results suggest clear implications for practice and research. For researchers, improving SBOM/AIBOM generators includes richer identification of component types and better handling of transitive and non-code artifacts. For software suppliers, choosers, and operators, the evidence supports stronger guidance on the adoption and use of SBOM and AIBOM.

A

PUBLICATIONS

The complete list of publications is provided below, with the ⋆ symbol marking the publications discussed in this thesis.

JOURNAL ARTICLES

[J7] ⋆ Sabato Nocera, Massimiliano Di Penta, Fatima Ahmed, Simone Romano, and Giuseppe Scanniello. "What We Know about AIBOMs: Results from a Multivocal Literature Review on Artificial Intelligence Bill of Materials." In: ACM Trans. Softw. Eng. Methodol. (Second round of review).

[J6] Sabato Nocera, Simone Romano, Rita Francese, Francesco Casertano, and Giuseppe Scanniello. "Securing Web Apps with ChatGPT: A Teaching Plan for Undergraduate Students." In: J. Syst. Softw. (Second round of review).

[J5] Sabato Nocera, Simone Romano, Rita Francese, and Giuseppe Scanniello. "Security Issues in Python Open-source Software: A Mining Study from GitHub." In: Empir. Softw. Eng. (Second round of review).

[J4] Sira Vegas, Sabato Nocera, Giuseppe Scanniello, and Natalia Juristo. "Investigating Causality in Software Repositories Using Cohort-Based Methods." In: IEEE Trans. Software Eng. (Under submission).

[J3] ⋆ Daniele Bifolco, Simone Romano, Sabato Nocera, Rita Francese, Giuseppe Scanniello, and Massimiliano Di Penta. "An empirical study on the accuracy of GitHub's dependency graph and the nature of its inaccuracy." In: Inf. Softw. Technol. 187 (2025), p. 107854. doi: 10.1016/J.INFSOF.2025.107854. url: https://doi.org/10.1016/j.infsof.2025.107854.

[J2] ⋆ Sabato Nocera, Simone Romano, Massimiliano Di Penta, Rita Francese, and Giuseppe Scanniello. "On the adoption of software bill of materials in open-source software projects." In: J. Syst. Softw. 230 (2025), p. 112540. doi: 10.1016/J.JSS.2025.112540. url: https://doi.org/10.1016/j.jss.2025.112540.

[J1] Sabato Nocera, Simone Romano, Rita Francese, and Giuseppe Scanniello. "Software engineering education: Results from a training intervention based on SonarCloud when developing web apps." In: J. Syst. Softw. 222 (2025), p. 112308. doi: 10.1016/J.JSS.2024.112308. url: https://doi.org/10.1016/j.jss.2024.112308.

CONFERENCE PROCEEDINGS

[C18] ⋆ Sabato Nocera, Massimiliano Di Penta, Simone Romano, and Giuseppe Scanniello. "AIRBORNE: Augmented fIne-grained softwaRe Bill Of mateRials geNerator for stack ovErflow." In: Submitted to the 48th IEEE/ACM International Conference on Software Engineering.

[C17] Sabato Nocera, Nyyti Saarimäki, Valentina Lenarduzzi, Davide Taibi, and Sira Vegas. "Stop Comparing Apples and Oranges: Matching for Better Results in Mining Software Repositories Studies." In: Submitted to the 23rd IEEE/ACM International Conference on Mining Software Repositories.

[C16] Sabato Nocera, Sira Vegas, Giuseppe Scanniello, Massimiliano Di Penta, and Natalia Juristo. "Causal or Correlational? A Cohort Study on the Effects of Code Smells on Class Change- and Fault-Proneness." In: Submitted to the 48th IEEE/ACM International Conference on Software Engineering.

[C15] Sabato Nocera, Davide Fucci, and Giuseppe Scanniello. "Dealing with SonarQube Cloud: Initial Results from a Mining Software Repository Study." In: Proceedings of the 19th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2025, Honolulu,

Hawai, September 28 - October 3, 2025. IEEE, Accepted and presented (proceedings forthcoming).

[C14] ⋆ Riccardo D'Avino, Sabato Nocera, Daniele Bifolco, Federica Pepe, Massimiliano Di Penta, and Giuseppe Scanniello. "ALOHA: A (IBoM) tooL generatOr for Hugging fAce." In: Proceedings of the 29th International Conference on Evaluation and Assessment in Software Engineering, EASE 2025, Istanbul, Turkey, June 17-20, 2025. ACM, 2025, Accepted and presented (proceedings forthcoming). doi: 10.1145/3756681.3756998. url: https://doi.org/10.1145/3756681.3756998.

[C13] Sabato Nocera and Giuseppe Scanniello. "Missing Threats: Dealing with the Treatment-sensitive Factorial Structure Bias in Empirical Software Engineering." In: Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering, FSE Companion 2025, Clarion Hotel Trondheim, Trondheim, Norway, June 23-28, 2025. Ed. by Leonardo Montecchi, Jingyue Li, Denys Poshyvanyk, and Dongmei Zhang. ACM, 2025, pp. 701–705. doi: 10.1145/3696630.3728527. url: https://doi.org/10.1145/3696630.3728527.

[C12] Sabato Nocera, Sira Vegas, Giuseppe Scanniello, and Natalia Juristo. "Causal Inference Needs More Than Analysis: The Role of Study Design." In: Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering, FSE Companion 2025, Clarion Hotel Trondheim, Trondheim, Norway, June 23-28, 2025. Ed. by Leonardo Montecchi, Jingyue Li, Denys Poshyvanyk, and Dongmei Zhang. ACM, 2025, pp. 1428–1431. doi: 10.1145/3696630.3731619. url: https://doi.org/10.1145/3696630.3731619.

[C11] Sabato Nocera, Sira Vegas, Giuseppe Scanniello, and Natalia Juristo. "Software Composition Analysis and Supply Chain Security in Apache Projects: an Empirical Study." In: 22nd IEEE/ACM International Conference on Mining Software Repositories, MSR@ICSE 2025, Ottawa, ON, Canada, April 28- 29, 2025. IEEE, 2025, pp. 103–115. doi:

10.1109/MSR66628.2025.00027. url: https://doi.org/10.1109/MSR66628.2025.00027.

[C10] Giuseppe Scanniello, Massimiliano Di Penta, Simone Romano, Rita Francese, Sabato Nocera, Pietro Cassieri, Daniele Bifolco, and Fiorella Zampetti. "MSR4SBOM: Min ing Software Repositories for enhanced Software Bills of Materials." In: Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2024, Barcelona, Spain, October 24-25, 2024. Ed. by Xavier Franch, Maya Daneva, Silverio Martínez- Fernández, and Luigi Quaranta. ACM, 2024, pp. 589–593. doi: 10.1145/3674805.3695390. url: https://doi.org/10.1145/3674805.3695390.

[C9] ⋆ Sabato Nocera, Massimiliano Di Penta, Rita Francese, Simone Romano, and Giuseppe Scanniello. "If it's not SBOM, then what? How Italian Practitioners Manage the Software Supply Chain." In: IEEE International Conference on Software Maintenance and Evolution, ICSME 2024, Flagstaff, AZ, USA, October 6-11, 2024. IEEE, 2024, pp. 730–740. doi: 10.1109/ICSME58944.2024.00077. url: https://doi.org/10.1109/ICSME58944.2024.00077.

[C8] Sabato Nocera, Simone Romano, Dario Di Nucci, Rita Francese, Fabio Palomba, and Giuseppe Scanniello. "Do Static Analysis Tools Improve Awareness and Attitude Toward Secure Software Development?" In: Quality of Information and Communications Technology - 17th International Conference on the Quality of Information and Communications Technology, QUATIC 2024, Pisa, Italy, September 11-13, 2024, Proceedings. Ed. by Antonia Bertolino, João Pascoal Faria, Patricia Lago, and Laura Semini. Vol. 2178. Communications in Computer and Information Science. Springer, 2024, pp. 399–407. doi: 10.1007/978-3-031-70245-7_28. url: https://doi.org/10.1007/978-3-031-70245-7_28.

[C7] ⋆ Daniele Bifolco, Sabato Nocera, Simone Romano, Massimiliano Di Penta, Rita Francese, and Giuseppe Scanniello. "On the Accuracy of GitHub's Dependency Graph." In:

Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering, EASE 2024, Salerno, Italy, June 18-21, 2024. ACM, 2024, pp. 242–251. doi: 10.1145/3661167.3661175. url: https://doi.org/10.1145/3661167.3661175.

[C6] Sabato Nocera, Simone Romano, Rita Francese, and Giuseppe Scanniello. "Training for Security: Results from Using a Static Analysis Tool in the Development Pipeline of Web Apps." In: Proceedings of the 46th International Conference on Software Engineering: Software Engineering Education and Training, SEET@ICSE 2024, Lisbon, Portugal, April 14-20, 2024. ACM, 2024, pp. 253–263. doi: 10.1145/3639474.3640073. url: https://doi.org/10.1145/3639474.3640073.

[C5] ⋆ Sabato Nocera, Simone Romano, Massimiliano Di Penta, Rita Francese, and Giuseppe Scanniello. "Software Bill of Materials Adoption: A Mining Study from GitHub." In: IEEE International Conference on Software Maintenance and Evolution, ICSME 2023, Bogotá, Colombia, October 1-6, 2023. IEEE, 2023, pp. 39–49. doi: 10.1109/ICSME58846.2023.00016. url: https://doi.org/10.1109/ICSME58846.2023.00016.

