Configurable SBOM and Legal Notice Exports for SCA Tool

BACHELOR THESIS

Robin Neubauer

Submitted on 1 October 2025

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

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

Declaration of Originality

I confirm that the submitted thesis is original work and was written by me. Appropriate credit has been given where reference has been made to the work of others. The thesis was not examined before, nor has it been published. The submitted electronic version of the thesis matches the printed version.

Artificial Intelligence (AI) tools ChatGPT (OpenAI) and Claude AI (Anthropic) were used for language editing, source summarization, and code development assistance. All AI-generated content was reviewed, verified, and edited by the author. The ideas, analysis, conclusions, and final interpretations presented in this thesis remain entirely my own work.

Erlangen, 1 October 2025

License

Erlangen, 1 October 2025

Abstract

Modern software development relies heavily on third-party components, creating transparency deficits in software supply chains. A Software Bill of Materials (SBOM) addresses this challenge by documenting component inventories. However, existing generation tools provide rigid outputs that cannot accommodate the diverse requirements of organizational stakeholders. This thesis develops a configurable SBOM export system that enables template-driven generation across multiple formats with embedded legal notices. The objective was to create a system that addresses the configuration gap in existing SBOM tools by providing accessible template management for organizational governance while maintaining technical flexibility for specialized use cases. The implementation employs a modular architecture for multi-format support (SPDX v2.3/v3.0.1, CycloneDX v1.5/v1.6), template management, and user interfaces. The system integrates within the SCA Tool infrastructure while introducing new capabilities such as customizable dependency filtering, vulnerability data inclusion, and license text integration. By doing so, this work extends SCA Tool with a practical and flexible export solution that enhances software supply chain transparency and compliance readiness.

Contents

List of Figures

List of Tables

Acronyms

API	Application Programming Interface
BOM	Bill of Materials
BSI	German Federal Office for Information Security
CDXA	CycloneDX Attestations
CISA	Cybersecurity and Infrastructure Security Agency
CPE	Common Platform Enumeration
CRA	Cyber Resilience Act
CRUD	Create, Read, Update, Delete
CVE	Common Vulnerabilities and Exposures
CVSS	Common Vulnerability Scoring System
CWE	Common Weakness Enumeration
EU	European Union
FDA	Food and Drug Administration
FLOSS	Free/Libre Open Source Software
HTTP	HyperText Transfer Protocol
IEC	International Electrotechnical Commission
ISO	International Organization for Standardization
IT	Information Technology
JPA	Jakarta Persistence API
JSF	JSON Signature Format
JSON	JavaScript Object Notation

JSON-LD	JavaScript Object Notation for Linked Data
MD5	Message Digest 5
METI	Ministry of Economy, Trade and Industry
NPM	Node Package Manager
NTIA	National Telecommunications and Information Administration
ORT	OSS Review Toolkit
OSS	open source software
OSV	Open Source Vulnerability
OWASP	Open Worldwide Application Security Project
PURL	Package Uniform Resource Locator
PyPI	Python Package Index
RDF	Resource Description Framework
REST	Representational State Transfer
RFC	Request for Comments
SAE	Society of Automotive Engineers
SBOM	Software Bill of Materials
SCA	Software Composition Analysis
SHA	Secure Hash Algorithm
SPDX	Software Package Data Exchange
SWID	Software Identification
UI	user interface
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
UUID	Universally Unique Identifier
VEX	Vulnerability Exploitability eXchange
XML	Extensible Markup Language
XMLSIG	XML Signature
YAML	YAML Ain't Markup Language

1 Introduction

Modern software development operates on a fundamental paradox. While many applications utilize thousands of third-party components to achieve unprecedented development velocity, the majority of organizations remain unaware of the composition and provenance of their own software products. This opacity in an increasingly interconnected ecosystem has turned dependency management into a core concern for transparency and risk mitigation in software ecosystems.

1.1 The Software Supply Chain Crisis and the Transparency Deficit

According to a report on the state of the software supply chain by Sonatype (2021), software supply chain attacks have surged, with a 650% year-over-year increase in 2021, fundamentally transforming how the technology industry approaches security and transparency. This growth in supply chain vulnerabilities has transformed cybersecurity from a peripheral concern to a central strategic imperative for organizations worldwide.

Recent high-profile incidents have demonstrated the catastrophic potential of these attacks. The SolarWinds 2020 breach exemplifies this threat, where attackers compromised the company's Orion network management software through sophisticated manipulation of the software supply chain (Martínez, J. & Durán, J.M., 2021). The scale of this attack was unprecedented, infecting more than 18,000 customers and 40 public entities from various sectors, including government entities, technology companies, insurance companies, financial companies, retail companies, and organizations from all continents (Martínez, J. & Durán, J.M., 2021).

The underlying challenge stems from modern software development practices, where 85% to 97% of the code in applications or tools from software providers comes from the reuse of open source code frameworks, software repositories, and third-party Application Programming Interfaces (APIs) (Martínez, J. & Durán, J.M., 2021). This extensive dependency on external components, when performed

without employing software composition analysis tools, creates a critical transparency deficit.

This operational blindness in the security, compliance, and risk management functions creates extended exposure periods and increased security risks. Organizations lacking comprehensive visibility into their software composition face particularly severe challenges during such crises, forcing costly emergency remediation efforts while operating under the assumption that a compromise has occurred.

A Software Bill of Materials (SBOM) addresses this transparency deficit by providing a comprehensive inventory of all software components, dependencies, and their hierarchical relationships within an application (Alrich T. et al., 2021).

1.2 The Configuration Challenge

The recognition of these transparency deficits has prompted significant regulatory action. The US Executive Order 14028 mandates SBOM requirements for federal software procurement, while the European Union's Cyber Resilience Act emphasizes supply chain transparency as fundamental to cybersecurity. Industry response has been substantial, with 90% of surveyed organizations initiating SBOM implementation efforts (Xia et al., 2023). However, creating tailored SBOM documents that meet the diverse requirements of various stakeholders represents a significant challenge.

The core challenge lies not simply in generating SBOMs, but in generating the appropriate documentation for specific stakeholders and use cases. Consider a critical vulnerability discovered in a widely used cryptographic library. The security team requires comprehensive technical data, including affected component versions, dependency tree locations, cryptographic hashes, and vulnerability correlation data, to assess exposure and cascading effects. Executive leadership requires streamlined documentation that focuses on business impact, including affected product counts, criticality ratings, and remediation timelines. External customers require confirmation of deployment impact without exposure of internal architecture details or proprietary component identifiers. A single, static SBOM cannot simultaneously serve these divergent information requirements. This limitation was also observed by Stalnaker et al. (2024), who highlight the trade-off between completeness and data privacy and propose tailoring SBOM fields through defined information levels.

This thesis addresses the critical need for configurable export systems that accommodate diverse organizational requirements while maintaining compliance with established standards. The primary objective is to develop a sophisticated, configurable export system for SBOMs with integrated legal notices within

SCA Tool[1], enabling configurable inclusion of copyright statements, license texts, and legal attributions directly within SBOM documents. This system supports customization of information content based on intended use case and audience, supporting multiple SBOM standards and versions for broad compatibility. The integration into the SCA Tool user interface (UI) simplifies the export process for non-technical users while preserving the configurability that technical users require for specialized workflows.

By addressing current limitations in configurability and integration, this thesis aims to contribute to more transparent, secure, and legally compliant software development practices.

1.3 Thesis Structure

The thesis is structured into eight chapters, addressing the problem systematically. Following this introduction, Chapter 2 provides essential background on SBOM concepts and standards necessary for understanding the research context. Chapter 3 builds upon this foundation by reviewing existing tools and identifying critical gaps in configurability that motivate the research approach. Based on this analysis and the high-level research objective outlined above, Chapter 4 defines the functional and non-functional requirements for the configurable export system within SCA Tool. The solution is presented through Chapter 5, which outlines the system architecture, and Chapter 6, which details the design decisions and implementation approach. In Chapter 7, the effectiveness of the solution is assessed by systematically evaluating the implemented system against the defined functional and non-functional requirements. Finally, the thesis concludes with Chapter 8, which summarizes the work and provides an outlook on future work.

[1]SCA Tool is a SCA Cloud Service for open source governance, license compliance, SBOM management, and supply chain security. https://app.scatool.com

2 Fundamentals

This chapter establishes the foundational concepts necessary for understanding SBOMs and their role in modern software development.

2.1 Software Bill of Materials: Core Concepts

SBOM constitutes a comprehensive inventory that systematically documents all software components within a product or system. BSI (2024) specifically defines it as ”a machine-processable file containing supply chain relationships and details of the components used in a software product”. This formal artifact functions analogously to a manufacturing bill of materials, providing transparency into the software supply chain through detailed component documentation.

Essential components of any SBOM include standardized component identification through unique identifiers such as Package Uniform Resource Locator (PURL), Common Platform Enumeration (CPE), or Software Identification (SWID) tags. Version information must be precisely documented, as even minor differences can have a significant impact on security and compatibility. Dependency relationships require comprehensive mapping, capturing both direct dependencies explicitly declared by developers and transitive dependencies automatically resolved through package managers. Metadata such as author and timestamp establish provenance and support updates (NTIA, 2021).

2.2 Practical Applications

SBOMs enable several critical capabilities that address modern software development challenges:

License Compliance: Explicit documentation of licensing conditions, obligations, and restrictions supports legal risk management (Haddad, 2024). It enables informed decision-making about component usage and distribution.

Vulnerability Management: Precise component-to-vulnerability mapping en-

ables rapid identification and remediation of security issues (OWASP Foundation, 2024). When zero-day vulnerabilities affecting widely used components are disclosed, organizations with comprehensive SBOMs can quickly determine exposure across their software portfolio in order to prioritize remediation efforts effectively.

Supply Chain Integrity: Clear documentation of component provenance enables verification of software authenticity. It also facilitates secure distribution practices. This transparency becomes essential for establishing trust relationships across software supply chains (OWASP Foundation, 2024).

2.3 SPDX: The Compliance-Focused Standard

Software Package Data Exchange (SPDX) represents the most established SBOM format, maintained by the Linux Foundation and standardized as ISO/IEC 5962: 2021. Emerging from the open-source community's need for standardized license documentation, SPDX has evolved into a comprehensive framework for software transparency (Stewart, 2022).

2.3.1 Document Structure

The analysis of SPDX document structure presented draws primarily from Stewart's comprehensive overview (Stewart, 2022), the official SPDX 2.3 specification (Linux Foundation, 2022), and the SPDX 3.0.1 specification (Linux Foundation, 2023). The SPDX architecture employs a multi-section document structure optimized for detailed software documentation.

Document Creation Information provides essential metadata about the SPDX document itself, including SPDX version information, creation timestamps, creator identification, and tool information. This metadata enables document lifecycle management and provenance tracking, ensuring forward and backward compatibility for processing tools.

Package Information forms the core of SPDX documents, containing comprehensive details about software packages. Each package entry includes unique SPDX identifiers that enable unambiguous reference throughout the document. Package naming follows standardized formats, including Package Uniform Resource Locator (URL) specification, ensuring consistency across different tools and platforms. Download locations, verification codes, and cryptographic checksums provide the means to obtain and validate packages.

File Information capabilities enable granular documentation extending to individual files within packages, including file checksums, licenses, and copyright information. This granularity can be used for complex licensing scenarios where different files within a package may have different licenses or copyright holders.

Organizations managing legacy codebases or conducting license audits benefit from this detailed attribution capability. SPDX further supports Snippet Information for denoting when portions of files originate from different sources.

Relationship definitions capture complex dependencies and contextual relationships between SPDX elements, including packages, files, and snippets. SPDX 2.x supports many relationship types, including dependency patterns, build relationships, licensing relationships, and lifecycle relationships. These relationships enable precise mapping of software composition and facilitate impact assessment across software lifecycles (Alrich T. et al., 2021).

2.4 CycloneDX: The Security-Centric Approach

CycloneDX was introduced in 2017 by the Open Worldwide Application Security Project (OWASP) community as a lightweight SBOM standard made for modern DevSecOps workflows (OWASP Foundation, 2023; OWASP Foundation & Ecma International, 2024). In contrast to the compliance-driven orientation of SPDX, CycloneDX is explicitly designed to address operational security concerns and to integrate seamlessly into security tooling ecosystems (OWASP Foundation, 2024).

The standard's design philosophy emphasizes pragmatic security practices with high levels of automation and a strong focus on risk identification (OWASP Foundation, 2024). This orientation is reflected in the format's native support for vulnerability information, attestations, and evidence documentation. CycloneDX is particularly well-suited for integration with vulnerability scanners, dependency analyzers, and security orchestration platforms.

2.4.1 Security-Optimized Features

Component documentation: CycloneDX captures security-relevant metadata as a default element of component entries. Alongside identifiers and versioning information, components can be enriched with vulnerability data, trust indicators, and assessment results. Native support for CPE and PURL identifiers enables automated correlation with vulnerability databases and integration with tools such as OWASP Dependency-Track (OWASP Foundation, 2024).

Dependency graphs: The specification models both direct and transitive dependencies, enabling organizations to analyze how vulnerabilities propagate across dependency chains. Relationships between components can be annotated with security-relevant information, and "compositions" allow authors to describe the completeness of inventories and relationships (OWASP Foundation, 2023; OWASP Foundation & Ecma International, 2024). These features facilitate sophisticated risk analysis and support cascading vulnerability assessments.