[C4] Sabato Nocera, Simone Romano, Rita Francese, Riccardo Burlon, and Giuseppe Scanniello. "Managing Vulnerabilities in Software Projects: the Case of NTT Data." In: 49th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2023, Durres, Albania, September 6-8, 2023. IEEE, 2023, pp. 247–253. doi: 10.1109/SEAA60479.2023.00046. url: https://doi.org/10.1109/SEAA60479.2023.00046.

[C3] Sabato Nocera, Simone Romano, Rita Francese, and Giuseppe Scanniello. "A Large-scale Fine-grained Empirical Study on Security Concerns in Open-source Software." In: 49th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2023, Durres, Albania, September 6-8, 2023. IEEE, 2023, pp. 418–425. doi:

10.1109/SEAA60479.2023.00069. url: https://doi.org/10.1109/SEAA60479.2023.00069.

[C2] Sabato Nocera, Rita Francese, and Giuseppe Scanniello. "Training Bachelor Students to Design Better Quality Web Apps: Preliminary Results from a Prospective Empirical Investigation." In: Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering, EASE 2023, Oulu, Finland, June 14-16, 2023. ACM, 2023, pp. 465–469. doi: 10.1145/3593434.3593957. url: https://doi.org/10.1145/3593434.3593957.

[C1] Sabato Nocera, Simone Romano, Rita Francese, and Giuseppe Scanniello. "Training for Security: Planning the Use of a SAT in the Development Pipeline ofWeb Apps." In: 45th IEEE/ACM International Conference on Software Engineering: Software Engineering Education and Training, SEET@ICSE 2023, Melbourne, Australia, May 14-20, 2023. IEEE, 2023, pp. 40–45. doi: 10.1109/ICSE-SEET58685.2023.00010. url: https://doi.org/10.1109/ICSE-SEET58685.2023.00010.

BIBLIOGRAPHY

[1] ACM. *ACM Digital Library*. 2025. URL: https://dl.acm.org/.

[2] ATLAS.ti. *Computer-assisted qualitative data analysis software*. 2024. URL: https://atlasti.com/.

[3] Jean Adams, Frances C Hillier-Brown, Helen J Moore, Amelia A Lake, Vera Araujo-Soares, Martin White, and Carolyn Summerbell. "Searching and synthesising 'grey literature'and 'grey information'in public health: critical reflections on three case studies." In: *Systematic reviews* 5 (2016), pp. 1–11. DOI: 10.1186/s13643-016-0337-y. URL: https://doi.org/10.1186/s13643-016-0337-y.

[4] Richard J Adams, Palie Smart, and Anne Sigismund Huff. "Shades of grey: guidelines for working with the grey literature in systematic reviews for management and organizational studies." In: *International journal of management reviews* 19.4 (2017), pp. 432–454. DOI: 10.1111/ijmr.12102. URL: https://doi.org/10.1111/ijmr.12102.

[5] Adem Ait, Javier Luis Cánovas Izquierdo, and Jordi Cabot. "On the suitability of hugging face hub for empirical studies." In: *Empir. Softw. Eng.* 30.2 (2025), p. 57. DOI: 10.1007/S10664-024-10608-8. URL: https://doi.org/10.1007/s10664-024-10608-8.

[6] Mahmoud Alfadel, Diego Elias Costa, and Emad Shihab. "Empirical analysis of security vulnerabilities in Python packages." In: *Empir. Softw. Eng.* 28.3 (2023), p. 59. DOI: 10.1007/S10664-022-10278-4. URL: https://doi.org/10.1007/s10664-022-10278-4.

[7] Anchore. *SCA vs. SBOM: Differences + Benefits of Combining the Two*. Anchore. 2025. URL: https://anchore.com/sbom/sca-vs-sbom/.

[8] Anna Hermansen and Cailean Osborne. *The Economic and Workforce Impacts of Open Source AI: Insights from Industry, Academia, and Open Source Research Publications*. The Linux Foundation. 2025. URL: https://www.linuxfoundation.org/research/economic-impacts-of-open-source-ai.

[9] Anonymous. *What We Know about AIBoMs: Results from a Multivocal Literature Review on Artificial Intelligence Bill of Materials - Supplementary Materials*. 2025. URL: https://figshare.com/s/c696c3e160d521599ddf.

[10] Aqua Security Software. *Trivy Home - Trivy*. 2024. URL: https://trivy.dev/.

[11] Archi. *Archi- Open Source ArchiMate Modelling*. 2024. URL: https://www.archimatetool.com/.

[12] Musard Balliu, Benoit Baudry, Sofia Bobadilla, Mathias Ekstedt, Martin Monperrus, Javier Ron Arteaga, Aman Sharma, Gabriel Skoglund, César Soto-Valero, and Martin Wittlinger. "Challenges of Producing Software Bill of Materials for Java." In: *IEEE Secur. Priv.* 21.6 (2023), pp. 12–23. DOI: 10.1109/MSEC.2023.3302956. URL: https://doi.org/10.1109/MSEC.2023.3302956.

[13] Sebastian Baltes, Lorik Dumani, Christoph Treude, and Stephan Diehl. "SOTorrent: reconstructing and analyzing the evolution of stack overflow posts." In: *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*. Ed. by Andy Zaidman, Yasutaka Kamei, and Emily Hill. ACM, 2018, pp. 319–330. DOI: 10.1145/3196398.3196430. URL: https://doi.org/10.1145/3196398.3196430.

[14] Iain Barclay, Alun D. Preece, Ian J. Taylor, Swapna Krishnakumar Radha, and Jarek Nabrzyski. "Providing assurance and scrutability on shared data and machine learning models with verifiable credentials." In: *Concurr. Comput. Pract. Exp.* 35.18 (2023). DOI: 10.1002/CPE.6997. URL: https://doi.org/10.1002/cpe.6997.

[15] Iain Barclay, Alun D. Preece, Ian J. Taylor, and Dinesh C. Verma. "Towards Traceability in Data Ecosystems using a Bill of Materials Model." In: *Proceedings of the 11th Interna-*

tional Workshop on Science Gateways, Ljubljana, Slovenia, 12-14 June, 2019. Ed. by Vlado Stankovski and Sandra Gesing. Vol. 2975. CEUR Workshop Proceedings. CEUR-WS.org, 2019. URL: https://ceur-ws.org/Vol-2975/paper3.pdf.

[16] Daniel Bardenstein, Nitish Kulkarni, Jakob Frick, Marc Frankel, Tiff Hsieh, and Amy Villasenor. *Driving AI Transparency: the AI Bill of Materials*. White paper. 2023. URL: https://manifest-public-shareable-resources.s3.us-east-1.amazonaws.com/Manifest%2BAIBOM%2BWhitepaper.pdf.

[17] Victor R Basili. *Software modeling and measurement: the Goal/Question/Metric paradigm*. University of Maryland at College Park, 1992.

[18] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. "How the Apache community upgrades dependencies: an evolutionary study." In: *Empir. Softw. Eng.* 20.5 (2015), pp. 1275–1317. DOI: 10.1007/S10664-014-9325-9. URL: https://doi.org/10.1007/s10664-014-9325-9.

[19] Daniel Bendler and Michael Felderer. "Competency Models for Information Security and Cybersecurity Professionals: Analysis of Existing Work and a New Model." In: *ACM Trans. Comput. Educ.* 23.2 (2023), 25:1–25:33. DOI: 10.1145/3573205. URL: https://doi.org/10.1145/3573205.

[20] Karen Bennet, Gopi Krishnan Rajbahadur, Arthit Suriyawongkul, and Kate Stewart. *Implementing AI Bill of Materials (AI BOM) with SPDX 3.0 - A Comprehensive Guide to Creating AI and Dataset Bill of Materials*. White paper. 2024. URL: https://www.linuxfoundation.org/hubfs/LF%20Research/lfr_spdx_aibom_102524a.pdf.

[21] Tingting Bi, Boming Xia, Zhenchang Xing, Qinghua Lu, and Liming Zhu. "On the Way to SBOMs: Investigating Design Issues and Solutions in Practice." In: *ACM Trans. Softw. Eng. Methodol.* 33.6 (2024), p. 149. DOI: 10.1145/3654442. URL: https://doi.org/10.1145/3654442.

[22] Daniele Bifolco, Sabato Nocera, Simone Romano, Rita Francese, Giuseppe Scanniello, and Massimiliano Di Penta. *An Empirical Study on the Accuracy of GitHub's Dependency Graph and the Nature of Its Inaccuracy: A Replication Package*. May 2025. DOI: 10.5281/zenodo.15412178. URL: https://doi.org/10.5281/zenodo.15412178.

[23] Christopher Bogart, Christian Kästner, and James D. Herbsleb. "When It Breaks, It Breaks: How Ecosystem Developers Reason about the Stability of Dependencies." In: *30th IEEE/ACM International Conference on Automated Software Engineering Workshops, ASE Workshops 2015, Lincoln, NE, USA, November 9-13, 2015*. IEEE Computer Society, 2015, pp. 86–89. DOI: 10.1109/ASEW.2015.21. URL: https://doi.org/10.1109/ASEW.2015.21.

[24] Laura Bottner, Artur Hermann, Jeremias Eppler, Thomas Thüm, and Frank Kargl. "Evaluation of Free and Open Source Tools for Automated Software Composition Analysis." In: *Proceedings of the 7th ACM Computer Science in Cars Symposium, CSCS 2023, Darmstadt, Germany, 5 December 2023*. Ed. by Björn Brücher, Christoph Krauß, Mario Fritz, Hans-Joachim Hof, and Oliver Wasenmüller. ACM, 2023, 3:1–3:11. DOI: 10.1145/3631204.3631862. URL: https://doi.org/10.1145/3631204.3631862.

[25] Lionel C. Briand, Domenico Bianculli, Shiva Nejati, Fabrizio Pastore, and Mehrdad Sabetzadeh. "The Case for Context-Driven Software Engineering Research: Generalizability Is Overrated." In: *IEEE Softw.* 34.5 (2017), pp. 72–75. DOI: 10.1109/MS.2017.3571562. URL: https://doi.org/10.1109/MS.2017.3571562.

[26] Anesu Chaora, Nathan L. Ensmenger, and L. Jean Camp. "Discourse, Challenges, and Prospects Around the Adoption and Dissemination of Software Bills of Materials (SBOMs)." In: *IEEE International Symposium on Technology and Society, ISTAS 2023, Swansea, United Kingdom, September 13-15, 2023*. IEEE, 2023, pp. 1–4. DOI: 10.1109/ISTAS57930.2023.10305922. URL: https://doi.org/10.1109/ISTAS57930.2023.10305922.

[27] Checkmarx. *Application Security Testing Tool - Checkmarx Appsec Solution*. 2024. URL: https://checkmarx.com/.

[28] Xiangping Chen, Furen Xu, Yuan Huang, Xiaocong Zhou, and Zibin Zheng. "An empirical study of code reuse between GitHub and stack overflow during software development." In: *J. Syst. Softw.* 210 (2024), p. 111964. DOI: 10.1016/J.JSS.2024.111964. URL: https://doi.org/10.1016/j.jss.2024.111964.

[29] Jennifer Cobbe, Michael Veale, and Jatinder Singh. "Understanding accountability in algorithmic supply chains." In: *Proceedings of the 2023 ACM Conference on Fairness, Accountability, and Transparency, FAccT 2023, Chicago, IL, USA, June 12-15, 2023*. ACM, 2023, pp. 1186–1197. DOI: 10.1145/3593013.3594073. URL: https://doi.org/10.1145/3593013.3594073.

[30] Faith Kimberley Coslett. *Intern Poster AIBOMs in Automated Defense: Capabilities & Challenges*. Poster. 2024. URL: https://inldigitallibrary.inl.gov/sites/STI/STI/Sort_123578.pdf.

[31] CycloneDX Core Working Group. *Authoritative Guide to SBOM - Implement and optimize use of Software Bill of Materials*. 2024. URL: https://cyclonedx.org/guides/OWASP_CycloneDX-Authoritative-Guide-to-SBOM-en.pdf.

[32] CycloneDX. *Authoritative Guide to SBOM - OWASP CycloneDX Guide*. 2023. URL: https://cyclonedx.org/guides/sbom/OWASP_CycloneDX-SBOM-Guide-en.pdf/.

[33] CycloneDX. *CycloneDX Specification Overview - Recognized predicate type*. 2023. URL: https://cyclonedx.org/specification/overview/\#recognized-predicate-type/.

[34] CycloneDX. *CycloneDX Tool Center*. 2023. URL: https://cyclonedx.org/tool-center/.

[35] CycloneDX. *CycloneDX/cyclonedx-cli: CycloneDX CLI tool for SBOM analysis, merging, diffs and format conversions*. 2023. URL: https://github.com/CycloneDX/cyclonedx-cli/.

[36] CycloneDX. *CycloneDX/sbom-utility: Utility that provides an API platform for validating, querying and managing BOM data*. 2023. URL: https://github.com/CycloneDX/sbom-utility/.

[37] CycloneDX. *History*. 2024. URL: https://cyclonedx.org/about/history/.

[38] CycloneDX. *OWASP CycloneDX Software Bill of Materials (SBOM) Standard*. 2025. URL: https://cyclonedx.org.

[39] Riccardo D'Avino, Sabato Nocera, Daniele Bifolco, Federica Pepe, Massimiliano Di Penta, and Giuseppe Scanniello. *ALOHA: A(IBoM) tooL generatOr for Hugging fAce - Replication Package of the Empirical Evaluation*. DOI: 10.6084/m9.figshare.28846169. 2025. URL: https://doi.org/10.6084/m9.figshare.28846169.

[40] Riccardo D'Avino, Sabato Nocera, Daniele Bifolco, Federica Pepe, Massimiliano Di Penta, and Giuseppe Scanniello. *ALOHA: A(IBoM) tooL generatOr for Hugging fAce*. Mar. 2025. DOI: 10.5281/zenodo.15052346. URL: https://doi.org/10.5281/zenodo.15052347.

[41] Riccardo D'Avino, Sabato Nocera, Daniele Bifolco, Federica Pepe, Massimiliano Di Penta, and Giuseppe Scanniello. *ALOHA: A(IBoM) tooL generatOr for Hugging fAce*. 2025. URL: https://github.com/MSR4SBOM/ALOHA.

[42] DVC. *Data Version Control*. DVC. 2025. URL: https://dvc.org/.

[43] Ozren Dabic, Emad Aghajani, and Gabriele Bavota. "Sampling Projects in GitHub for MSR Studies." In: *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021, Madrid, Spain, May 17-19, 2021*. IEEE, 2021, pp. 560–564. DOI: 10.1109/MSR52588.2021.00074. URL: https://doi.org/10.1109/MSR52588.2021.00074.

[44] ⋆ Daniele Bifolco, Sabato Nocera, Simone Romano, Massimiliano Di Penta, Rita Francese, and Giuseppe Scanniello. "On the Accuracy of GitHub's Dependency Graph." In: *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering, EASE 2024,*

Salerno, Italy, June 18-21, 2024. ACM, 2024, pp. 242–251. DOI: 10.1145/3661167.3661175. URL: https://doi.org/10.1145/3661167.3661175.

[45] ⋆ Daniele Bifolco, Simone Romano, Sabato Nocera, Rita Francese, Giuseppe Scanniello, and Massimiliano Di Penta. "An empirical study on the accuracy of GitHub's dependency graph and the nature of its inaccuracy." In: *Inf. Softw. Technol.* 187 (2025), p. 107854. DOI: 10.1016/J.INFSOF.2025.107854. URL: https://doi.org/10.1016/j.infsof.2025.107854.

[46] Alexandre Decan, Tom Mens, and Eleni Constantinou. "On the impact of security vulnerabilities in the npm package dependency network." In: *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*. Ed. by Andy Zaidman, Yasutaka Kamei, and Emily Hill. ACM, 2018, pp. 181–191. DOI: 10.1145/3196398.3196401. URL: https://doi.org/10.1145/3196398.3196401.

[47] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. "Keep me Updated: An Empirical Study of Third-Party Library Updatability on Android." In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. Ed. by Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu. ACM, 2017, pp. 2187–2200. DOI: 10.1145/3133956.3134059. URL: https://doi.org/10.1145/3133956.3134059.

[48] Bill Doerrfeld. *Leading SBOM Standard CycloneDX Now Incorporates Machine Learning*. News Article. 2023. URL: https://cloudwars.com/cybersecurity/leading-sbom-standard-cyclonedx-now-incorporates-machine-learning/.

[49] Paul M. Duvall, Steve Matyas, and Andrew Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Signature Series (Addison-Wesley). Boston, MA: Addison-Wesley Professional, 2007. ISBN: 9780321336385.

[50] Tore Dybå, Dag I. K. Sjøberg, and Daniela S. Cruzes. "What works for whom, where, when, and why?: on the role of context in empirical software engineering." In: *2012 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '12, Lund, Sweden - September 19 - 20, 2012*. Ed. by Per Runeson, Martin Höst, Emilia Mendes, Anneliese Amschler Andrews, and Rachel Harrison. ACM, 2012, pp. 19–28. DOI: 10.1145/2372251.2372256. URL: https://doi.org/10.1145/2372251.2372256.

[51] ⋆ Riccardo D'Avino, Sabato Nocera, Daniele Bifolco, Federica Pepe, Massimiliano Di Penta, and Giuseppe Scanniello. "ALOHA: A (IBoM) tooL generatOr for Hugging fAce." In: *Proceedings of the 29th International Conference on Evaluation and Assessment in Software Engineering, EASE 2025, Istanbul, Turkey, June 17-20, 2025*. ACM, 2025, Accepted and presented (proceedings forthcoming). DOI: 10.1145/3756681.3756998. URL: https://doi.org/10.1145/3756681.3756998.

[52] Steve Easterbrook, Janice Singer, Margaret-Anne D. Storey, and Daniela E. Damian. "Selecting Empirical Methods for Software Engineering Research." In: *Guide to Advanced Empirical Software Engineering*. Ed. by Forrest Shull, Janice Singer, and Dag I. K. Sjøberg. Springer, 2008, pp. 285–311. DOI: 10.1007/978-1-84800-044-5_11. URL: https://doi.org/10.1007/978-1-84800-044-5_11.

[53] William Enck and Laurie A. Williams. "Top Five Challenges in Software Supply Chain Security: Observations From 30 Industry and Government Organizations." In: *IEEE Secur. Priv.* 20.2 (2022), pp. 96–100. DOI: 10.1109/MSEC.2022.3142338. URL: https://doi.org/10.1109/MSEC.2022.3142338.