Native Vulnerability Exploitability eXchange (VEX) integration: A distinctive feature of CycloneDX is its ability to incorporate the (VEX) format directly into the SBOM. This allows producers to communicate whether specific vulnerabilities in included components are actually exploitable, thereby distinguishing between theoretical and practical risks. By embedding VEX information, CycloneDX simplifies vulnerability management workflows and reduces noise in remediation activities (OWASP Foundation, 2024).

Evidence and attestation: CycloneDX provides mechanisms for documenting evidence supporting component identification, vulnerability assessments, and compliance claims. The CycloneDX Attestation (CDXA) framework formalizes this process, while digital signatures using XMLSIG and JSF ensure the integrity and authorship of SBOM documents (OWASP Foundation, 2023; OWASP Foundation & Ecma International, 2024). These features enable robust verification and trust in shared SBOM data.

2.5 Comparative Analysis

The choice between SPDXand CycloneDX depends primarily on organizational priorities and use cases. Understanding their relative strengths and differences enables informed selection of standards.

Feature	**SPDX**	**CycloneDX**
File-level granularity	Yes	No
Native VEX integration	No	Yes
ISO standardization	Yes	No
Security attestations	No	Yes
Cryptographic signatures	Yes	Yes
Dependency graph	Yes	Yes
Multiple format support (JSON/XML/YAML)	Yes	Yes
Pedigree/provenance tracking	Yes	Yes
Service/API documentation	No	Yes

Table 2.1: Comparison of SPDX and CycloneDX features

SPDX excels in scenarios requiring detailed license documentation and regulatory compliance. Organizations in highly regulated industries or those managing complex intellectual property portfolios benefit from SPDX 's comprehensive capabilities. CycloneDX optimizes for security-centric workflows and operational efficiency. Organizations prioritizing vulnerability management and continuous security monitoring find CycloneDX's native security features more aligned with their needs.

2.6 Alternative Approaches

While SPDXand CycloneDX dominate the SBOM landscape, alternative approaches serve specific needs and contexts. Two alternative approaches to SBOMs are occasionally used but remain outside the scope of this work.

SWID Tags: Software Identification (SWID) tags, standardized under ISO/IEC 19770-2, provide XML-based identification of installed software. They are primarily designed for IT asset management, focusing on lifecycle events such as installation, patching, and removal. Their limited adoption in developer workflows and lack of rich security metadata make them unsuitable as a foundation for modern SBOM practices.

Manual Tracking: Organizations sometimes rely on manually maintained inventories such as spreadsheets or documents to record component names, versions, licenses, and suppliers. This method can be sufficient for small projects, but quickly becomes error-prone and unsustainable as complexity grows. Without automation or standardization, manual inventories cannot support large-scale transparency or reliable information exchange.

2.7 Legal Notices in Software Distribution

Legal notices serve as formal documentation of licensing information, attribution requirements, copyright statements, and legally mandated disclosures that must accompany software products that contain third-party components. According to Riehle, D. and Harutyunyan, N. (2019), common license obligations include license file provision, which requires providing the license file of each open-source component, and copyright notice provision, which requires providing all copyright notices from all files of each open-source component. The notices serve as both legal instruments and ethical acknowledgments, ensuring compliance with licensing obligations while recognizing the contributions of upstream developers.

The legal framework varies by jurisdiction and license type, but generally requires preservation of specific notices as mandated by component licenses. For example, the MIT License[1] explicitly requires preservation of copyright notices and the complete permission notice, whereas the Apache 2.0 License[2] mandates more comprehensive preservation, including copyright, patent, trademark, and attribution notices, disclosure of modifications, and NOTICE file contents when applicable.

[1] https://mit-license.org
[2] https://www.apache.org/licenses/LICENSE-2.0.txt

2.7.1 Best Practices for Notice Management

Contemporary best practices emphasize automation and standardization:

Accessible Integration: Legal notices should be prominently accessible within software interfaces through dedicated sections (Wagner, 2023). This enables users to access licensing information easily. Failure to provide accessible attribution can have severe legal consequences.

Standardized License Identification: Using standardized identifiers such as SPDX License IDs enables precise identification and automated processing (Stewart, 2022). This ensures cross-tool compatibility and facilitates automated verification of compliance.

Consolidated Documentation: Aggregating all third-party licenses, attributions, and copyright statements creates comprehensive documents that provide users with complete transparency regarding licensing obligations.

Automated Generation: Implementing automated processes for generating and updating legal notices from SBOM data offers multiple benefits. It reduces manual errors, ensures completeness, and maintains synchronization between software composition and legal documentation.

SBOM-Driven Workflow Integration: Since the generation of legal notices already requires an SBOM (Wagner, 2023), notices can be directly incorporated into the SBOM itself, as each dependency and its associated metadata are explicitly listed. This approach consolidates both artifacts into a single file, reducing administrative overhead while maintaining consistency between technical documentation and legal obligations.

Building upon this foundation, the following chapter examines related research and existing tools that support configurable SBOM and legal notice generation systems.

3 Related Work

This chapter examines the current landscape of SBOM adoption and tooling to establish the foundation for configurable export systems. It traces the evolution from reactive security responses to regulatory mandates, analyzes the disconnect between compliance and operational utility, and assesses existing tool capabilities through empirical studies. The analysis reveals a critical gap in configurability support that directly informs the design requirements for the proposed solution.

3.1 The Evolution of SBOM Adoption

The transformation of SBOMs from niche compliance tools to strategic security infrastructure reflects a convergence of catalytic security incidents and accelerating regulatory mandates. High-profile vulnerabilities, such as SolarWinds and Log4Shell, exposed critical gaps in software supply chain visibility. Subsequent regulatory frameworks across multiple jurisdictions established concrete requirements for software transparency. This section examines how these parallel developments reshaped organizational approaches to software composition management, creating the foundation for modern SBOM adoption.

3.1.1 From Security Incidents to Strategic Imperatives

While the SolarWinds incident demonstrated the dangers of compromised build processes, the Log4Shell vulnerability discovered in December 2021 further intensified concerns about supply chain transparency. This critical remote code execution vulnerability in the Apache Log4j logging library affected an estimated 4% of packages on Maven Central, with scanning activity spiking within days of the disclosure as both researchers and attackers raced to identify vulnerable systems (Hiesgen et al., 2022). Unlike SolarWinds' compromised build process, Log4Shell exploited a widespread dependency present in countless applications, demonstrating how a single library vulnerability could cascade across entire software ecosystems.

Response times varied considerably. Organizations with established software

component inventories were able to evaluate their exposure more efficiently, while those lacking visibility faced delays in assessing potential impact (Bi et al., 2024). The discovery of Spring4Shell [1] a few months later, reinforced this pattern. Although its scope was narrower than Log4Shell, studies show that organizations leveraging SBOM-driven transparency could promptly determine whether their deployments met the specific exploitation conditions (Xia et al., 2023). These incidents underscored that reactive vulnerability scanning was insufficient; proactive mechanisms such as SBOM-based visibility were required to understand software composition before vulnerabilities emerged (Mirakhorli et al., 2024).

These incidents catalyzed a shift from viewing SBOMs as an optional compliance document to recognizing it as essential security infrastructure. Organizations began requiring SBOMs from their suppliers. Software supply chain attacks had become too frequent and too damaging to ignore.

3.1.2 Regulatory Acceleration and Standardization Efforts

Building on the regulatory milestones already outlined, the US Executive Order 14028 on Improving the Nation's Cybersecurity (issued in May 2021) quickly transitioned from high-level mandates to concrete technical requirements. The National Telecommunications and Information Administration (NTIA) published the first federal guidance in July 2021, specifying minimum SBOM elements across data fields, automation support, and operational practices (NTIA, 2021). These baseline requirements marked the transition from policy intent to enforceable practice, setting a foundation that later agencies, including Cybersecurity and Infrastructure Security Agency (CISA), would expand upon as SBOM adoption matured.

With the updated version of these requirements, released in August 2025, CISA introduced additional mandatory elements that reflect the evolution of SBOM practices. The new minimum elements require a Component Hash for integrity verification, License information for legal compliance, Tool Name for generation transparency, and Generation Context to indicate the lifecycle phase of creation. These additions address gaps identified through three years of implementation experience (CISA, 2025).

In parallel, the European Union's Cyber Resilience Act (CRA), adopted in 2024, established comprehensive transparency requirements. It mandates that all manufacturers provide machine-readable SBOMs for software products sold in the EU market, with non-compliance punishable by fines of up to 15 million Euros or 2.5% of their global annual turnover (BSI, 2024).

[1] Spring4Shell (CVE-2022-22965) See: "Spring Framework RCE, Early Announcement" 2022, https://spring.io/blog/2022/03/31/spring-framework-rce-early-announcement

These efforts demonstrate that standardization is not limited to single jurisdictions. At the international level, the ISO formalized SPDX as a global SBOM standard, ISO/IEC 5962:2021 (Stewart, 2022). National authorities have also issued complementary guidelines, such as the German German Federal Office for Information Security (BSI) 's technical guideline TR-03183 for critical infrastructure, while other countries, including Japan, are developing comparable frameworks aligned with international standards.

Momentum further extends into sector-specific domains. Regulatory bodies in critical industries have begun embedding SBOM requirements directly into their compliance frameworks: the US Food and Drug Administration (FDA) mandates them for medical device submissions (Haddad, 2024), while in the automotive sector, SBOMs are increasingly recognized as a way to satisfy supply-chain transparency obligations under ISO/SAE 21434 and are already being accepted by certification bodies in this context (Hannah, J. & Wilczynski, A., 2023). Financial regulators across multiple jurisdictions are likewise developing rules for systemically important institutions (Haddad, 2024).

These regulatory developments created both opportunities and challenges. Organizations gained precise requirements and standardized formats. However, the diversity of regulations across jurisdictions and sectors also created complexity. This complexity revealed fundamental limitations in how existing tools handle diverse stakeholder needs.

3.2 Configurable Export Systems: Addressing the Customization Gap

Current SBOM implementations reveal a critical disconnect between regulatory success and operational effectiveness: while compliance frameworks have successfully driven widespread adoption, the resulting tools often generate documents that fulfill legal requirements without providing meaningful business value. This gap emerges from the fundamental tension between standardized formats optimized for universal compatibility and the diverse, context-specific needs of different stakeholders within the software supply chain. This section examines the practical challenges that arise when organizations attempt to operationalize SBOMs beyond mere compliance, revealing significant limitations in current tooling capabilities. The analysis highlights how different stakeholder requirements necessitate configurable approaches that tailor SBOM content to specific use cases, ultimately arguing that flexibility in SBOM generation is essential for realizing the full potential of software supply chain transparency.

3.2.1 Implementation Challenges and Practitioner Perspectives

Despite regulatory mandates and standardization efforts, practitioners report significant challenges in SBOM implementation that reveal fundamental gaps in current tooling capabilities. Research across multiple studies has identified a consistent theme: organizations often do not struggle with generating SBOMs, but with making them useful for their specific contexts.

Survey data from Xia et al. (2023) reveal that 90% of organizations have initiated SBOM efforts, yet practitioners express deep concerns about the practical utility. As one respondent articulated: "I hope that the hype around SBOM will lead to something productive [...] and will not just be something which is a compliance requirement that's going to be met minimally."(Stalnaker et al., 2024). This sentiment reflects widespread concern that current SBOM implementations prioritize regulatory compliance over operational value, mainly because existing tools lack the configurability to adapt outputs to different use cases. This results in having a SBOM document that might even meet the compliance requirements, but not the right SBOM document you would actually want. Studies further emphasize that current SBOM tools are still immature, with limited interoperability and poor usability, restricting their practical utility: "SBOMs are at the dawn of adoption… current tools suffer from immaturity, poor interoperability, and limited configurability" (Dalia et al., 2024).

The utilization challenge becomes more apparent when examining organizational readiness. A practitioner's option stated: "You know, if you are a large organization and, say, you take a magic wand, and tomorrow all your software vendors start to provide accurate SBOM, what are you going to do with this?" (Stalnaker et al., 2024). The observation underscores that the barrier is no longer simply SBOM generation, but the operational capacity to integrate, analyze, and respond to this information at scale. Without mature processes, automation, and governance models, even perfect SBOMs risk becoming static documents rather than actionable intelligence.

Taken together, these practitioner insights highlight two systemic obstacles that existing SBOM tools leave unresolved. The first is a utilization gap. While SBOMs can provide unprecedented visibility, organizations often lack the analytic workflows, automation, and governance structures needed to convert this raw data into meaningful security or compliance actions. The second is a flexibility gap. Current approaches offer limited means of tailoring SBOM content to the heterogeneous requirements of different stakeholders, leaving a single static format inadequate for diverse use cases.

This thesis addresses the latter challenge through a configurable SBOM export mechanism. By enabling the creation of tailored SBOMs that match specific

consumer needs, the approach also improves overall usability, ensuring that the information delivered is not only available but also relevant and actionable.

3.2.2 Divergent Stakeholder Requirements

As outlined in the introduction, the needs of different stakeholders make it impossible for a single, static SBOM to be universally effective. Prior research and practitioner reports extend this observation by categorizing these divergent requirements into distinct use cases.

Security-Focused Internal Documentation: Studies emphasize that security teams require comprehensive SBOMs enriched with proprietary dependencies, detailed dependency graphs, and vulnerability correlation data (Haddad, 2024). These documents enable rapid incident response by helping teams identify affected components, estimate remediation efforts, and prioritize vulnerabilities (OWASP Foundation, 2024). In this context, SBOMs function less as compliance artifacts and more as operational instruments for proactive risk mitigation.