[54] Gorkem Ercan. *Unifying Documentation and Provenance for AI and ML: A Developer's Guide to Navigating the Chaos*. Blogs. 2024. URL: https://dev.to/kitops/unifying-documentation-and-provenance-for-ai-and-ml-a-developers-guide-to-navigating-the-chaos-1n1o.

[55] Ericka Chickowski. *AI and the software supply chain: AppSec just got way more complicated*. ReversingLabs. 2023. URL: https://www.reversinglabs.com/blog/ai-further-complicates-software-supply-chain-security.

[56] European Commission. *Cyber Resilience Act - EUR-Lex - 52022PC0454 - EN - EUR-Lex*. 2022. URL: https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=celex%3A52022PC0454.

[57] European Union Agency for Cybersecurity. *Guidelines for Securing the Internet of Things - ENISA*. 2020. URL: https://www.enisa.europa.eu/publications/guidelines-for-securing-the-internet-of-things.

[58] European Union. *EU AI Act - Regulation - EU - 2024/1689 - EN - EUR-Lex*. 2024. URL: https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:32024R1689/.

[59] Hugging Face. *AutoTrain*. Hugging Face Inc. 2025. URL: https://huggingface.co/autotrain.

[60] Hugging Face. *Displaying carbon emissions for your model*. Hugging Face Inc. 2025. URL: https://huggingface.co/docs/hub/model-cards-co2#how-is-the-carbon-footprint-of-my-model-calculated-.

[61] Hugging Face. *Hub API Endpoints*. Hugging Face Inc. 2025. URL: https://huggingface.co/docs/hub/api.

[62] Hugging Face. *Hub client library*. Hugging Face Inc. 2025. URL: https://huggingface.co/docs/huggingface_hub/en/index.

[63] Hugging Face. *Model Cards — How are model tags determined?* Hugging Face Inc. 2025. URL: https://huggingface.co/docs/hub/model-cards#how-are-model-tags-determined.

[64] Hugging Face. *Model Cards — Specifying a library*. Hugging Face Inc. 2025. URL: https://huggingface.co/docs/hub/model-cards#specifying-a-library.

[65] Hugging Face. *Model Cards — Specifying a task (pipeline_tag)*. Hugging Face Inc. 2025. URL: https://huggingface.co/docs/hub/model-cards#specifying-a-task--pipelinetag-.

[66] Hugging Face. *configs*. Hugging Face Inc. 2025. URL: https://huggingface.co/docs/transformers.js/en/api/configs.

[67] Davide Fucci, Massimiliano Di Penta, Simone Romano, and Giuseppe Scanniello. "Augmenting Software Bills of Materials with Software Vulnerability Description: A Preliminary Study on GitHub." In: *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering, FSE Companion 2025, Clarion Hotel Trondheim, Trondheim, Norway, June 23-28, 2025*. Ed. by Leonardo Montecchi, Jingyue Li, Denys Poshyvanyk, and Dongmei Zhang. ACM, 2025, pp. 631–635. DOI: 10.1145/3696630.3728513. URL: https://doi.org/10.1145/3696630.3728513.

[68] Vahid Garousi, Michael Felderer, and Tuna Hacaloglu. "Software test maturity assessment and test process improvement: A multivocal literature review." In: *Inf. Softw. Technol.* 85 (2017), pp. 16–42. DOI: 10.1016/J.INFSOF.2017.01.001. URL: https://doi.org/10.1016/j.infsof.2017.01.001.

[69] Vahid Garousi, Michael Felderer, and Mika V. Mäntylä. "Guidelines for including grey literature and conducting multivocal literature reviews in software engineering." In: *Inf. Softw. Technol.* 106 (2019), pp. 101–121. DOI: 10.1016/J.INFSOF.2018.09.006. URL: https://doi.org/10.1016/j.infsof.2018.09.006.

[70] Vahid Garousi and Baris Küçük. "Smells in software test code: A survey of knowledge in industry and academia." In: *J. Syst. Softw.* 138 (2018), pp. 52–81. DOI: 10.1016/J.JSS.2017.12.013. URL: https://doi.org/10.1016/j.jss.2017.12.013.

[71] Vahid Garousi and Mika V. Mäntylä. "When and what to automate in software testing? A multi-vocal literature review." In: *Inf. Softw. Technol.* 76 (2016), pp. 92–117. DOI: 10.1016/J.INFSOF.2016.04.015. URL: https://doi.org/10.1016/j.infsof.2016.04.015.

[72] Bijit Ghosh. *AI Bill of Materials (AI BOM)*. Blogs. 2024. URL: https://medium.com/@bijit211987/ai-bill-of-materials-ai-bom-80d48f9d75e0.

[73] GitHub. *Dependabot*. 2023. URL: https://docs.github.com/en/code-security/dependabot.

[74] GitHub. *Help improve GitHub dependency graph with your feedback!* 2023. URL: https://github.com/orgs/community/discussions/43364.

[75] GitHub. *Introducing self-service SBOMs - The GitHub Blog*. 2023. URL: https://github.blog/2023-03-28-introducing-self-service-sboms.

[76] GitHub. *About the Dependency Graph*. 2025. URL: https://docs.github.com/en/code-security/supply-chain-security/understanding-your-software-supply-chain/about-the-dependency-graph.

[77] GitHub. *Dependency graph supported package ecosystems*. 2025. URL: https://docs.github.com/en/code-security/supply-chain-security/understanding-your-software-supply-chain/dependency-graph-supported-package-ecosystems.

[78] GitHub. *Troubleshooting the dependency graph*. 2025. URL: https://docs.github.com/en/enterprise-server@3.9/code-security/supply-chain-security/understanding-your-software-supply-chain/troubleshooting-the-dependency-graph\#are-there-limits-which-affect-the-dependency-graph-data.

[79] Google Online Security Blog. *Understanding the Impact of Apache Log4j Vulnerability*. 2021. URL: https://security.googleblog.com/2021/12/understanding-impact-of-apache-log4j.html.

[80] Google. *Google*. 2025. URL: https://www.google.com/.

[81] James Governor. *Introducing the AI Bill of Materials*. Blogs. 2023. URL: https://redmonk.com/jgovernor/2023/10/18/introducing-the-ai-bill-of-materials/.

[82] Greg Guest, Arwen Bunce, and Laura Johnson. "How many interviews are enough? An experiment with data saturation and variability." In: *Field methods* 18.1 (2006), pp. 59–82. DOI: 10.1177/1525822X05279903. URL: https://doi.org/10.1177/1525822X05279903.

[83] Neal Robert Haddaway, Alexandra Mary Collins, Deborah Coughlin, and Stuart Kirk. "The role of Google Scholar in evidence reviews and its applicability to grey literature searching." In: *PloS one* 10 (2015), e0138237. DOI: 10.1371/journal.pone.0138237. URL: https://doi.org/10.1371/journal.pone.0138237.

[84] Herman Hartmann and Tim Trew. "Using Feature Diagrams with Context Variability to Model Multiple Product Lines for Software Supply Chains." In: *Software Product Lines, 12th International Conference, SPLC 2008, Limerick, Ireland, September 8-12, 2008, Proceedings*. IEEE Computer Society, 2008, pp. 12–21. DOI: 10.1109/SPLC.2008.15. URL: https://doi.org/10.1109/SPLC.2008.15.

[85] IEEE. *IEEE Xplore*. 2025. URL: https://ieeexplore.ieee.org/Xplore/home.jsp/.

[86] ISO. *ISO/IEC 5962:2021 - Information technology — SPDX® Specification V2.2.1*. 2021. URL: https://www.iso.org/standard/81870.html.

[87] Nasif Imtiaz, Seaver Thorn, and Laurie A. Williams. "A comparative study of vulnerability reporting by software composition analysis tools." In: *ESEM '21: ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, Bari, Italy, October 11-15, 2021*. Ed. by Filippo Lanubile, Marcos Kalinowski, and Maria Teresa Baldassarre. ACM, 2021, 5:1–5:11. DOI: 10.1145/3475716.3475769. URL: https://doi.org/10.1145/3475716.3475769.

[88] Chadni Islam, Muhammad Ali Babar, and Surya Nepal. "A Multi-Vocal Review of Security Orchestration." In: *ACM Comput. Surv.* 52.2 (2019), 37:1–37:45. DOI: 10.1145/3305268. URL: https://doi.org/10.1145/3305268.

[89] JFrog. *End to End Security Tool for DevOps Teams - JFrog Security*. 2024. URL: https://jfrog.com/xray/.

[90] Mahmoud Jahanshahi, David Reid, and Audris Mockus. "Beyond Dependencies: The Role of Copy-Based Reuse in Open Source Software Development." In: *ACM Trans. Softw. Eng. Methodol.* (2025). DOI: 10.1145/3715907. URL: https://doi.org/10.1145/3715907.

[91] Joinup. *UUID (RFC 4122) - Universally Unique IDentifier (UUID) URN Namespace | Joinup*. 2023. URL: https://joinup.ec.europa.eu/collection/ict-standards-procurement/solution/uuid-rfc-4122-universally-unique-identifier-uuid-urn-namespace/about/.

[92] Sydney J. Freedberg Jr. *'AI-BOM' bombs: Army backs off, will demand less detailed data from AI vendors*. News Article. 2024. URL: https://breakingdefense.com/2024/04/ai-bom-bombs-army-backs-off-will-demand-less-detailed-data-from-ai-vendors/.

[93] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. Germán, and Daniela E. Damian. "The promises and perils of mining GitHub." In: *11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India*. Ed. by Premkumar T. Devanbu, Sung Kim, and Martin Pinzger. ACM, 2014, pp. 92–101. DOI: 10.1145/2597073.2597074. URL: https://doi.org/10.1145/2597073.2597074.

[94] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. "CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code." In: *IEEE Trans. Software Eng.* 28.7 (2002), pp. 654–670. DOI: 10.1109/TSE.2002.1019480. URL: https://doi.org/10.1109/TSE.2002.1019480.

[95] Diana Kelley. *Why MLBOMs Are Useful for Securing the AI/ML Supply Chain*. News Article. 2024. URL: https://www.darkreading.com/vulnerabilities-threats/mlboms-are-useful-for-securing-ai-ml-supply-chain.

[96] Nigel King. "Essential Guide to Qualitative Methods in Organizational Research." In: London: Sage, 2004. Chap. Using Templates in the Thematic Analysis of Text. DOI: https://doi.org/10.4135/9781446280119.

[97] Barbara A. Kitchenham, Pearl Brereton, David Budgen, Mark Turner, John Bailey, and Stephen G. Linkman. "Systematic literature reviews in software engineering - A systematic literature review." In: *Inf. Softw. Technol.* 51.1 (2009), pp. 7–15. DOI: 10.1016/J.INFSOF.2008.09.009. URL: https://doi.org/10.1016/j.infsof.2008.09.009.

[98] Barbara Kitchenham and Stuart Charters. *Guidelines for performing systematic literature reviews in software engineering*. 2007.

[99] Berend Kloeg, Aaron Yi Ding, Sjoerd Pellegrom, and Yury Zhauniarovich. "Charting the Path to SBOM Adoption: A Business Stakeholder-Centric Approach." In: *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security, ASIA CCS 2024, Singapore, July 1-5, 2024*. Ed. by Jianying Zhou, Tony Q. S. Quek, Debin Gao, and Alvaro A. Cárdenas. ACM, 2024. DOI: 10.1145/3634737.3637659. URL: https://doi.org/10.1145/3634737.3637659.

[100] Chris Lamb and Stefano Zacchiroli. "Reproducible Builds: Increasing the Integrity of Software Supply Chains." In: *IEEE Softw.* 39.2 (2022), pp. 62–70. DOI: 10.1109/MS.2021.3073045. URL: https://doi.org/10.1109/MS.2021.3073045.

[101] Stefano Lambiase, Gemma Catolino, Fabio Palomba, and Filomena Ferrucci. "Motivations, Challenges, Best Practices, and Benefits for Bots and Conversational Agents in Software Engineering: A Multivocal Literature Review." In: *ACM Comput. Surv.* 57.4 (2025), 93:1–93:37. DOI: 10.1145/3704806. URL: https://doi.org/10.1145/3704806.

[102] James Lee. *Model-Based Systems Engineering(MBSE) Approach to Develop an ArtificialIntelligence Bill of Material (AIBOM) forAI System Compliance Verification*. Presentation. 2024. URL: https://sercuarc.org/wp-

content/uploads/2024/09/Slides-Model-Based-Systems-Engineering-MBSE-Approach-to-Develop-an-Artificial-Intelligence-Bill-of-Material-AIBOM-for-AI-System-Compliance-Verification.pdf.

[103] Veronika Lesch, Marwin Züfle, André Bauer, Lukas Iffländer, Christian Krupitzer, and Samuel Kounev. "A literature review of IoT and CPS - What they are, and what they are not." In: *J. Syst. Softw.* 200 (2023), p. 111631. DOI: 10.1016/J.JSS.2023.111631. URL: https://doi.org/10.1016/j.jss.2023.111631.

[104] Elias Levy. "Poisoning the Software Supply Chain." In: *IEEE Secur. Priv.* 1.3 (2003), pp. 70–73. DOI: 10.1109/MSECP.2003.1203227. URL: https://doi.org/10.1109/MSECP.2003.1203227.

[105] Liqian Lim. *The Essential Guide to AI Bills of Materials (AI-BOMs)*. Blogs. 2024. URL: https://snyk.io/articles/ai-security/ai-bill-of-materials-aibom/.

[106] Yue Liu, Dawen Zhang, Boming Xia, Julia Anticev, Tunde Adebayo, Zhenchang Xing, and Moses Machao. "Blockchain-Enabled Accountability in Data Supply Chain: A Data Bill of Materials Approach." In: *IEEE International Conference on Blockchain, Blockchain 2024, Copenhagen, Denmark, August 19-22, 2024*. IEEE, 2024, pp. 557–562. DOI: 10.1109/BLOCKCHAIN62396.2024.00082. URL: https://doi.org/10.1109/Blockchain62396.2024.00082.

[107] Qinghua Lu, Liming Zhu, Xiwei Xu, Yue Liu, Zhenchang Xing, and Jon Whittle. "A Taxonomy of Foundation Model based Systems through the Lens of Software Architecture." In: *Proceedings of the IEEE/ACM 3rd International Conference on AI Engineering - Software Engineering for AI, CAIN 2024, Lisbon, Portugal, April 14-15, 2024*. Ed. by Jane Cleland-Huang, Jan Bosch, Henry Muccini, and Grace A. Lewis. ACM, 2024, pp. 1–6. DOI: 10.1145/3644815.3644956. URL: https://doi.org/10.1145/3644815.3644956.

[108] MLflow Project. *MLflow | MLflow — mlflow.org*. MLflow Project. 2025. URL: https://mlflow.org/.

[109] Saraj Singh Manes and Olga Baysal. "How often and what StackOverflow posts do developers reference in their GitHub projects?" In: *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada*. Ed. by Margaret-Anne D. Storey, Bram Adams, and Sonia Haiduc. IEEE / ACM, 2019, pp. 235–239. DOI: 10.1109/MSR.2019.00047. URL: https://doi.org/10.1109/MSR.2019.00047.

[110] Yuko McGrath, Mark Bellis, Keith Edmonds, Jim McVeigh, and Harry Sumnall. *Review of grey literature on drug prevention among young people*. National Institute for Health and Clinical Excellence London, 2006.

[111] Matthew B Miles and A Michael Huberman. *Qualitative data analysis: An expanded sourcebook*. sage, 1994.

[112] Margaret Mitchell, Simone Wu, Andrew Zaldivar, Parker Barnes, Lucy Vasserman, Ben Hutchinson, Elena Spitzer, Inioluwa Deborah Raji, and Timnit Gebru. "Model Cards for Model Reporting." In: *Proceedings of the Conference on Fairness, Accountability, and Transparency, FAT* 2019, Atlanta, GA, USA, January 29-31, 2019*. Ed. by danah boyd and Jamie H. Morgenstern. ACM, 2019, pp. 220–229. DOI: 10.1145/3287560.3287596. URL: https://doi.org/10.1145/3287560.3287596.

[113] David Mor-Ofek. *It's Time to Talk About AI/ML BOM (Artificial Intelligence Bill of Materials) And Vulnerability Management*. Blogs. 2024. URL: https://c2a-sec.com/its-time-to-talk-about-ai-ml-bom-artificial-intelligence-bill-of-materials-and-vulnerability-management/.

[114] NTIA. *Framing Software Component Transparency: Establishing a Common Software Bill of Materials (SBOM)*. 2021. URL: https://www.ntia.gov/files/ntia/publications/ntia_sbom_framing_2nd_edition_20211021.pdf.

[115] NTIA. *Roles and Benefits for SBOM Across the Supply Chain | NTIA Multistakeholder Process on Software Component Transparency | Use Cases and State of Practice Working Group*. 2021. URL: https://www.ntia.gov/files/ntia/publications/ntia_sbom_use_cases_roles_benefits-nov2019.pdf.

[116] NTIA. *SBOM Myths vs. Facts*. 2021. URL: https://www.ntia.gov/files/ntia/publications/sbom_myths_vs_facts_nov2021.pdf.

[117] NTIA. *SBOM Options and Decision Points*. 2021. URL: https://www.ntia.gov/files/ntia/publications/sbom_options_and_decision_points_20210427-1.pdf.

[118] NTIA. *Software Identification | Challenges and Guidance*. 2021. URL: https://www.ntia.gov/sites/default/files/publications/ntia_sbom_software_identity-2021mar30_0.pdf.

[119] NTIA. *Survey of Existing SBOM Formats and Standards*. 2021. URL: https://www.ntia.gov/files/ntia/publications/sbom_formats_survey-version-2021.pdf.

[120] Shripad Nadgowda. "Engram: the one security platform for modern software supply chain risks." In: *Proceedings of the Eighth International Workshop on Container Technologies and Container Clouds, WoC 2022, Quebec City, Quebec, Canada, 7 November 2022*. Ed. by Kaiwen Zhang, Abdelouahed Gherbi, and Paolo Bellavista. ACM, 2022, pp. 7–12. DOI: 10.1145/3565384.3565889. URL: https://doi.org/10.1145/3565384.3565889.

[121] National Institute of Standards and Technology. *supply chain - Glossary | CSRC — csrc.nist.gov*. 2025. URL: https://csrc.nist.gov/glossary/term/supply_chain.