Regulatory Compliance Documentation: Compliance-driven SBOMs are typically more constrained, including only those fields mandated by specific frameworks such as federal procurement rules or sectoral regulations. These requirements usually cover component creators, names, versions, and dependency relationships, along with recursive dependency resolution (BSI, 2024; CISA, 2025). Their purpose is to foster accountability and ensure that organizations can demonstrate conformity with legal and contractual obligations (Haddad, 2024).

External Transparency Initiatives: For external stakeholders, such as customers or partners, SBOMs must strike a balance between transparency and the protection of sensitive intellectual property. This enables approaches where organizations selectively disclose information, allowing external consumers to assess their exposure without revealing proprietary architecture or detailed dependency structures.

Vulnerability Management Integration: Beyond documentation, some use cases emphasize integration with vulnerability management workflows. The NTIA survey identifies vulnerability management as a key consumption use case for SBOMs, noting that consumers may use them as input to tools that support vulnerability management and incident response among other operational functions (Alrich T. et al., 2021). These SBOMs incorporate enriched metadata such as vulnerability correlations and threat intelligence feeds, designed for automated processing across enterprise deployments. They support continuous monitoring, patch management, and forensic analysis, thereby extending SBOMs into active components of organizational security operations.

Taken together, these categories illustrate that the utility of SBOM is not a ques-

tion of availability alone, but of alignment with context-specific needs. Existing monolithic tools struggle to deliver this adaptability, reinforcing the demand for configurable export mechanisms.

3.3 Current Tool Limitations in Configuration Support

This analysis synthesizes findings from multiple empirical studies. These include comprehensive evaluations by Mirakhorli et al. (2024), Xia et al. (2023), and Bi et al. (2024). They evaluated functionality, accuracy, performance, and usability dimensions. Mirakhorli et al. conducted the most extensive analysis. They examined 84 SBOM-related tools and implemented a rigorous "Plugfest" benchmarking study. The study used five representative tools across standardized Java projects. Their methodology provides particularly valuable insights into current capabilities and limitations.

3.3.1 Tool Ecosystem Landscape and Capabilities

The SBOM tool ecosystem continues to grow rapidly, yet remains fragmented in scope and consistency of capability. Mirakhorli et al. (2024))report that the majority of tools focus on SBOM generation (54 tools, 64%), while consumption (31 tools, 37%), interoperability (15 tools, 18%), and quality assurance (14 tools, 17%) remain comparatively underrepresented. This disparity suggests a strong generation orientation but less maturity in downstream processing and quality validation.

The SBOM tool ecosystem demonstrates substantial progress in generation capabilities, with tools such as Syft (Anchore) and OSS Review Toolkit (ORT) offering extensive configuration options. However, their flexibility often comes with complexity: Syft requires careful parameterization to avoid oversized outputs, while ORT provides deep customization at the cost of a steep learning curve. Other widely used tools, including CycloneDX Generator, Trivy (Aqua Security), and the SPDX SBOM Generator, support multiple ecosystems but provide more limited configurability, typically through fixed command-line flags or format options. Despite this progress, recent analyses emphasize that SBOM tooling overall remains immature, with a pressing need for more reliable, easy-to-use, standards-compliant, and interoperable solutions (Dalia et al., 2024).

Overall, the challenge is less about the absence of configuration features and more about their usability trade-off: flexible tools often come with steep learning curves, and simpler tools may limit adaptability. This highlights the need for a more accessible, configurable solution that allows SBOM exports without

sacrificing flexibility.

3.3.2 Usability Challenges

While the preceding section outlined functional limitations of SBOM tools, practitioner studies by Xia et al. (2023) reveal that usability barriers are equally significant. Survey data indicate that 64.6% of practitioners agree existing SBOM tools can be challenging to use. Reported difficulties stem from complexity, unintuitive workflows, and insufficient generalization capabilities. These challenges limit adoption even when technical functionality is available, underscoring the need for approaches that make SBOM tools more accessible and practical in everyday organizational contexts. This emphasis on usability directly aligns with the objectives of SCA Tool and this thesis, which focus on lowering barriers through intuitive user interfaces.

3.4 Upstream and Validation Limitations

While not directly addressable by configurability solutions, upstream analysis and validation limitations in current SBOM tools create important design constraints for export systems. Empirical studies reveal that 73% of SBOM generators depend critically on package management infrastructure, with missing data increasing from 35% to 65% when build systems are unavailable (Mirakhorli et al., 2024). Tools exhibit fundamental limitations, including the inability to distinguish between actual component usage and declared dependencies, inconsistent support for identifier standards (CPE vs. PURL), and divergent dependency resolution approaches that yield incompatible outputs. Moreover, studies highlight that SBOM production is often belated and not dynamically maintained, leaving many documents obsolete shortly after creation (Dalia et al., 2024).

Validation challenges compound these upstream deficiencies: 69.2% of practitioners cannot verify SBOM integrity, and no standardized quality metrics exist for assessing it (Xia et al., 2023). Tools analyzing multiple languages often produce divergent field populations, depending on the ecosystem, which undermines stakeholder confidence in the resulting documents (Mirakhorli et al., 2024).

While configurable export systems cannot resolve these underlying analysis problems, they must operate within the constraints of incomplete and inconsistent input data. The configurable system developed in this thesis must therefore be designed to handle missing fields, inconsistent identifier formats, and varying levels of data completeness that characterize current SBOM generation tools. This necessitates architectural decisions that accommodate data variability, such as graceful handling of absent fields and flexible format mappings, rather than assuming perfect inputs. Although SCA Tool continues to improve analysis accur-

acy through specialized language-specific analyzers, these enhancements remain outside the scope of this thesis. The focus here is on developing an export system that remains functional and useful despite the inherent limitations of its input data, providing organizations with the ability to generate tailored SBOMs even when working with imperfect upstream analysis results.

3.5 Research Contribution: Bridging the Configuration Gap

The analysis reveals that configurability represents a distinct and critical limitation in the SBOM ecosystem. While generation tools proliferate and standards mature, organizations lack the means to adapt SBOM outputs to their specific contexts. This gap manifests as format rigidity, where tools generate fixed outputs that do not match stakeholder needs. It also creates complexity barriers that require deep technical expertise for configuration and setup. Organizations face manual overhead as they resort to error-prone post-processing. Compliance conflicts arise when a single output cannot satisfy multiple regulatory requirements. As a result, organizations often manually configure the SBOM content, develop their own tooling, or rely on third-party solutions due to the insufficient configurability of existing tools (Bi et al., 2024).

The NTIA explicitly acknowledges this need, stating that different use cases necessitate different information subsets (NTIA, 2021). However, current tools fail to provide accessible mechanisms for achieving this adaptability.

This thesis directly addresses the configuration gap through a systematic approach that provides **template-driven configurability** through pre-defined and customizable templates that encode stakeholder-specific requirements, eliminating the need for technical configuration expertise while maintaining compliance with standards. The system offers **multi-format support**, enabling the generation of multiple SBOM formats in a single application, eliminating the need to maintain different tools for different formats.

Furthermore, the solution provides **stakeholder-adaptive generation** through flexible configuration options that can be tailored to different consumer requirements, including security analysis, compliance reporting, and external partner communications, with appropriate field inclusion, format selection, and detail levels configured for each use case. Finally, it delivers **integrated workflow automation** by offering the option to include legal information directly in SBOM documents, making legal notices more easily available.

These capabilities directly address each configuration challenge identified in the practitioner studies and empirical analyses, providing a practical solution that

makes SBOMs actionable across diverse organizational contexts.

3.6 Summary: From Challenge to Solution

This chapter has traced the evolution of SBOM adoption from reactive security responses to proactive regulatory requirements, highlighting an ongoing challenge: limited support for configurable export mechanisms that adapt SBOM outputs to diverse stakeholder needs. While some tools already explore this direction, much of the current landscape still prioritizes generation accuracy and standard compliance over the flexibility required to make SBOMs operationally valuable across different contexts.

The empirical evidence demonstrates that diverse stakeholder requirements demand different SBOM configurations. Current tools offer either rigid outputs that serve a single use case or complex configurations requiring deep technical expertise, limiting their practical utility across diverse organizational contexts.

Building on this foundation of empirical evidence and stakeholder needs, the following chapter establishes the comprehensive requirements specification that guides the system's design and implementation, translating identified challenges into technical and functional objectives.

4 Requirements

The requirements presented in this chapter emerged through a two-phase methodology. Initially, a comprehensive analysis of SCA Tool's existing export capabilities was conducted, identifying fundamental limitations in configurability and support for format versions. This baseline assessment, extended by the literature review findings presented in Chapter 3, established the core requirement categories. Subsequently, during the iterative implementation process detailed in Chapters 5 and 6, these requirements underwent refinement and extension based on technical feasibility assessments, emerging use case patterns, and practical constraints encountered during development. This evolutionary approach ensured that requirements remained both ambitious in addressing identified gaps and pragmatic in their implementability within the existing SCA Tool architecture.

The requirements focus specifically on implementing configurable generators that leverage existing SCA Tool services for data provision, rather than reimplementing core functionalities. The SCA Tool, as the implementation platform for this research, provides foundational services for dependency resolution, vulnerability correlation, and license analysis, which the configurable export system transforms into tailor-made SBOM documents, including full legal notices.

Requirements are specified using the standard requirement definition language defined in RFC 2119 (Bradner, 1997). In this convention, SHALL (or MUST) indicates mandatory requirements, SHOULD indicates recommended requirements that are strongly advised but not strictly necessary, MAY indicates optional requirements that provide flexibility or additional functionality, and SHALL NOT (or MUST NOT) indicates explicitly prohibited behaviors or constraints.

4.1 Stakeholder Analysis

The configurable SBOM export system serves a diverse ecosystem of stakeholders with varying levels of technical expertise and distinct motivations for SBOM generation. This analysis examines two primary stakeholder groups whose needs

directly inform the system's functional requirements and use cases.

Developers and Security Engineers represent the technically proficient stakeholder group with a deep understanding of software composition analysis and SBOM specifications. Their core requirement is operational flexibility: the ability to generate SBOMs in multiple formats (SPDX, CycloneDX) with varying levels of granularity depending on specific project contexts. Developers may require lightweight SBOMs for internal dependency tracking, while security engineers need comprehensive dependency graphs for vulnerability correlation and threat assessment. This group leverages the integrated legal notice export capability to embed license texts and copyright attributions directly within SBOM documents, enabling immediate license compatibility assessment during development and reducing attribution omissions in distributed software. The configurable legal notice options provide precise control over the depth of information, ranging from basic license identifiers to complete license texts, tailored to specific compliance requirements or downstream consumer needs. For this stakeholder group, templates provide consistency and standardization across projects while preserving the flexibility to customize configurations for specialized use cases.

Compliance-Driven Non-Technical Users constitute the second stakeholder group, encompassing project managers, product owners, legal teams, and business stakeholders who require SBOMs primarily for regulatory compliance rather than technical analysis. These users typically lack in-depth knowledge of SBOM specifications, dependency analysis, or software composition intricacies. Nevertheless, they face increasing regulatory requirements, such as the European Union (EU) Cyber Resilience Act, executive orders mandating software supply chain transparency, or industry-specific compliance frameworks. Their primary need is simplicity: generating compliant SBOMs without requiring extensive technical knowledge of SBOM structure, format differences, or configuration nuances. The template management system proves particularly valuable for this stakeholder group by enabling the provision of pre-configured, compliance-focused templates that encode regulatory requirements into reusable configurations. For example, a CISA-compliant template can automatically include all mandatory fields without requiring users to understand the underlying technical specifications. These stakeholders can select the appropriate regulatory template and generate compliant SBOMs without navigating complex configuration options or risking non-compliance through incomplete manual configuration.

The template approach transforms SBOM generation from a technical barrier into an accessible business process for non-technical stakeholders. By abstracting complex configuration decisions into named, purpose-driven templates, the system enables regulatory compliance without requiring domain expertise. Users can confidently generate SBOMs knowing that organizational or regulatory requirements are automatically satisfied through template-encoded policies.

The convergence of SBOM generation and legal notice export into a single configurable system represents a significant advancement in software supply chain transparency. This architectural decision addresses the traditional inefficiency of maintaining separate workflows for tracking technical dependencies and documenting legal compliance. The template-driven approach further amplifies this benefit by making compliance accessible to non-technical stakeholders while preserving the flexibility that technical users require for specialized analysis workflows.

This stakeholder-driven approach ensures that the system addresses both immediate technical requirements and broader organizational governance objectives, enabling organizations to achieve comprehensive software supply chain transparency regardless of their teams' technical SBOM expertise.

4.2 Functional Requirements

The system requirements are organized using a structured numbering convention, where "FR-x" denotes functional requirements and "NFR-x" denotes non-functional requirements. This systematic approach ensures precise specification and enables systematic traceability during the evaluation phase presented in Chapter 7.

Functional requirements define the specific behaviors, functions, and capabilities that the configurable SBOM export system shall deliver to stakeholders. Following established software engineering practices and SBOM lifecycle considerations, these requirements are organized into four core functional domains that address the system's primary responsibilities: format compliance and generation capabilities, data processing and integration mechanisms, administrative template management, and export customization options. Each requirement group addresses distinct aspects of SBOM generation while ensuring comprehensive coverage of stakeholder needs identified in Section 4.1.

4.2.1 Multi-Format SBOM Generation

FR-1: The system SHALL generate SPDX v2.3 compliant SBOM documents in JavaScript Object Notation (JSON) serialization format and SPDX v3.0.1 compliant SBOM documents in JSON-LD[1] serialization format.

FR-2: The system SHALL generate CycloneDX v1.5 and v1.6 compliant SBOM documents in JSON output format.

[1] JavaScript Object Notation for Linked Data (JSON-LD) extends standard JSON by adding semantic context through the *@context* property, enabling linked data representation and RDF graph processing while maintaining JSON's human-readable format.

FR-3a: The system SHALL ensure core component metadata (package name, version, supplier, download location, and component hash) remains consistent across all generated SBOM formats for identical input data.

FR-3b: The system SHALL maintain licensing information consistency across formats, ensuring that if a component has specific licenses detected in one format, the same licenses appear in all formats where license export is enabled.

FR-3c: The system SHALL preserve vulnerability data consistency across formats, ensuring that detected vulnerabilities for a component are included in all generated formats when vulnerability export is enabled.

4.2.2 Data Integration and Processing

FR-4: The system SHALL integrate with existing SCA Tool services to consume dependency data, licensing information, vulnerability data, and component metadata.

FR-5: The system SHALL apply configuration options uniformly across all supported formats, ensuring that when a processing option is enabled (such as license normalization or dependency filtering), all formats supporting that option apply the same processing logic and produce semantically equivalent results.

4.2.3 Template Management and Configuration

FR-6: The system SHALL provide template creation, modification, and deletion operations through an administrative interface.

FR-7: The system SHALL implement role-based access permissions for template operations, including read, write, and execute permissions.

FR-8: The system SHALL validate template configurations against format-specific requirements and provide real-time feedback during configuration through two distinct validation mechanisms: enforcing unique combinations of template name and SBOM format through immediate input validation, and dynamically restricting configuration options to only those supported by the selected SBOM format and version, ensuring users can only select valid format-compatible options. If validation is not successful, then saving SHALL be disabled.

4.2.4 Export Customization Capabilities

FR-9: The system SHALL enable configuration of dependency inclusion scope for direct, transitive, and development dependencies.

FR-10: The system SHALL provide configuration options to control inclusion of component checksums, license information, copyright statements, and full license

texts, with Base64 encoding applied to license texts exclusively in CycloneDX v1.5 and v1.6 formats.

FR-11: The system SHALL enable the inclusion of vulnerability data with configurable granularity for external references (CVE/CWE/advisory URLs) and timestamps (published/modified). In contrast, vulnerability ID, source, severity rating, and Common Vulnerability Scoring System (CVSS) vector strings SHALL be included if vulnerabilities are contained.

FR-12: The system SHALL support addition of new SBOM formats and versions with minimal modifications to existing format implementations or template configurations.

FR-13: The system SHALL provide configurable legal notice integration within SBOM documents, enabling embedded copyright statements, license texts, and license acknowledgments directly within the SBOM structure using format-specific mechanisms.

4.3 Non-Functional Requirements

Non-functional requirements establish the quality attributes and operational constraints governing system performance. These requirements are organized into four domains: performance and scalability, quality and compliance, system integration and reliability, and usability and maintainability. Each domain provides measurable criteria for evaluating the success of implementation against stakeholder expectations.

4.3.1 Performance and Scalability

NFR-1: The system SHALL complete SBOM generation within 5 minutes for projects containing fewer than 2000 components.

NFR-2: The system SHALL complete template management operations (including API request, backend processing, PostgreSQL database operations, and response delivery) within 3 seconds for 95% of requests under normal load conditions.

4.3.2 Quality and Compliance

NFR-3: The system SHALL achieve 100% compliance with SPDX v2.3, SPDX v3.0.1, and CycloneDX v1.5/v1.6 specifications.

NFR-4: Generated documents SHALL pass validation against official format schemas with automated compliance checking.

NFR-5: The system SHALL maintain semantic consistency for essential package information between SPDX and CycloneDX outputs, ensuring that package identifiers, versions, suppliers, checksums, and dependency relationships remain equivalent across formats despite structural differences.

4.3.3 System Integration and Reliability

NFR-6: The system SHALL integrate seamlessly with existing SCA Tool data services and user management infrastructure.

NFR-7: The system SHALL implement comprehensive error handling that generates descriptive error messages and detailed logging for all failure scenarios, enabling efficient debugging and root cause analysis.

4.3.4 Usability and Maintainability

NFR-8: The system SHOULD provide template creation and modification interfaces that enable users to create and configure templates without consulting documentation successfully.

NFR-9: The system SHALL maintain backward compatibility for all existing templates when introducing new configuration options by applying sensible defaults for unspecified parameters, ensuring templates continue functioning across system updates.

NFR-10: The system SHALL implement a modular architecture that enables independent evolution of format-specific generation components to support future SBOM standard additions without modification of existing code.

4.4 Requirements Overview

Chapter 7 presents the evaluation of the identified requirements. In total, 13 functional and 10 non-functional requirements form the foundation for the configurable SBOM export generator. The system emphasizes three core capabilities: template-driven configuration, multi-format support with semantic consistency, and robust integration within the SCA Tool architecture.

5 Architecture

The configurable SBOM export system operates as a critical component within the SCA Tool ecosystem, transforming analyzed software composition data into standards-compliant documentation. This chapter presents the architectural design that enables flexible, scalable, and maintainable SBOM generation while addressing the configurability requirements identified in previous chapters.

5.1 Architectural Overview

SCA Tool functions as a comprehensive Software Composition Analysis platform, analyzing software projects to identify dependencies, vulnerabilities, and licensing information. The platform operates through three distinct services: the Analyzer Service (Kotlin-based) extracts dependency information using the OSS Review Toolkit (ORT)[1] and custom analyzers, the Scanner Service (Java-based) performs comprehensive license and copyright scanning, and the core monolithic application provides access to all necessary information, like curated license data, vulnerability information, and project metadata, through established APIs and populated databases.

This thesis focuses exclusively on functionality implemented within the monolithic backend service. While SCA Tool consists of multiple services, the configurable SBOM generation system developed in this work resides entirely within the established monolithic architecture. The Analyzer and Scanner services represent existing infrastructure that provides input data to the newly implemented SBOM generators, but remain outside the scope of this work.

The monolithic service employs Java Spring Boot, a framework that simplifies enterprise Java development by providing auto-configuration, embedded application servers, and production-ready features for building backend services that handle HTTP requests, implement business logic, and manage data persistence through databases. Spring Boot reduces boilerplate code and configuration overhead, enabling rapid development of robust web applications and RESTful APIs. The im-

[1] https://oss-review-toolkit.org/ort/

plementation follows OpenAPI specification-driven development, where API contracts are formally defined in a standardized format that enables automatic code generation, interactive documentation, and contract validation between frontend and backend systems. The frontend uses React with TypeScript, a component-based JavaScript library enhanced with static typing, with components integrated into the existing SCA Tool interface. This established technology stack enables the implemented SBOM export functionality to leverage existing authentication, data access, and user interface components without requiring infrastructure modifications.

Within this ecosystem, the implemented configurable SBOM generators occupy the export layer, serving as the bridge between internal analysis results and external consumption requirements. Spring Boot's dependency injection framework enables the modular component architecture essential for supporting multiple SBOM formats and versions. At the same time, the OpenAPI-first development approach ensures API consistency and facilitates frontend-backend contract validation.

The implemented generators leverage existing analysis capabilities while addressing the configuration and export challenges identified in Chapter 3. The system produces standards-compliant SBOM documents in multiple formats: SPDX versions 2.3 and 3.0.1, and CycloneDX versions 1.5 and 1.6 with JSON output.

The architecture directly addresses the core requirements established in Chapter 4: multi-format generation (FR-1, FR-2), configuration flexibility (FR-9 through FR-11), and template management (FR-6 through FR-8). The architectural decisions presented in this chapter demonstrate how these requirements shaped the system's structure and pattern selection within the constraints of the existing monolithic framework.

5.2 Architectural Drivers and Quality Attributes

The architecture prioritizes configurability as the primary quality attribute, directly addressing the thesis's central challenge of enabling flexible SBOM generation without code modifications. This configurability requirement permeates every architectural layer, from the template management system that empowers non-technical stakeholders to the selective data loading mechanisms that allow runtime customization of SBOM content. The architecture must accommodate diverse configuration scenarios. On the one hand, it must offer organizational templates that enforce compliance policies. On the other hand, it needs to support direct configurations and SBOM export for exploratory analysis.

Performance and scalability constitute equally critical architectural drivers, particularly given the computational complexity of SBOM generation. Processing a project involves traversing dependency trees, correlating vulnerability databases, resolving license conflicts, and serializing structured data. The architecture addresses these challenges through asynchronous processing to prevent API blocking, selective data fetching to minimize unnecessary computation, and stateless generation enabling horizontal scaling.

The architecture incorporates sophisticated design patterns for long-term adaptability. The SBOM landscape remains volatile. Standards continue to evolve with the addition of new mandatory fields and incompatible structures between versions. This reality shaped fundamental decisions: the Factory Pattern used for creating the different generators enables adding new formats without modifying existing code, separate implementations acknowledge paradigm shifts between major versions, and the configuration schema supports format-specific optional fields.

5.3 System Interaction Flow

The SBOM generation process follows a structured flow involving multiple architectural layers, as illustrated in Figure 5.1. The sequence demonstrates the clear separation of concerns between presentation, business logic, generation, and data access layers within the system architecture.

The process initiates when a user configures SBOM generation through the Web Frontend interface. Users can either select from pre-created organizational templates that enforce standardized compliance policies or create custom configurations specifying format, version, and generation options for ad-hoc requirements. The API Gateway receives the generation request and forwards it to the SBOM Management layer, which coordinates the overall generation process. The Generation Engine applies the specified configuration and initiates SBOM generation, querying the Data Layer for comprehensive project information required for SBOM construction.

While the diagram shows representative data queries for components and dependencies, the actual data access pattern is significantly more comprehensive. The Generation Engine retrieves extensive project data, including curated license data for compliance analysis, vulnerability information from security databases, copyright notices, and component provenance data. This data collection enables the system to produce complete, standards-compliant SBOMs with accurate security and legal information.

The architectural flow demonstrates the asynchronous nature of SBOM generation, where the system immediately returns an asynchronous response to main-

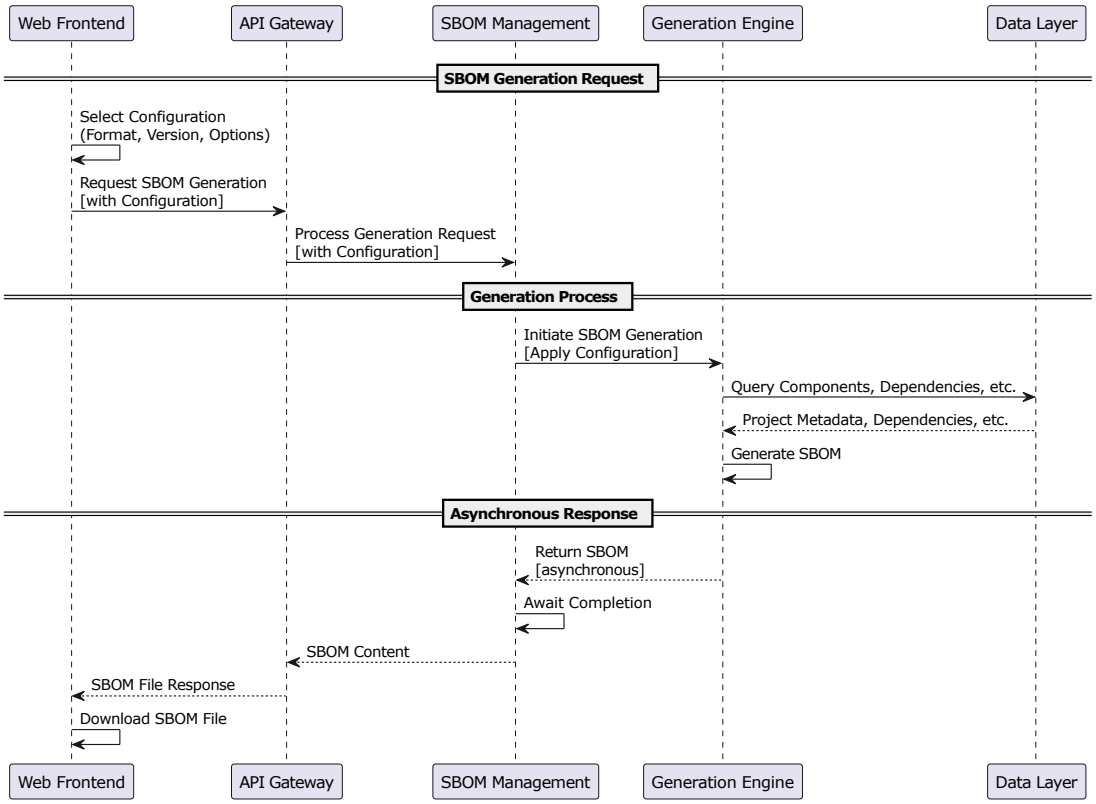

Figure 5.1: High-level SBOM Generation Sequence Diagram Showing Component Interactions

tain frontend responsiveness while processing continues in the background. Upon completion, the generated SBOM content is returned through the layered architecture, allowing users to download the final SBOM file. This separation enables the system to handle resource-intensive generation operations without blocking user interactions while maintaining clear boundaries between architectural concerns.