[122] Danny Nebenzahl. *ML-What? Understanding the ML-Bom Concept and Uses*. Blogs. 2023. URL: https://scribesecurity.com/blog/ml-sbom/.

[123] NetworkX. *NetworkX - NetworkX documentation*. 2023. URL: https://networkx.org/.

[124] Sabato Nocera, Massimiliano Di Penta, Simone Romano, and Giuseppe Scanniello. *AIRBORNE: Augmented fIne-grained softwaRe Bill Of mateRials geNerator for stack ovErflow*. 2025. URL: https://github.com/MSR4SBOM/airborne.

[125] Sabato Nocera, Simone Romano, Massimiliano Di Penta, Rita Francese, and Giuseppe Scanniello. *On the Adoption of Software Bill of Materials in Open-source Software Projects - A Replication Package*. 2024. DOI: 10.6084/m9.figshare.25003811. URL: https://doi.org/10.6084/m9.figshare.25003811.

[126] Sabato Nocera, Simone Romano, Rita Francese, Riccardo Burlon, and Giuseppe Scanniello. "Managing Vulnerabilities in Software Projects: the Case of NTT Data." In: *49th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2023, Durres, Albania, September 6-8, 2023*. IEEE, 2023, pp. 247–253. DOI: 10.1109/SEAA60479.2023.00046. URL: https://doi.org/10.1109/SEAA60479.2023.00046.

[127] ⋆ Sabato Nocera, Massimiliano Di Penta, Simone Romano, and Giuseppe Scanniello. "AIRBORNE: Augmented fIne-grained softwaRe Bill Of mateRials geNerator for stack ovErflow." In: *Submitted to the 48th IEEE/ACM International Conference on Software Engineering*.

[128] OWASP. *A06 Vulnerable and Outdated Components - OWASP Top 10:2021*. 2021. URL: https://owasp.org/Top10/A06_2021-Vulnerable_and_Outdated_Components/.

[129] Helen Oakley. *Building a Stronger Software Supply Chain with SBOM and AIBOM: Insights from Recent Industry Events*. Social Media Post. 2024. URL: https://www.linkedin.com/pulse/building-stronger-software-supply-chain-sbom-aibom-insights-oakley-g6p4c.

[130] Helen Oakley. *Securing AI Ecosystems: The Critical Role of AIBOM in Mitigating Software Supply Chain Risks*. Presentation. 2024. URL: https://csrc.nist.gov/csrc/media/Presentations/2024/securing-ai-ecosystems-the-critical-role-of-aibom/images-media/20240917_Securing%20AI%20Ecosystems.pdf.

[131] Rodney T Ogawa and Betty Malen. "Towards rigor in reviews of multivocal literatures: Applying the exploratory case study method." In: *Review of educational research* 61.3 (1991), pp. 265–286. DOI: 10.3102/00346543061003265. URL: https://doi.org/10.3102/00346543061003265.

[132] Open Text Corporation. *OpenText Fortify Static Code Analyzer | Static Code Analysis Security*. 2024. URL: https://www.opentext.com/products/fortify-static-code-analyzer/.

[133] Ipek Ozkaya. "What Is Really Different in Engineering AI-Enabled Systems?" In: *IEEE Softw.* 37.4 (2020), pp. 3–6. DOI: 10.1109/MS.2020.2993662. URL: https://doi.org/10.1109/MS.2020.2993662.

[134] PYPL. *2025 PYPL Index*. 2025. URL: https://pypl.github.io/PYPL.html.

[135] Paperpile. *The best research databases for computer science*. 2025. URL: https://paperpile.com/g/research-databases-computer-science/.

[136] Ivan Pashchenko, Duc-Ly Vu, and Fabio Massacci. "A Qualitative Study of Dependency Management and Its Security Implications." In: *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*. Ed. by Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna. ACM, 2020, pp. 1513–1531. DOI: 10.1145/3372297.3417232. URL: https://doi.org/10.1145/3372297.3417232.

[137] M Patton. *Qualitative Research & Evaluation Methods*. Sage, 2015.

[138] Severi Peltonen, Luca Mezzalira, and Davide Taibi. "Motivations, benefits, and issues for adopting Micro-Frontends: A Multivocal Literature Review." In: *Inf. Softw. Technol.* 136 (2021), p. 106571. DOI: 10.1016/J.INFSOF.2021.106571. URL: https://doi.org/10.1016/j.infsof.2021.106571.

[139] Federica Pepe, Vittoria Nardone, Antonio Mastropaolo, Gabriele Bavota, Gerardo Canfora, and Massimiliano Di Penta. "How do Hugging Face Models Document Datasets, Bias, and Licenses? An Empirical Study." In: *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension, ICPC 2024, Lisbon, Portugal, April 15-16, 2024*. Ed. by Igor Steinmacher, Mario Linares-Vásquez, Kevin Patrick Moran, and Olga Baysal. ACM,

2024, pp. 370–381. DOI: 10.1145/3643916.3644412. URL: https://doi.org/10.1145/3643916.3644412.

[140] Federica Pepe, Fiorella Zampetti, Antonio Mastropaolo, Gabriele Bavota, and Massimiliano Di Penta. "A Taxonomy of Self-Admitted Technical Debt in Deep Learning Systems." In: *IEEE International Conference on Software Maintenance and Evolution, ICSME 2024, Flagstaff, AZ, USA, October 6-11, 2024*. IEEE, 2024, pp. 388–399. DOI: 10.1109/ICSME58944.2024.00043. URL: https://doi.org/10.1109/ICSME58944.2024.00043.

[141] Kai Petersen and Claes Wohlin. "Context in industrial software engineering research." In: *Proceedings of the Third International Symposium on Empirical Software Engineering and Measurement, ESEM 2009, October 15-16, 2009, Lake Buena Vista, Florida, USA*. IEEE Computer Society, 2009, pp. 401–404. DOI: 10.1109/ESEM.2009.5316010. URL: https://doi.org/10.1109/ESEM.2009.5316010.

[142] Colin Potts. "Software-Engineering Research Revisited." In: *IEEE Softw.* 10.5 (1993), pp. 19–28. DOI: 10.1109/52.232392. URL: https://doi.org/10.1109/52.232392.

[143] Gede Artha Azriadi Prana, Abhishek Sharma, Lwin Khin Shar, Darius Foo, Andrew E. Santosa, Asankhaya Sharma, and David Lo. "Out of sight, out of mind? How vulnerable dependencies affect open-source projects." In: *Empir. Softw. Eng.* 26.4 (2021), p. 59. DOI: 10.1007/S10664-021-09959-3. URL: https://doi.org/10.1007/s10664-021-09959-3.

[144] PyGithub. *PyGithub documentation*. 2023. URL: https://pygithub.readthedocs.io/en/latest/introduction.html.

[145] Petar Radanliev, Omar Santos, and Alistair Brandon-Jones. "Capability hardware enhanced instructions and artificial intelligence bill of materials in trustworthy artificial intelligence systems: Analyzing cybersecurity threats, exploits, and vulnerabilities in new software bills of materials with artificial intelligence." In: *J. Def. Model. Simul.* (2024), p. 15485129241267919. DOI: 10.1177/15485129241267919. URL: https://doi.org/10.1177/15485129241267919.

[146] Petar Radanliev, Omar Santos, and Alistair Brandon-Jones. "Capability hardware enhanced instructions and artificial intelligence bill of materials in trustworthy artificial intelligence systems: analyzing cybersecurity threats, exploits, and vulnerabilities in new software bills of materials with artificial intelligence." In: *J. Def. Model. Simul.* (2024), p. 15485129241267919. DOI: 10.1177/15485129241267919. URL: https://www.doi.org/10.1177/15485129241267919.

[147] Adib Bin Rashid and M. D. Ashfakul Karim Kausik. "AI revolutionizing industries worldwide: A comprehensive overview of its diverse applications." In: *Hybrid Advances* 7 (2024), p. 100277. ISSN: 2773-207X. DOI: 10.1016/j.hybadv.2024.100277. URL: https://doi.org/10.1016/j.hybadv.2024.100277.

[148] Leonard Richardson. *BeautifulSoup*. 2023. URL: https://www.crummy.com/software/BeautifulSoup/.

[149] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (4th Edition)*. Pearson, 2020. ISBN: 9780134610993. URL: http://aima.cs.berkeley.edu/.

[150] SPDX. *Clause 11: Relationship between SPDX Elements Information - specification v2.2.2*. 2023. URL: https://spdx.github.io/spdx-spec/v2-draft/relationships-between-SPDX-elements/.

[151] SPDX. *Open Source Tools - SPDX*. 2023. URL: https://spdx.dev/spdx-tools/.

[152] SPDX. *SPDX Online Tool*. 2023. URL: https://tools.spdx.org/app/convert/.

[153] SPDX. *Overview - SPDX*. 2024. URL: https://spdx.dev/about/overview/.

[154] SPDX. *SPDX - Linux Foundation Projects Site*. 2025. URL: https://spdx.dev.

[155] ⋆ Sabato Nocera, Massimiliano Di Penta, Fatima Ahmed, Simone Romano, and Giuseppe Scanniello. "What We Know about AIBOMs: Results from a Multivocal Literature Review on Artificial Intelligence Bill of Materials." In: *ACM Trans. Softw. Eng. Methodol.* (Under submission).

[156] ⋆ Sabato Nocera, Massimiliano Di Penta, Rita Francese, Simone Romano, and Giuseppe Scanniello. "If it's not SBOM, then what? How Italian Practitioners Manage the Software Supply Chain." In: *IEEE International Conference on Software Maintenance and Evolution, ICSME 2024, Flagstaff, AZ, USA, October 6-11, 2024*. IEEE, 2024, pp. 730–740. DOI: 10.1109/ICSME58944.2024.00077. URL: https://doi.org/10.1109/ICSME58944.2024.00077.

[157] ⋆ Sabato Nocera, Simone Romano, Massimiliano Di Penta, Rita Francese, and Giuseppe Scanniello. "Software Bill of Materials Adoption: A Mining Study from GitHub." In: *IEEE International Conference on Software Maintenance and Evolution, ICSME 2023, Bogotá, Colombia, October 1-6, 2023*. IEEE, 2023, pp. 39–49. DOI: 10.1109/ICSME58846.2023.00016. URL: https://doi.org/10.1109/ICSME58846.2023.00016.

[158] ⋆ Sabato Nocera, Simone Romano, Massimiliano Di Penta, Rita Francese, and Giuseppe Scanniello. "On the adoption of software bill of materials in open-source software projects." In: *J. Syst. Softw.* 230 (2025), p. 112540. DOI: 10.1016/J.JSS.2025.112540. URL: https://doi.org/10.1016/j.jss.2025.112540.

[159] Omar Santos. *Artificial Intelligence Bill-of-Materials (AI BOMs): Ensuring AI Transparency and Traceability*. Blogs. 2023. URL: https://becomingahacker.org/artificial-intelligence-bill-of-materials-ai-boms-ensuring-ai-transparency-and-traceability-82322643bd2a.

[160] Omar Santos. *Understanding the SPDX 3.0 AI BOM Support*. Blogs. 2024. URL: https://becomingahacker.org/understanding-the-spdx-3-0-ai-bom-support-7f3dbdd28345.

[161] Giuseppe Scanniello, Massimiliano Di Penta, Simone Romano, Rita Francese, Sabato Nocera, Pietro Cassieri, Daniele Bifolco, and Fiorella Zampetti. "MSR4SBOM: Mining Software Repositories for enhanced Software Bills of Materials." In: *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2024, Barcelona, Spain, October*

24-25, 2024. Ed. by Xavier Franch, Maya Daneva, Silverio Martínez-Fernández, and Luigi Quaranta. ACM, 2024, pp. 589–593. DOI: 10.1145/3674805.3695390. URL: https://doi.org/10.1145/3674805.3695390.

[162] Scopus. *Scopus preview - Scopus - Welcome to Scopus*. 2025. URL: https://www.scopus.com/home.uri/.

[163] Carolyn B. Seaman. "Qualitative Methods in Empirical Studies of Software Engineering." In: *IEEE Trans. Software Eng.* 25.4 (1999), pp. 557–572. DOI: 10.1109/32.799955. URL: https://doi.org/10.1109/32.799955.

[164] Yuichi Semura, Norihiro Yoshida, Eunjong Choi, and Katsuro Inoue. "CCFinderSW: Clone Detection Tool with Flexible Multilingual Tokenization." In: *24th Asia-Pacific Software Engineering Conference, APSEC 2017, Nanjing, China, December 4-8, 2017*. Ed. by Jian Lv, He Jason Zhang, Mike Hinchey, and Xiao Liu. IEEE Computer Society, 2017, pp. 654–659. DOI: 10.1109/APSEC.2017.80. URL: https://doi.org/10.1109/APSEC.2017.80.

[165] Meixue Si, Shidong Pan, Dianshu Liao, Xiaoyu Sun, Zhen Tao, Wenchang Shi, and Zhenchang Xing. *A Solution toward Transparent and Practical AI Regulation: Privacy Nutrition Labels for Open-source Generative AI-based Applications*. Pre-print. 2024. URL: https://arxiv.org/abs/2407.15407.

[166] Luis Soeiro, Thomas Robert, and Stefano Zacchiroli. "Wild SBOMs: a Large-scale Dataset of Software Bills of Materials from Public Code." In: *22nd IEEE/ACM International Conference on Mining Software Repositories, MSR@ICSE 2025, Ottawa, ON, Canada, April 28-29, 2025*. IEEE, 2025, pp. 164–168. DOI: 10.1109/MSR66628.2025.00036. URL: https://doi.org/10.1109/MSR66628.2025.00036.

[167] Davide Spadini, Maurício Finavaro Aniche, and Alberto Bacchelli. "PyDriller: Python framework for mining software repositories." In: *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*. Ed. by Gary T. Leavens, Alessandro Garcia,

and Corina S. Pasareanu. ACM, 2018, pp. 908–911. DOI: 10.1145/3236024.3264598. URL: https://doi.org/10.1145/3236024.3264598.

[168] Stack Overflow. *2022 Developer Survey*. 2022. URL: https://survey.stackoverflow.co/2022.

[169] Trevor Stalnaker, Nathan Wintersgill, Oscar Chaparro, Laura A Heymann, Massimiliano Di Penta, Daniel M German, and Denys Poshyvanyk. *The ML Supply Chain in the Era of Software 2.0: Lessons Learned from Hugging Face*. Pre-print. 2025. URL: https://arxiv.org/abs/2502.04484.

[170] Trevor Stalnaker, Nathan Wintersgill, Oscar Chaparro, Massimiliano Di Penta, Daniel M. Germán, and Denys Poshyvanyk. "BOMs Away! Inside the Minds of Stakeholders: A Comprehensive Study of Bills of Materials for Software Systems." In: *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 2024, 44:1–44:13. DOI: 10.1145/3597503.3623347. URL: https://doi.org/10.1145/3597503.3623347.

[171] Stephen Hendrick. *Software Bill of Materials (SBOM) and Cybersecurity Readiness*. Foreword by Jim Zemlin. The Linux Foundation. 2022. URL: https://www.linuxfoundation.org/research/the-state-of-software-bill-of-materials-sbom-and-cybersecurity-readiness/.

[172] Klaas-Jan Stol and Brian Fitzgerald. "Guidelines for Conducting Software Engineering Research." In: *Contemporary Empirical Methods in Software Engineering*. Ed. by Michael Felderer and Guilherme Horta Travassos. Springer, 2020, pp. 27–62. DOI: 10.1007/978-3-030-32489-6_2. URL: https://doi.org/10.1007/978-3-030-32489-6_2.

[173] Anselm Strauss, Juliet Corbin, et al. *Basics of qualitative research*. Vol. 15. sage Newbury Park, CA, 1990.

[174] Amjed Tahir, Shawn Rasheed, Jens Dietrich, Negar Hashemi, and Lu Zhang. "Test flakiness' causes, detection, impact and responses: A multivocal review." In: *J. Syst. Softw.* 206 (2023), p. 111837. DOI: 10.1016/J.JSS.2023.

111837. URL: https://doi.org/10.1016/j.jss.2023.111837.

[175] Sema Team. *Generative AI Bill of Materials (GBOM): A Primer | Sema*. Blogs. 2023. URL: https://www.semasoftware.com/blog/generative-ai-bill-of-materials-gbom-tm-a-primer.

[176] Wiz Experts Team. *AI-BOM: Building an AI-Bill of Materials*. Blogs. 2025. URL: https://www.wiz.io/academy/ai-bom-ai-bill-of-materials.

[177] Tenable. *Nessus Vulnerability Scanner: Network Security Solution*. 2024. URL: https://www.tenable.com/products/nessus/.

[178] The Linux Foundation. *A Summary of Census II: Open Source Software Application Libraries the World Depends On - Linux Foundation*. 2022. URL: https://www.linuxfoundation.org/blog/blog/a-summary-of-census-ii-open-source-software-application-libraries-the-world-depends-on.

[179] Veronica A Thurmond. "The point of triangulation." In: *Journal of nursing scholarship* 33.3 (2001), pp. 253–258. DOI: 10.1111/j.1547-5069.2001.00253.x. URL: https://doi.org/10.1111/j.1547-5069.2001.00253.x.

[180] Eric Tooley and Erin Havens. *A Smarter Quieter Dependabot*. 2023. URL: https://github.blog/2023-01-12-a-smarter-quieter-dependabot/.

[181] Santiago Torres-Arias, Hammad Afzali, Trishank Karthik Kuppusamy, Reza Curtmola, and Justin Cappos. "in-toto: Providing farm-to-table guarantees for bits and bytes." In: *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*. Ed. by Nadia Heninger and Patrick Traynor. USENIX Association, 2019, pp. 1393–1410. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/torres-arias.

[182] Santiago Torres-Arias, Daniel E. Geer, and John Speed Meyers. "A Viewpoint on Knowing Software: Bill of Materials Quality When You See It." In: *IEEE Secur. Priv.* 21.6 (2023), pp. 50–54. DOI: 10.1109/MSEC.2023.3315887. URL: https://doi.org/10.1109/MSEC.2023.3315887.

[183] Adelin Travers, Michael Brown, and Heidy Khlaaf. *Our response to the US Army's RFI on developing AIBOM tools | Trail of Bits Blog*. Blogs. 2024. URL: https://www.trailofbits.com/documents/PEO_AIBOM_RFI_Response.pdf.

[184] Kimberly Underwood. *U.S. Army Is Considering AI Bill of Materials | AFCEA International*. News Article. 2023. URL: https://www.afcea.org/signal-media/cyber-edge/us-army-considering-ai-bill-materials.