5.4 Core Architectural Decisions

The implemented system's architectural foundation rests on critical decisions that shape its entire structure: the selection of a layered architecture extending the existing monolithic SCA Tool structure and the deliberate separation of template management from generation logic. These decisions were made after careful analysis of integration constraints and performance requirements within the established monolithic framework.

5.4.1 Layered Architecture Selection

The implemented SBOM generation functionality employs a layered architecture that extends the existing monolithic SCA Tool structure rather than introducing a separate service architecture. This decision directly aligns with NFR-6, which involves integrating the system within the existing services by leveraging established patterns and introducing new capabilities.

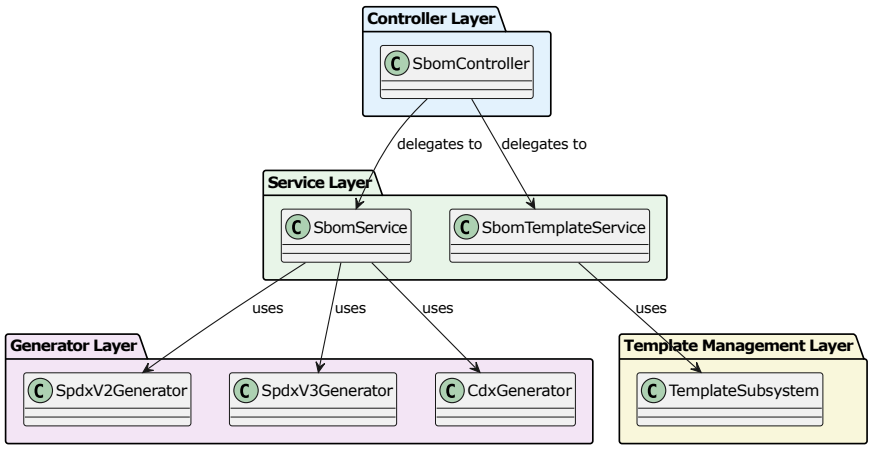

Figure 5.2: Backend Layered Architecture: Controller, Service, Generator, and Template Management Layers

Figure 5.2 illustrates the implemented layered structure showing the separation between controller logic, business services, generation engines, and template management concerns.

Controller Layer: The implemented controllers utilize Spring Boot's @Controller annotations and follow established SCA Tool patterns for request handling. Controllers implement interfaces generated from OpenAPI specifications to ensure specification compliance and prevent API drift between documentation and implementation. This layer provides API consistency with existing SCA Tool endpoints while introducing new SBOM-specific operations for both template management and generation workflows.

Service Layer: The business logic layer implements the critical separation between template management and generation orchestration through dedicated service classes. The *SbomService* coordinates generation workflows, while the *SbomTemplateService* manages template lifecycle operations independently. This separation enables two distinct usage patterns: template-based generation for standardized organizational exports and direct configuration for ad-hoc requirements. Spring Boot's @Service annotations and dependency injection framework provide clear architectural boundaries while enabling modular component management.

Generator Layer: Format-specific generation logic resides in dedicated generator classes that implement the actual SBOM document construction. This layer isolates the complexity of different SBOM standards (SPDX v2.3, SPDX v3.0.1, CycloneDX v1.5/v1.6) while providing a unified interface for the service layer. The Factory Pattern enables runtime generator selection based on user-specified format and version combinations, while different design patterns within generators handle version-specific implementation challenges. The following section 5.5 will dive further into these design choices.

Template Management Layer: Template persistence and project associations utilize Spring Boot's @Repository patterns for data access. This layer handles both template definitions at the organizational level and project-template associations through dedicated repository interfaces. The Template Management Layer maintains separation of concerns between template storage and template application, enabling configuration reuse without coupling to generation logic.

The layered approach within the monolith service was selected over microservices architecture, where functionality is decomposed into small, independently deployable services that communicate over network protocols, primarily due to integration requirements with existing SCA Tool infrastructure. Creating separate microservices would have required extensive API development for cross-service communication, introduced distributed transaction complexity for template management, and complicated deployment without providing benefits for the expected load patterns. The monolithic integration approach enables direct access to existing data repositories while maintaining clear architectural boundaries through Spring Boot's annotation-driven layer separation.

5.4.2 Template-Generation Separation

The architecture deliberately separates template management from SBOM generation, enabling maximum flexibility while maintaining simplicity. This separation allows two distinct usage patterns: template-based generation for standardized organizational exports, and direct configuration for one-time custom requirements without persistence.

This architectural decision contrasts with alternative approaches. A tightly coupled template-generation system would have simplified the API but eliminated support for ad-hoc configurations. A custom generation-only approach without templates would have maximized flexibility but sacrificed usability and governance capabilities. The chosen separation provides an optimal balance between convenience and flexibility.

The frontend orchestrates the relationship between templates and generation, extracting configurations from templates before invoking generation services. A detailed description of this mechanism will be in the next chapter in 6.6.5. This design ensures the backend generation system remains independent of template concerns, improving testability, maintainability, and development.

5.5 Pattern Architecture for different Formats

The system employs multiple design patterns to manage the complexity of supporting multiple SBOM formats with different structures. The pattern selection reflects a pragmatic balance between code reuse and maintainability, acknowledging that forcing uniformity across fundamentally different standards would create more problems than it solves.

5.5.1 Factory Pattern for Runtime Selection

The Factory Pattern, which encapsulates object creation logic behind a common interface, was chosen as the solution for runtime generator selection, addressing the requirement for dynamic format and version determination based on user input. This creational pattern delegates instantiation decisions to specialized factory methods that can evaluate runtime conditions and return appropriate concrete implementations.

Several alternative approaches were considered for this requirement. Dependency Injection containers, which are frameworks that automatically manage dependency creation and injection, could have handled runtime selection through factory beans, conditional binding, or named qualifiers based on user input parameters. Modern DI containers, such as Spring, support dynamic selection through features like specific annotations, factory methods, and runtime bean resolution.

The Service Locator pattern, where components query a central registry to locate their dependencies, could have provided runtime selection capabilities; however, it would have created implicit dependencies and complicated unit testing by requiring the setup of a mock registry.

Pure dependency injection without a factory abstraction would have required either eager instantiation of all generators (increasing startup time and memory usage) or complex conditional injection logic scattered throughout the codebase rather than centralized in a single location.

The Factory Pattern was ultimately chosen for the following reasons: simplicity of implementation, explicit control over creation timing, and architectural clarity. The factory approach provides a single, easily testable entry point for generator creation, eliminating the need for complex DI container configuration or annotations. The pattern makes the generator selection logic explicit and self-contained, providing a clear separation between the generator selection logic and the generators themselves.

This approach maintains type safety while providing the runtime flexibility essential for user-driven format selection, enabling clean integration with version-specific strategies for formats that benefit from shared implementation approaches.

5.5.2 Pattern Application in the Generators

The architecture's approach to design patterns reflects fundamental differences between the evolution patterns of SBOM standards, as illustrated in Figure 5.3.

For standards with evolutionary version changes, such as CycloneDX, which maintain substantial structural similarity, the Strategy Pattern provides significant code reuse benefits. The Strategy Pattern defines a family of interchangeable algorithms or behaviors through a common interface, allowing clients to select different implementations at runtime without modifying the code that uses them. This pattern proves particularly valuable when version differences are primarily additive rather than fundamentally different, enabling CycloneDX version-specific behaviors to be isolated within dedicated strategy implementations while preserving interface consistency across the generation framework.

For standards with revolutionary changes between major versions, the Template Method Pattern manages version-specific implementations within a shared framework. The Template Method Pattern defines the skeleton of an algorithm in a base class, with specific steps deferred to subclasses through abstract or virtual methods, enabling subclasses to override particular steps without altering the overall algorithm structure. SPDX demonstrates this approach, where the fundamental paradigm shift from document-centric to graph-based structures makes strategy abstraction counterproductive. The Template Method Pattern

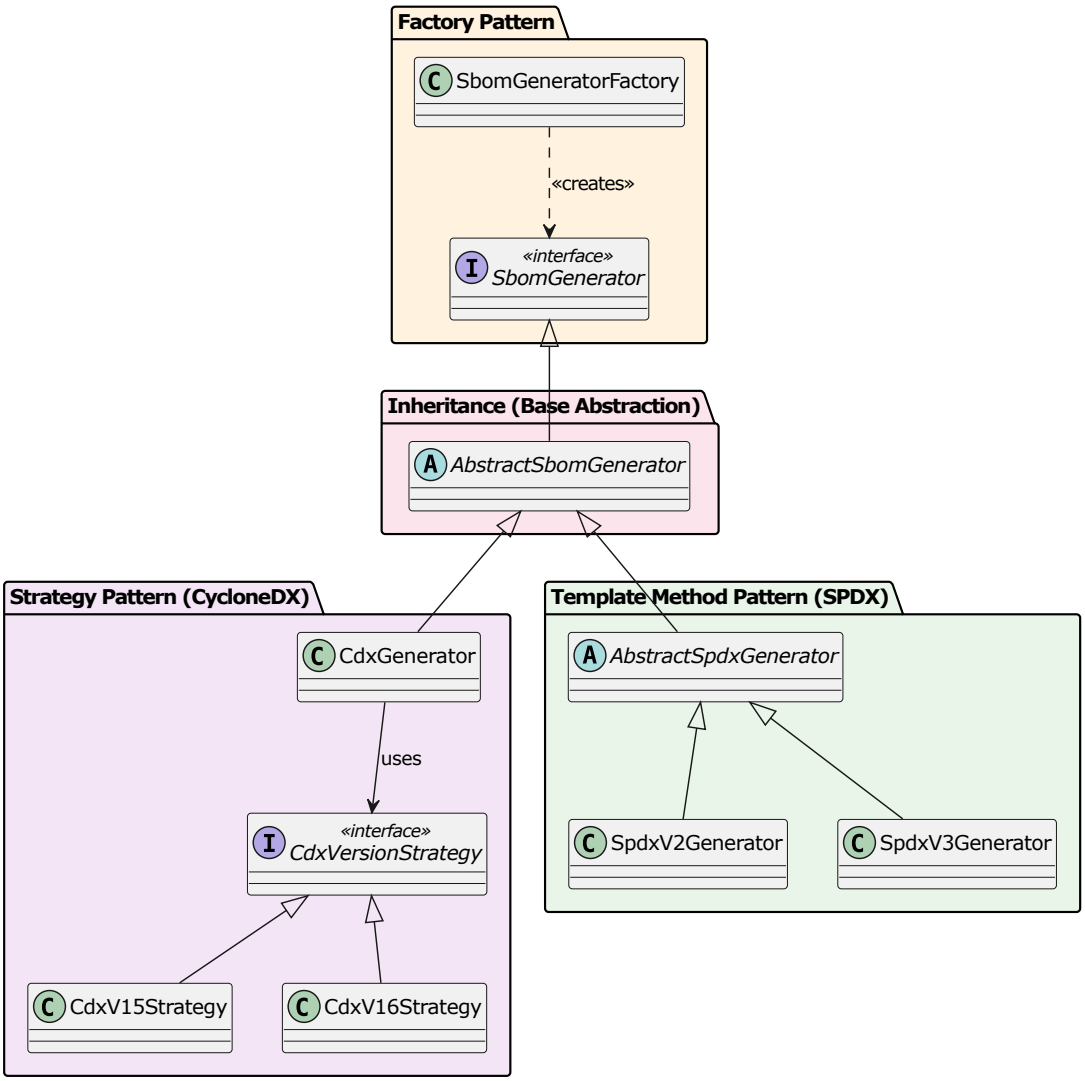

Figure 5.3: Design Pattern Implementation showing Factory, Inheritance, Strategy (CycloneDX), and Template Method (SPDX) Patterns

enables shared initialization logic while accommodating incompatible object models through deferred implementation methods that each version can implement according to its specific requirements.

This architectural divergence reflects the principle of applying patterns only where they provide genuine value rather than forcing uniformity. The Template Method Pattern acknowledges fundamental incompatibilities while extracting meaningful commonalities for shared infrastructure, while the Strategy Pattern leverages structural similarities to maximize code reuse across compatible versions.

5.6 Template Management Architecture

The architecture implements an External Configuration Store pattern, which separates configuration data from application code by storing configuration parameters in external, persistent storage systems that can be modified independently of the application lifecycle. Templates are stored in PostgreSQL at the organizational level, allowing for runtime configuration changes without requiring redeployment.

The template system follows a layered approach with clear separation of concerns. The backend employs Spring Boot data access patterns through Jakarta Persistence API (JPA) repositories for template persistence, while service layer abstraction provides business logic isolation. Template-to-project associations are tracked through a separate relationship table, allowing templates to be shared across projects while maintaining individual preferences.

On the frontend, localized React state management coordinates template operations with TypeScript interfaces, ensuring type-safe access to data. This architecture enables seamless synchronization between project templates and interface state without complex global state management overhead.

5.7 Performance Considerations

The system's architecture prioritizes configurability and modularity as primary design objectives, with performance benefits emerging as natural consequences of sound architectural decisions rather than through dedicated performance engineering. This approach demonstrates how well-designed systems achieve efficiency through clean separation of concerns, selective processing, and resource-conscious implementation patterns.

The asynchronous SBOM generation implementation prevents API thread blocking during long-running operations, enabling concurrent request handling without

performance degradation. This design choice primarily serves to enhance user experience by maintaining frontend responsiveness while also enabling better resource utilization in multi-user scenarios.

The configuration-driven, selective data fetching mechanism reduces database query overhead by retrieving only the necessary data based on user-specified options. When users turn off vulnerability information or license text inclusion, the system skips the corresponding expensive database operations entirely. This configurability feature provides both customization and inherent performance optimization.

The Factory Pattern implementation avoids unnecessary memory overhead by instantiating only the required generator implementations based on user format selection, rather than creating all possible generators eagerly. Template reuse reduces the overhead of repeated configuration processing when multiple projects use identical generation settings.

5.8 Security Architecture Integration

Rather than implementing custom authentication and authorization mechanisms, the architecture leverages SCA Tool's existing Authorization Service infrastructure. This decision reduces security implementation complexity while ensuring consistent access control across the platform.

Controllers verify user permissions before processing requests, ensuring users can only generate SBOMs or access templates within their organizational scope with appropriate permissions. The frontend performs preliminary permission checks to control UI element visibility, though the backend remains the authoritative security enforcement point. These architectural decisions align with FR-7, enforcing role-based access permissions on template operations.

5.9 Frontend Integration Architecture

The frontend architecture employs React with TypeScript for type-safe development, integrating SBOM functionality into the existing SCA Tool interface through modular component design. The React framework's component-based architecture suits SBOM operations well, as these workflows are self-contained and do not require global state coordination across the entire application. This architectural alignment enables focused user workflows that operate independently without complex application-wide dependencies.

The implementation leverages TypeScript interfaces generated from the OpenAPI specification to ensure type safety between frontend and backend interactions,

preventing both runtime errors and interface drift. Custom React hooks encapsulate SBOM-specific business logic, while this specification-driven approach provides an enhanced development experience through compile-time validation of API contracts. The architecture maintains controlled coupling between frontend and backend through API abstraction, where the frontend manages template-to-configuration extraction. In contrast, the backend remains independent of template concepts, allowing both layers to evolve without compromising the integrity of the integration.

State management follows a simplified approach that proves sufficient for the limited complexity of SBOM operations. Template operations, configuration management, and generation status tracking are facilitated through focused context providers and custom hooks, rather than requiring global state coordination. This localized state management reduces architectural complexity while maintaining adequate functionality for the discrete workflows involved in SBOM generation and template management.

Security enforcement in the frontend operates as an advisory layer that enhances user experience without replacing authoritative backend controls. Permission checks control UI element visibility to hide unavailable operations, but server-side enforcement remains the definitive security boundary. This approach improves usability by preventing users from attempting unauthorized actions while maintaining security integrity through proper backend authorization validation.

5.10 Architectural Validation

The architecture addresses the requirements established in Chapter 4 through deliberate structural decisions. The separation of template management and generation logic enables configurability requirements, while the Factory Pattern with format-specific strategies fulfills modularity and multi-format generation.

The layered structure provides clear separation of concerns while maintaining integration with existing SCA Tool infrastructure. The asynchronous processing architecture ensures system responsiveness under varying load conditions. The external configuration approach enables runtime flexibility without deployment overhead.

Chapter 6 details the concrete realization of these architectural decisions through specific classes, algorithms, and implementation patterns.

6 Design and Implementation

This chapter follows a top-down design decomposition approach, systematically examining how the architectural components established in Chapter 5 materialize into concrete implementations, algorithms, and technical solutions. Beginning with high-level system components, we progressively decompose each layer into specific classes, interfaces, and processing logic that deliver the configurability and performance requirements identified in the requirements analysis.

The implementation leverages established design patterns and modern software engineering practices, integrating seamlessly with the existing SCA Tool ecosystem while introducing new capabilities for template management, multi-format generation, and user-driven configuration.

6.1 System Component Design

The system component design realizes the layered architecture established in Chapter 5 through concrete implementation within the existing SCA Tool structure, extending the established Controller and Service layer patterns to accommodate SBOM generation and template management operations. The implementation maintains consistency with existing endpoints while introducing new capabilities through careful integration with the monolithic architecture.

Following the four-layer architecture defined earlier, each layer implements specific responsibilities through concrete classes and interfaces that integrate with the existing SCA Tool patterns. This section provides an overview of the implementation approach, with detailed mechanisms covered in subsequent sections.

Controller Layer Implementation: The *SbomController* follows the established OpenAPI-first design approach, implementing interfaces generated from formal API specifications. The controller maintains standard request validation, error handling, and security integration patterns, with permission checks operating at organization, project, and resource levels consistent with the existing authorization framework. The implementation ensures API consistency with existing SCA Tool endpoints while introducing new SBOM-specific operations for

both template management and generation workflows.

Service Layer Implementation: The business logic layer implements the architectural separation between template management and generation orchestration through dedicated service classes. The *SbomService* coordinates generation workflows, utilizing established transaction management approaches with read-only transactions for generation operations. The *SbomTemplateService* manages template lifecycle operations independently, enabling the dual usage patterns of template-based generation and direct configuration. The detailed template management implementation, including persistence mechanisms and project associations, is covered in Section 6.2.

Generator Layer Implementation: Format-specific generation logic resides in concrete generator classes (*SpdxV2Generator*, *SpdxV3Generator*, *CdxGenerator*) that implement the actual SBOM document construction. The *SbomGeneratorFactory* provides runtime generator selection based on user-specified format and version combinations, implementing the Factory Pattern established in the architectural design. This layer isolates the complexity of different SBOM standards while providing a unified interface for the service layer, with Strategy patterns handling CycloneDX version variations and Template Method patterns managing SPDX version differences. The comprehensive generation engine design and pattern implementations are detailed in Section 6.3.

Template Management Layer Implementation: Template persistence utilizes Spring Boot's @Repository patterns through *SbomTemplateRepository* and *SbomTemplateApplicationRepository* interfaces extending JpaRepository for standardized data access. JPA provides object-relational mapping capabilities that abstract database interactions through entity annotations and repository patterns, enabling the automatic generation of SQL and transaction management. The *SbomTemplateEntity* stores template definitions at the organizational level with JSON configuration objects, while the *SbomTemplateApplicationEntity* tracks project-template associations through a dedicated relationship table. This layer maintains separation of concerns between template storage and template application, implementing the External Configuration Store pattern through database persistence. The complete persistence architecture and backward compatibility mechanisms are examined in Section 6.2.

The implementation integrates seamlessly with existing SCA Tool design patterns while introducing the specialized patterns required for SBOM functionality. The data processing pipeline, detailed in Section 6.4, handles the systematic aggregation and transformation of software composition data from existing repositories. Format-specific implementations, covered in Section 6.5, demonstrate how each SBOM standard's unique requirements are addressed through the shared architectural foundation. Established error propagation strategies ensure consistent exception handling across all layers. In contrast, the new Factory and Strategy

patterns extend the system's capabilities to support configurable, multi-format export requirements within the existing architectural constraints.

6.2 Template Management System Implementation

The template management system implements a configuration storage mechanism that enables organizational governance while providing project-specific template persistence. Templates are stored at the organization level, with a single system template shared across all organizations. The *TemplateApplicationService* manages which template was last selected for each project, enabling persistent template associations without storing templates at the project level.

Figure 6.1 illustrates the service methods and controller interactions, including the underlying repositories.

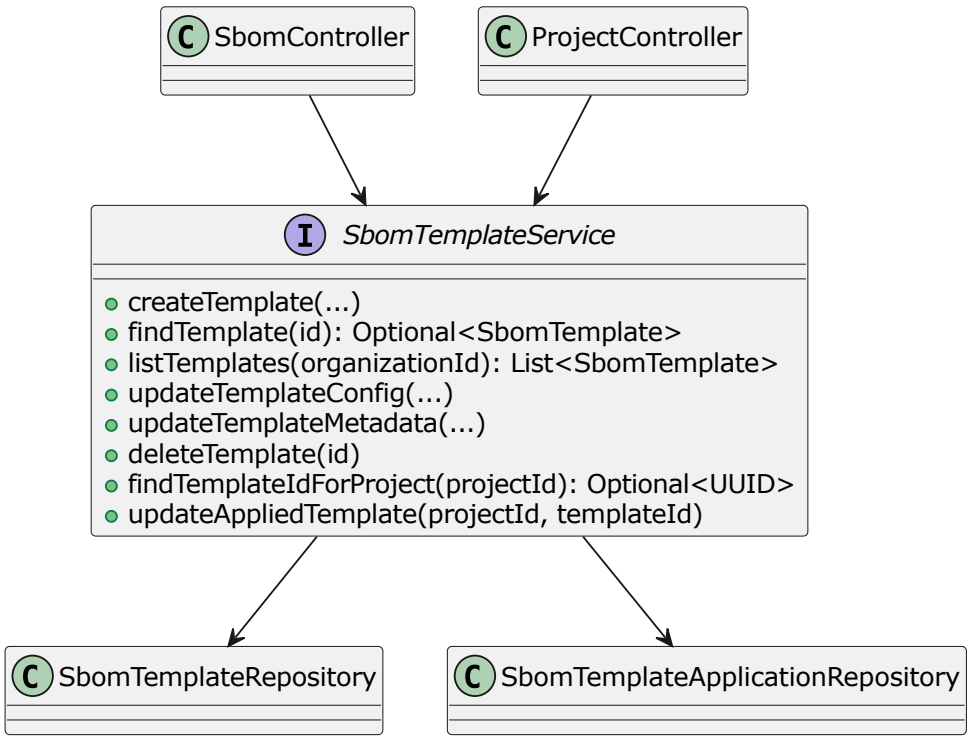

Figure 6.1: Template Service Methods and Dependencies

The *SbomTemplateService* serves as the central orchestrator for template operations, providing comprehensive template lifecycle management through Create, Read, Update, Delete (CRUD) methods. The service architecture demonstrates

clear separation between SBOM-specific operations handled by the *SbomController* and general project operations managed by the *ProjectController*. Notably, the *ProjectController* integration is minimal, utilizing only the *findTemplateIdForProject()* method to retrieve the currently applied template ID for the project context.

The service layer delegates persistence operations to two specialized repositories: *SbomTemplateRepository* for storing and retrieving templates, and *SbomTemplateApplicationRepository* for managing project-template associations. This separation ensures that template definitions and their applications to projects remain architecturally distinct while supporting efficient queries for both organizational template management and project-specific template resolution.

6.2.1 Template Persistence Architecture

The persistence architecture stores templates in a Postgres database at the organization level, with one system template available to all organizations. The system template uses a special Universally Unique Identifier (UUID) constant, null as, and is protected from modification through explicit immutability checks. The *SbomTemplateEntity* uses a nullable *organizationId* field, which indicates system templates if set to null, making them available for all organizations, while non-null values represent organization-specific templates. Organization templates support full CRUD operations for users with appropriate permissions, as shown in figure 6.1. The permission checks are handled by the existing *AuthorizationService*.

Spring Boot Data Access Implementation: The persistence layer leverages Spring Boot's data access capabilities through repository interfaces extending JPA Repositories. The *SbomTemplateRepository* provides all necessary operations for saving, editing, deleting, and custom query methods, such as *findByOrganizationIdOrOrganizationIdIsNull()*, to retrieve both organization-specific and system templates efficiently. Spring Boot's auto-configuration handles the underlying Hibernate session management and transaction boundaries. At the same time, custom repository methods utilize Spring Data JPA 's query derivation from method names for type-safe database operations.

Template configurations are stored as JSON documents using JPA 's *@JdbcTypeCode(SqlTypes.JSON)* annotation, providing schema flexibility while maintaining type safety through dedicated conversion mechanisms. The *SbomConfig* record encapsulates all configurable aspects of SBOM generation through various boolean options. Entity-DTO mapping employs *MapStruct* for compile-time safety and performance optimization, separating database concerns from business logic. This separation ensures that database-specific annotations and persistence concerns remain isolated from the domain model, while the *SbomTemplate* record provides a clean interface for service layer operations.

The backwards compatibility mechanism operates through a two-stage process that ensures seamless schema evolution. The direct compatibility stage occurs in the frontend, where a comprehensive configuration map contains all available options with their defaults. This map is generated from the frontend's single-source data model, which contains all options with attributes including defaults, labels, and categories. When new configuration options are introduced, they must be added to this frontend data model with appropriate default values. When a template is retrieved from the database, the frontend only updates the configuration map entries that are present in the template's stored configuration, leaving missing options at their predefined defaults. This approach ensures that older templates automatically inherit sensible defaults for newly introduced features. The frontend stage is particularly crucial because it prevents user interface inconsistencies and ensures that configuration options appear properly categorized and labeled, even when templates lack complete metadata.

The backend compatibility operates through the *SbomConfig* record's *@JsonCreator* constructor, which actively handles two critical scenarios: API request deserialization when the frontend sends configuration data, and database entity mapping when Hibernate deserializes stored JSON from the *sbom_config_templates.config* column. The constructor uses default Booleans for any missing configuration options to create the internally used record. This mechanism ensures backend consistency across both API interactions and database operations, providing essential robustness for external API clients and database entity hydration. This approach supports continuous deployment scenarios without requiring data migration or template updates, ensuring that templates created with earlier versions continue to function correctly as new features are introduced, existing ones are updated, or deleted through the automatic application of default values at multiple system levels.