[185] United States Department of Commerce. *The Minimum Elements For a Software Bill of Materials (SBOM)*. 2021. URL: https://www.ntia.doc.gov/files/ntia/publications/sbom_minimum_elements_report.pdf.

[186] United States Government. *Executive Order on Improving the Nation's Cybersecurity | The White House*. 2021. URL: https://bidenwhitehouse.archives.gov/briefing-room/presidential-actions/2025/01/16/executive-order-on-strengthening-and-promoting-innovation-in-the-nations-cybersecurity/.

[187] Maike Vollstedt and Sebastian Rezat. "An introduction to grounded theory with a special focus on axial coding and the coding paradigm." In: *Compendium for early career researchers in mathematics education* 13.1 (2019), pp. 81–100. DOI: 10.1007/978-3-030-15636-7_4. URL: https://doi.org/10.1007/978-3-030-15636-7_4.

[188] Duc-Ly Vu, Ivan Pashchenko, Fabio Massacci, Henrik Plate, and Antonino Sabetta. "Towards Using Source Code Repositories to Identify Software Supply Chain Attacks." In: *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*. Ed. by Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna. ACM, 2020, pp. 2093–2095. DOI: 10.

1145/3372297.3420015. URL: https://doi.org/10.1145/3372297.3420015.

[189] Shenao Wang, Yanjie Zhao, Xinyi Hou, and Haoyu Wang. "Large Language Model Supply Chain: A Research Agenda." In: *ACM Trans. Softw. Eng. Methodol.* 34.5 (2025), 147:1–147:46. DOI: 10.1145/3708531. URL: https://doi.org/10.1145/3708531.

[190] David Gray Widder and Dawn Nafus. "Dislocated accountabilities in the "AI supply chain": Modularity and developers' notions of responsibility." In: *Big Data & Society* 10.1 (2023), p. 20539517231177620.

[191] Laurie Williams, Giacomo Benedetti, Sivana Hamer, Ranindya Paramitha, Imranur Rahman, Mahzabin Tamanna, Greg Tystahl, Nusrat Zahan, Patrick Morrison, Yasemin Acar, et al. "Research directions in software supply chain security." In: *ACM Trans. Softw. Eng. Methodol.* 34 (2025), pp. 1–38. DOI: 10.1145/3714464. URL: https://dl.acm.org/doi/10.1145/3714464.

[192] Willy Leichter. *From SBOM to AI-BOM: Rethinking Supply Chain Security in the AI Era*. PointGuard AI. 2025. URL: https://www.pointguardai.com/blog/from-sbom-to-ai-bom-rethinking-supply-chain-security-in-the-ai-era.

[193] Claes Wohlin. "Guidelines for snowballing in systematic literature studies and a replication in software engineering." In: *18th International Conference on Evaluation and Assessment in Software Engineering, EASE '14, London, England, United Kingdom, May 13-14, 2014*. Ed. by Martin J. Shepperd, Tracy Hall, and Ingunn Myrtveit. ACM, 2014, 38:1–38:10. DOI: 10.1145/2601248.2601268. URL: https://doi.org/10.1145/2601248.2601268.

[194] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering, Second Edition*. Springer, 2024. ISBN: 978-3-662-69305-6. DOI: 10.1007/978-3-662-69306-3. URL: https://doi.org/10.1007/978-3-662-69306-3.

[195] Boming Xia, Tingting Bi, Zhenchang Xing, Qinghua Lu, and Liming Zhu. "An Empirical Study on Software Bill of Materials: Where We Stand and the Road Ahead." In: *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 2630–2642. DOI: 10.1109/ICSE48619.2023.00219. URL: https://doi.org/10.1109/ICSE48619.2023.00219.

[196] Boming Xia, Dawen Zhang, Yue Liu, Qinghua Lu, Zhenchang Xing, and Liming Zhu. "Trust in Software Supply Chains: Blockchain-Enabled SBOM and the AIBOM Future." In: *IEEE/ACM International Workshop on Engineering and Cybersecurity of Critical Systems and Second IEEE/ACM International Workshop on Software Vulnerability, EnCyCriS/SVM 2024, Lisbon, Portugal, April 15, 2024*. ACM / IEEE, 2024, pp. 12–19. DOI: 10.1145/3643662.3643957. URL: https://doi.org/10.1145/3643662.3643957.

[197] Sheng Yu, Wei Song, Xunchao Hu, and Heng Yin. "On the Correctness of Metadata-Based SBOM Generation: A Differential Analysis Approach." In: *54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2024, Brisbane, Australia, June 24-27, 2024*. IEEE, 2024, pp. 29–36. DOI: 10.1109/DSN58291.2024.00018. URL: https://doi.org/10.1109/DSN58291.2024.00018.

[198] Nusrat Zahan, Elizabeth Lin, Mahzabin Tamanna, William Enck, and Laurie A. Williams. "Software Bills of Materials Are Required. Are We There Yet?" In: *IEEE Secur. Priv.* 21.2 (2023), pp. 82–88. DOI: 10.1109/MSEC.2023.3237100. URL: https://doi.org/10.1109/MSEC.2023.3237100.

[199] Nusrat Zahan, Thomas Zimmermann, Patrice Godefroid, Brendan Murphy, Chandra Shekhar Maddila, and Laurie A. Williams. "What are Weak Links in the npm Supply Chain?" In: *44th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2022, Pittsburgh, PA, USA, May 22-24, 2022*. IEEE, 2022, pp. 331–340. DOI: 10.1109/ICSE-SEIP55303.2022.9794068. URL: https://doi.org/10.1109/ICSE-SEIP55303.2022.9794068.

[200] Rodrigo Elizalde Zapata, Raula Gaikovina Kula, Bodin Chinthanet, Takashi Ishio, Kenichi Matsumoto, and Akinori Ihara. "Towards Smoother Library Migrations: A Look at Vulnerable Dependency Migrations at Function Level for npm JavaScript Packages." In: *2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018*. IEEE Computer Society, 2018, pp. 559–563. DOI: 10.1109/ICSME.2018.00067. URL: https://doi.org/10.1109/ICSME.2018.00067.

[201] Dawen Zhang, Boming Xia, Yue Liu, Xiwei Xu, Thong Hoang, Zhenchang Xing, Mark Staples, Qinghua Lu, and Liming Zhu. "Privacy and Copyright Protection in Generative AI: A Lifecycle Perspective." In: *Proceedings of the IEEE/ACM 3rd International Conference on AI Engineering - Software Engineering for AI, CAIN 2024, Lisbon, Portugal, April 14-15, 2024*. Ed. by Jane Cleland-Huang, Jan Bosch, Henry Muccini, and Grace A. Lewis. ACM, 2024, pp. 92–97. DOI: 10.1145/3644815.3644952. URL: https://doi.org/10.1145/3644815.3644952.

[202] Lida Zhao, Sen Chen, Zhengzi Xu, Chengwei Liu, Lyuye Zhang, Jiahui Wu, Jun Sun, and Yang Liu. "Software Composition Analysis for Vulnerability Detection: An Empirical Study on Java Projects." In: *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*. Ed. by Satish Chandra, Kelly Blincoe, and Paolo Tonella. ACM, 2023, pp. 960–972. DOI: 10.1145/3611643.3616299. URL: https://doi.org/10.1145/3611643.3616299.

[203] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. "Small World with High Risks: A Study of Security Threats in the npm Ecosystem." In: *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*. Ed. by Nadia Heninger and Patrick Traynor. USENIX Association, 2019, pp. 995–1010. URL: https://www.usenix.

org / conference / usenixsecurity19 / presentation / zimmerman.

[204] anchore. *anchore*. 2024. URL: https://github.com/anchore/anchore/.

[205] anchore. *grype*. 2024. URL: https://github.com/anchore/grype/.

[206] anchore. *syft*. 2024. URL: https://github.com/anchore/syft/.

[207] eWeek. *Heartbleed SSL Flaw's True Cost Will Take Time to Tally*. 2014. URL: https://www.eweek.com/security/heartbleed-ssl-flaw-s-true-cost-will-take-time-to-tally/.

[208] nan. *Bringing in the BoM Squad: Defining and Generating AI Bill of Materials - Securin*. Blogs. 2024. URL: https://www.securin.io/articles/bringing-in-the-bom-squad-defining-and-generating-ai-bill-of-materials/.

[209] nan. *Public and/or Shared AI Models Cannot be Trusted Until an AI Bill of Materials Become the Norm*. Blogs. 2024. URL: https://www.pivotpointsecurity.com/public-and-or-shared-ai-models-cannot-be-trusted-until-an-ai-bill-of-materials-become-the-norm/.

[210] nan. *AI - SPDX*. Blogs. 2025. URL: https://spdx.dev/learn/areas-of-interest/ai/.

[211] nan. *CycloneDX v1,6 JSON Reference*. Documentation. 2025. URL: https://cyclonedx.org/docs/.

[212] nan. *Machine Learning Bill of Materials (ML-BOM) | CycloneDX*. Blogs. 2025. URL: https://cyclonedx.org/capabilities/mlbom/.

[213] in toto. *attestation/README.md at main · in-toto/attestation*. 2023. URL: https://github.com/in-toto/attestation/blob/main/README.md.

[214] in toto. *attestation/spec at v0.1.0 · in-toto/attestation*. 2023. URL: https://github.com/in-toto/attestation/tree/v0.1.0/spec/#statement.