6.2.2 TemplateApplicationService Implementation

The *TemplateApplicationService* manages persistent associations between projects and templates through a dedicated *SbomTemplateApplication* entity that stores project-to-template references. Figure 6.2 illustrates the database relationships supporting this template application system.

Repository Layer Architecture: The service utilizes Spring Boot's repository pattern through the *SbomTemplateApplicationRepository* interface. This repository provides specialized query methods for retrieving existing template associations, as well as *findAllByTemplateId*, which facilitates the cascading deletion of project-to-template associations when a template is deleted. Standard CRUD operations, like *save()* for creating new project-template associations, are inherited from JPA. Spring Boot's *@Transactional* annotation ensures the atomicity of these upsert operations, while the underlying JPA implementation handles

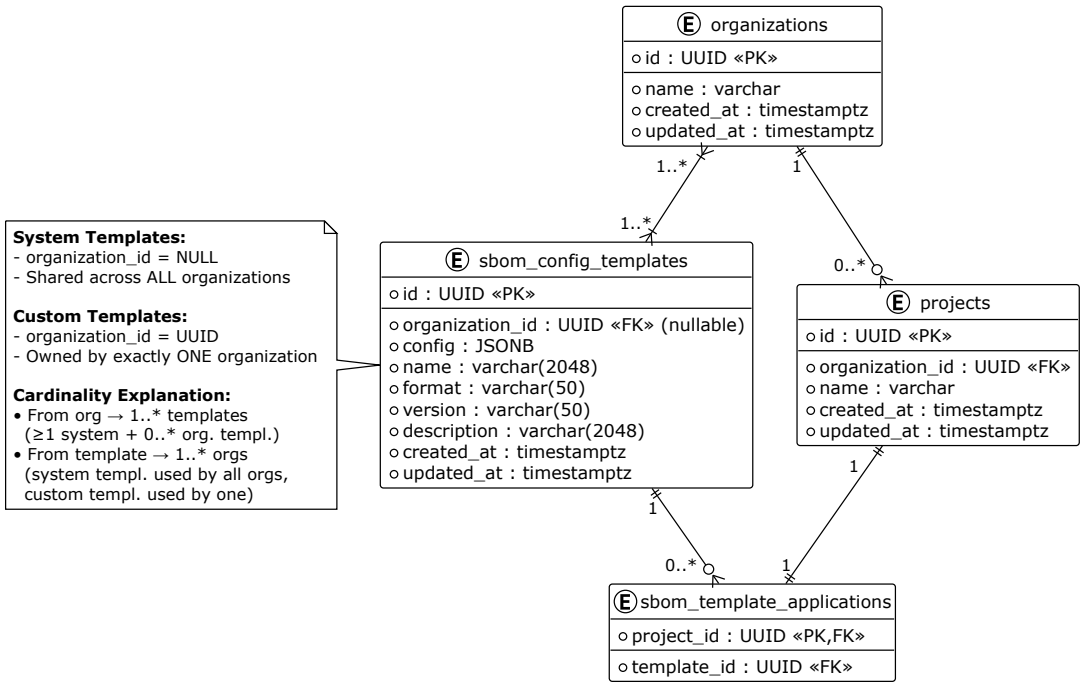

Figure 6.2: Template Application Database Schema Relationships

optimistic locking and constraint validation automatically.

The schema illustrates the organizational hierarchy, where organizations own both projects and templates. The *sbom_template_applications* table establishes the many-to-one relationship between projects and their selected templates. Each project can reference exactly one template, while the same template can be shared across multiple projects.

The service implements template selection persistence through upsert operations that create or update associations based on user selections. When a user selects a template for SBOM generation, the service automatically persists this choice in the *sbom_template_applications* table, ensuring that subsequent visits to the same project default to the previously selected template. This approach provides a consistent user experience while maintaining referential integrity between projects and their selected templates through the database constraints shown in the schema.

6.3 SBOM Generation Engine Design

The SBOM generation engine implements the core business logic for transforming analyzed software composition data into standards-compliant SBOM documents. The engine employs different design patterns based on the characteristics of each

SBOM standard: the Factory Pattern for runtime generator selection, the *AbstractSbomGenerator* foundation for shared functionality, the Strategy Pattern for handling CycloneDX versions, and the Template Method Pattern for SPDX generators. This approach reflects the varying degrees of compatibility between different standards and their evolution patterns.

Figure 5.3 from the previous chapter provides an overview of the complete design pattern structure across the generation engine.

6.3.1 Factory Pattern Implementation

The Factory Pattern provides runtime generator selection based on user-specified format and version combinations. The *SbomGeneratorFactory* offers two creation methods: one using the latest versions for each format, and another accepting specific version requirements.

The generator selection follows a structured flow that begins when the *SbomService* instantiates a generator using the factory. If an explicit version parameter is provided, the factory first validates the format version compatibility through validation methods that check if a version belongs to the SPDX or CycloneDX formats. Invalid or null versions trigger fallback mechanisms that default to the latest stable version for the requested format.

The selection process branches based on the format type. For SPDX requests, the factory calls *createSpdxGenerator()*, which uses switch statements to instantiate either *SpdxV2Generator* for version 2.3 or *SpdxV3Generator* for version 3.0.1. For CycloneDX requests, *createCdxGenerator()* creates CdxGenerator instances with the appropriate strategy injection - either *CdxV15Strategy* or *CdxV16Strategy* based on the version requirement.

When no version is specified, the factory automatically defaults to the latest versions: SPDX v3.0.1 and CycloneDX v1.6. This simplifies API usage while ensuring users receive the most current implementations unless they explicitly request older versions.

The factory employs separate methods for creating SPDX and CycloneDX artifacts, while maintaining a unified interface. Spring dependency injection ensures all required services and strategies are available during generator construction. This approach prevents invalid format-version combinations from reaching the generation stage while providing clear error messages for unsupported combinations.

6.3.2 AbstractSbomGenerator Foundation

The *AbstractSbomGenerator* serves as the foundational superclass for all SBOM generators, providing shared functionality and utilities that both SPDX and CycloneDX generators inherit. This class establishes a common infrastructure that eliminates code duplication while enabling format-specific implementations in concrete subclasses.

The abstract class manages service dependencies through constructor injection, providing consistent access to data services across all generator implementations. Data preparation occurs through the unified *prepareData()* method, detailed in Section 6.4, which creates structured *SbomData* records optimized for format-specific serialization.

The class provides format-independent utility methods for license expression processing, string encoding, supplier mapping, and dependency classification. This foundation enables significant code reuse while maintaining implementation flexibility in concrete SPDX and CycloneDX generator classes.

6.3.3 CycloneDX Strategy Pattern Implementation

The Strategy Pattern implementation for CycloneDX generators effectively handles evolutionary version changes. CycloneDX versions 1.5 and 1.6 maintain substantial structural compatibility, with version 1.6 requiring approximately 100 additional lines beyond version 1.5, primarily for new fields and adjusted data types.

The shared *CdxVersionStrategy* interface defines version-specific behaviors through key methods: *configureMetadata()* handles manufacturer information (not supported in v1.5, available in v1.6), *configureComponent()* manages author fields (v1.5 uses single author strings while v1.6 supports *OrganizationalContact* objects), and *isCompatibleExternalReference()* determines supported external reference types (v1.5 has limited support, v1.6 supports all types). Additionally, a method for setting the license acknowledgment highlights the differences between v1.6, which can mark licenses as concluded or declared, and v1.5, which lacks this capability.

As described in Subsection 6.3.1, strategy injection occurs at the factory level where Spring-managed strategy instances are provided to generators during construction.

6.3.4 SPDX Template Method Implementation

SPDX generators employ the Template Method Pattern rather than the Strategy Pattern due to fundamental paradigm differences between versions. The trans-

ition from SPDX 2.3's document-centric structure to version 3.0.1's graph-based model shares minimal commonality, making strategy abstraction counterproductive.

The *AbstractSpdxGenerator* implements the Template Method Pattern by defining *init()* as a template method that coordinates the initialization sequence while deferring *registerModelInfo()* to concrete subclasses. This approach ensures proper version-specific library initialization while sharing common setup logic, including creator information formatting and document namespace generation.

Document structure differences require distinct handling approaches for each version. Version 2.3 operates on a package-centered model where packages serve as containers for all information in a flat structure. Version 3.0.1 implements a graph-based approach with separate objects (License, Vulnerability, Assessment entities) connected through relationships. This fundamental difference between a package as a container versus interconnected entities makes the versions architecturally incompatible despite sharing the same format.

Version-specific implementations reflect these structural differences. *SpdxV2Generator* and *SpdxV3Generator* extend *AbstractSpdxGenerator* and provide their specific *registerModelInfo()* implementations.

The Template Method Pattern extracts genuinely shared utilities in the superclass while enabling independent version-specific implementation. This design offers code reuse for shared functions, while providing separation for version-specific document construction and serialization logic.

6.4 Data Processing Pipeline

The data processing pipeline establishes the internal mechanisms for organizing, retrieving, and preparing software composition data before format-specific generation occurs. It manages the systematic aggregation of component metadata, dependency relationships, vulnerability assessments, and licensing information from existing SCA Tool repositories into structured collections optimized for subsequent SBOM assembly operations.

6.4.1 Data Classification and Flow Analysis

The system categorizes data based on processing requirements. Directly used data includes component metadata such as Package URLs, CPE identifiers, and vulnerability identifiers that require only format-specific serialization, and preprocessed data that undergoes transformation to meet SBOM standards, including

URL encoding for safe document embedding and component scope analysis for development dependency classification.

Data Access Layer Implementation: The pipeline leverages existing data provider services that encapsulate repository access through dedicated *JpaRepository* extensions. These services, which are existing system components rather than newly developed ones in this work, include *ComponentService* for accessing software component data, *DependencyService* for relationship graphs, *FindingCurationService* for legal compliance data, and *VulnerabilityService* for security vulnerability information. Each service abstracts the underlying repository pattern, with repositories such as *ComponentRepository*, *DependencyRepository*, and *VulnerabilityRepository* extending *JpaRepository* interfaces for standardized data access operations.

The data transformation pipeline operates through five stages within the *prepareData()* method: core component data (always fetched), conditional dependency graphs, legal data enrichment, license expression mapping, and vulnerability data retrieval. This staged approach enables performance optimization through selective processing, where expensive operations, such as dependency graph retrieval, legal data enrichment, and vulnerability information collection, are skipped when not required.

6.4.2 Bulk Data Aggregation

The system implements efficient bulk data retrieval through batch database queries, fetching all component metadata, dependency relationships, and associated security and licensing information upfront. Thereafter, it aggregates the data in Map-based structures for optimized lookups during SBOM generation. Dependency scope filtering is handled later, uniformly across all format implementations through dedicated filtering logic detailed in Section 6.5.

Asynchronous Processing Architecture: The system utilizes Spring Boot's @Async annotation on *SbomService* methods, returning a *CompletableFuture* for non-blocking SBOM generation. Transaction boundaries are managed through *@Transactional(readOnly = true)* annotations to ensure data consistency during the read-heavy operations required for SBOM compilation.

6.4.3 License Data Processing Algorithms

The license processing employs two distinct mechanisms. The first is optional declared license normalization, and the second is hash-based mapping for discovered and concluded license texts. Declared license normalization is disabled by default to preserve maximum information transparency, allowing users to opt in when standardization is preferred over completeness.

The normalization process applies only to declared licenses from package metadata, converting variations through LicenseLynx[1] integration to canonical SPDX identifiers. Unmapped licenses are converted to "NOASSERTION" rather than being dropped to maintain transparency about potentially lost license information in compound expressions. This prevents users from unknowingly losing license data through the mapping process.

In contrast to declared license processing, discovered and concluded licenses from ScanCode scannings are not normalized since they already provide valid SPDX identifiers or specific ScanCode references like "LicenseRef-scancode-iso-8879" that cannot be meaningfully mapped by LicenseLynx. Normalizing these would result in information loss rather than standardization.

Hash-based license expression mapping provides a mechanism for resolving license information from the *FindingCurationService* output. The *FindingCurationService* returns SHA256 hash identifiers (*license_text_sha256*) for concluded license content, which requires correlation with license data stored in the *license_findings* table. The mapping process enables the retrieval of complete license expressions associated with each hash identifier. This approach maintains referential integrity between concluded license content and validated SPDX expressions by allowing the system to resolve hash identifiers to their corresponding license information.

6.5 Format-Specific Implementation

The format-specific implementations demonstrate how architectural patterns materialize into concrete SBOM document generation through specialized Java libraries. Each format leverages distinct library ecosystems: SPDX implementations utilize the SPDX Java Library, which includes model factories and storage mechanisms. At the same time, CycloneDX employs the CycloneDX Core Java library, featuring its object model API. The implementations maintain consistency through a shared *AbstractSbomGenerator* foundation while accommodating format-specific requirements through dedicated internal data structures and processing algorithms.

6.5.1 Common Implementation Patterns

All implementations share configuration-driven processing through the *SbomConfig* object's boolean flags, with identical component filtering logic across formats. The *SbomConfig* record encapsulates twenty distinct boolean configuration options that control the granularity of generation across four primary categories: dependency inclusion, vulnerability information, legal information, and technical

[1]LicenseLynx is an open-source license normalization library that maps license name variations to canonical SPDX identifiers. See: https://licenselynx.org

metadata. This comprehensive configuration system enables users to generate SBOMs ranging from minimal dependency lists to comprehensive security and compliance documents with full license texts, vulnerability assessments, and component checksums.

The *SbomConfig* record encapsulates all configuration options for SBOM generation, as shown in the complete definition:

```
@JsonIgnoreProperties(ignoreUnknown = true)
public record SbomConfig(
    boolean timestampedSpdxUris,
    boolean includeDependencies,
    boolean includeDirectDependencies,
    boolean includeTransitiveDependencies,
    boolean includeAllDependencyScopes,
    boolean includeComponentChecksums,
    boolean includeVulnerabilities,
    boolean includeVulnBasicInfo,
    boolean includeVulnExternalReferences,
    boolean includeVulnCwes,
    boolean includeVulnTimestamps,
    boolean includeVulnAnalysisNotes,
    boolean includeLegalInfo,
    boolean includeLicenseInfo,
    boolean includeLicenseTexts,
    boolean encodeLicTexts,
    boolean exactLicTexts,
    boolean includeCopyrights,
    boolean normalizeDeclaredLic,
    boolean includeExternalReferences
) {...}
```

Internal data structures employ HashMap-based caching strategies within the single generation flow: component filtering uses a *HashSet* for inclusion checks. Simultaneously, license processing utilizes a *HashMap* for SHA256 hash-to-expression resolution from the *FindingCurationService*.

License classification follows consistent patterns: declared licenses originate from package metadata, concluded licenses represent cleared scanning results, with compound expressions constructed using AND operators wrapped in parentheses for multi-license scenarios. License text inclusion operates through a two-tier strategy: non-listed SPDX licenses always include full license text regardless of configuration, while SPDX-listed licenses [2] include text only when *config.includeLicenseTexts()* is enabled.

[2] See SPDX License List: https://spdx.org/licenses/

Copyright text inclusion operates uniformly across all formats when *config.includeCopyrights()* is enabled, concatenating multiple copyright statements using semicolon separators.

Component integrity verification employs configurable checksum inclusion through *config.includeComponentChecksums()*, supporting multiple hash algorithms including MD5, SHA-1, SHA-256, SHA-512, and more. The implementation normalizes algorithm names through case-insensitive matching with delimiter removal, ensuring consistent hash representation across different package management systems.

All implementations employ unified dependency scope filtering through the *isProductionDependency()* method, which identifies components required for production runtime by excluding development, test, optional, and build-time dependencies. The method recognizes diverse package management patterns: Maven test/provided scopes, NPM devDependencies/peerDependencies/optionalDependencies, Gradle test variants (testImplementation, testApi, testRuntimeOnly), and Python development patterns (tests_require, dev-dependencies). The *includeAllDependencyScopes* configuration flag controls this filtering: when disabled (production-focused), only components identified as production dependencies are included in the SBOM; when enabled, all dependency scopes are included regardless of their production relevance. This comprehensive scope analysis ensures accurate SBOM generation across diverse package management ecosystems while providing users control over dependency inclusion granularity.

6.5.2 SPDX v3.0.1 Graph-Based Implementation

The SPDX v3.0.1 implementation follows the official specification (Linux Foundation, 2023). It uses the package *org.spdx.library.model.v3_0_1* together with a *JsonLDStore*, which wraps an *InMemSpdxStore* to provide graph-based RDF storage. The *SpdxV3Generator* utilizes *SpdxModelClassFactoryV3.getModelObject()* for element instantiation and *JsonLDStore.serialize()* for JSON-LD output generation.

Graph Construction and Element Creation: Document initialization creates the root *Sbom* element via *SpdxModelClassFactoryV3.getModelObject()* with *SpdxConstantsV3.SOFTWARE_SBOM* type specification. The *SpdxDocument* element references the Sbom through its *rootElement* property, while a shared *CreationInfo* object provides generation metadata. Element IDs follow the pattern *prefix + "ElementType/" + identifier*, with namespace prefixes constructed as *https://spdx.scatool.com/{verUnitId}/* or timestamped variants based on *config.timestampedSpdxUris()*.

Internal Data Structures and Caching: The implementation employs multiple HashMap-based caching mechanisms to eliminate redundant object cre-

ation and enable efficient relationship construction. *HashMap<String, Element> licenseTextElementMap* prevents duplicate license element creation by caching *ListedLicense* and *SimpleLicensingText* objects keyed by license expression and text content combinations, ensuring that identical license texts across multiple components reference the same SPDX element with an own relationship element rather than creating separate instances. *HashMap<UUID, Element> componentElements* provides constant lookup access to SPDX Package elements during relationship construction, enabling efficient dependency relationship creation (*DEPENDS_ON*) and vulnerability association (*AFFECTS*) without requiring linear searches through the document's element collection. *HashMap<String, Element> licensesElements* caches *LicenseExpression* elements to prevent recreating identical license expressions when multiple components reference the same license identifier, particularly valuable for compound expressions that require expensive string processing. *HashMap<String, Agent> supplierCache* reuses ecosystem-specific supplier agents (Maven, NPM, PyPI) across components from the same package management system, avoiding redundant *createOrganization()* calls and reducing memory footprint when processing projects with many dependencies from identical ecosystems. These caching strategies collectively reduce object instantiation overhead, memory consumption, and relationship construction time during large SBOM generation operations.

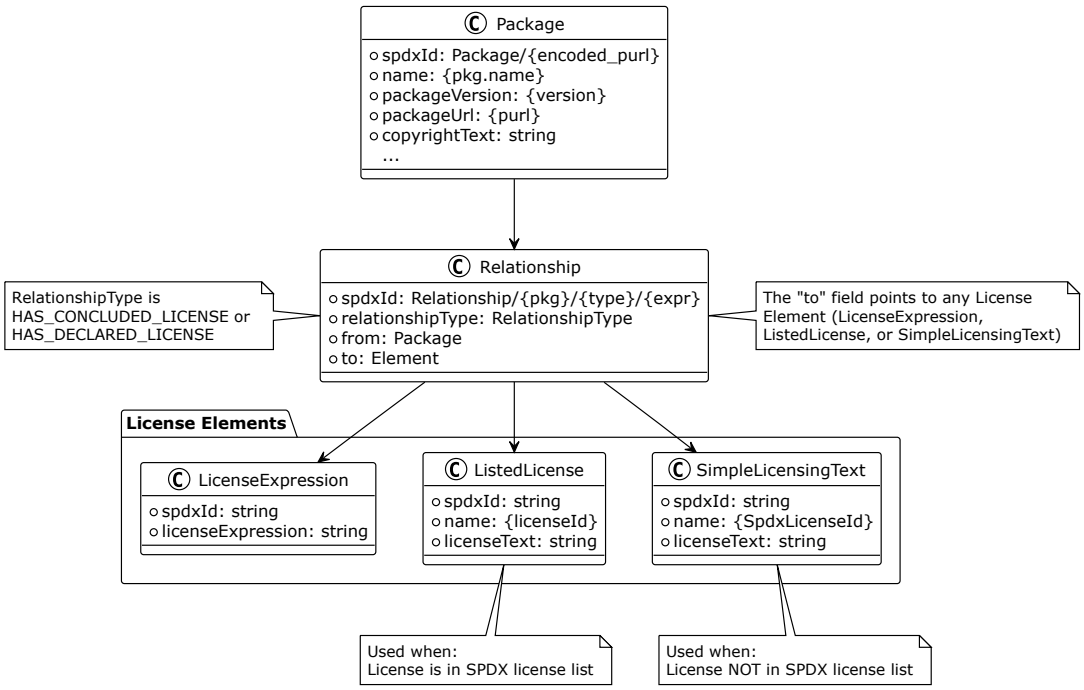

Figure 6.3: SPDX v3.0.1 License Model

License Model Architecture: License processing creates three distinct ele-

ment types through dedicated factory methods. *LicenseExpression* elements represent expressions without text content. *ListedLicense* elements handle SPDX-listed licenses with license text population, and *SimpleLicensingText* elements accommodate custom licenses. The *exactLicTexts* configuration controls text preservation: when enabled, unique text variations create separate elements with hashCode-derived URIs; when disabled, identical license identifiers collapse into single objects with escaped text.

Relationship Construction Mechanisms: The graph model creates explicit relationship elements where each relationship has a "from" element and a "to" array to establish connections between components. These relationships use type-specific enumerations. Relationships of type *HAS_DECLARED_LICENSE* connect packages to declared license expressions, while *HAS_CONCLUDED_LICENSE* links to concluded license elements, which are the cleared discovered license references and licenses with full text. An example showing this structure can be found in Appendix A. Dependency relationships employ the *DEPENDS_ON* type, and vulnerability relationships utilize *AFFECTS* for vulnerability-to-package connections and *HAS_ASSESSMENT_FOR* for assessment to package associations.

Vulnerability Integration with Assessment Relationships: Vulnerability elements are instantiated through *createVulnerability()* with mandatory fields populated unconditionally: vulnerability ID, summary, and description from Open Source Vulnerability (OSV) data, and GitHub Advisory external references created as *SECURITY_ADVISORY* type references. CVSS assessment creation demonstrates version-specific handling through *createCvssV2VulnAssessmentRelationship()*, *createCvssV3VulnAssessmentRelationship()*, and *createCvssV4VulnAssessmentRelationship()* builder methods, with severity classifications and vector strings always populated.

Optional vulnerability fields include analysis notes, temporal metadata using OSV timestamps, additional external references beyond GitHub Advisories, and Common Weakness Enumeration (CWE) weakness classifications extracted from OSV *databaseSpecific* fields. The implementation employs a priority-based CVSS version selection method using numerical weights (V4=4, V3=3, V2=2), with a reverse-ordered comparison logic, to ensure that contemporary vulnerability assessment methodologies take precedence over older versions.

6.5.3 SPDX v2.3 Implementation

SPDX v2.3 generation, driven by the official specification from Linux Foundation (2022), utilizes *org.spdx.library.model.v2* package classes through *SpdxModelFactoryCompatV2* with *InMemSpdxStore* for document assembly. The *SpdxV2Generator* creates flat, document-centric structures where packages exist at identical

hierarchical levels rather than being connected through separate relationship elements.

Document Assembly and Storage: Document creation uses the SPDX v2 factory (*createSpdxDocumentV2*) together with the default model store initializer to establish the storage context. A *MultiFormatStore* wrapper enables JSON serialization through *Format.JSON_PRETTY* specification, contrasting with v3.0.1's JSON-LD semantics.

Package Creation with Upfront License Parameters: Package instantiation requires three license parameters during *document.createPackage()* invocation: concluded license, copyright text, and declared license. This contrasts with v3.0.1's flexible relationship-based approach. The *licenseInfosFromFile* field populates with *List<AnyLicenseInfo>* entries for individual cleared discovered licenses, while *licenseConcluded* generates compound expressions through parenthetical wrapping logic for complex scenarios.

License Text Handling and Extracted License Creation: License texts are represented using *ExtractedLicenseInfo* objects, while their identifiers are referenced in the package's *licenseInfosFromFile* field. Custom license texts are invariably added as extracted license objects and must use the identifier pattern *documentnamespace#LicenseRef-{licensename}*. In contrast, standard SPDX-listed licenses are included directly with their official SPDX identifiers, if the option is enabled.

Vulnerability Encoding Limitations: The most significant v2.3 constraint involves encoding vulnerability data into external reference comments due to the absence of native vulnerability elements. The implementation creates *ExternalRef* objects with *ReferenceCategory.SECURITY* and advisory-type references pointing to GitHub Advisory URLs.

Always included vulnerability information comprises the vulnerability ID, advisory source URL, vulnerability summary when available, severity classification based on risk scores, and CVSS vector strings from all available versions. Configurable vulnerability fields include CWE identifiers, published and modified timestamps, and analysis notes from internal vulnerability assessments.

Due to the structural limitations of external references, all vulnerability metadata is concatenated into a single comment string using pipe delimiters, creating human-readable but programmatically challenging vulnerability documentation. This approach contrasts significantly with the rich structured representations available in SPDX v3.0.1 and CycloneDX formats, where vulnerability data exists as dedicated objects with typed fields and explicit relationships.

6.5.4 CycloneDX v1.5/v1.6 Implementation

CycloneDX implementation, based on the official specifications for v1.5 (OWASP Foundation, 2023) and v1.6 (OWASP Foundation & Ecma International, 2024) employs the *org.cyclonedx.model* package hierarchy through *BomJsonGenerator* for JSON serialization. The *CdxGenerator* orchestrates Bill of Materials (BOM) assembly through root *Bom* object instantiation with selective population based on configuration flags and version-specific strategy injection.

Strategy Pattern Implementation and Version Differentiation: The implementation utilizes *CdxVersionStrategy* interface injection for version-specific handling. Strategy methods include *configureMetadata()* for manufacturer information (v1.6 only), *configureComponent()* for author field management (v1.5 single strings vs. v1.6 *OrganizationalContact* objects), and *isCompatibleExternalReference()* for reference type filtering. License acknowledgment marking is handled through *setDeclaredAcknowledgement()* and *setConcludedAcknowledgement()*, which are exclusive in Version 1.6.

BOM Construction and Component Assembly: BOM initialization creates document properties including *bomFormat* set to "CycloneDX", *specVersion* (1.5 or 1.6), UUID-based *serialNumber*, and numeric *version* field. The *Metadata* class encapsulates document headers with the *Component.Type.APPLICATION* root component. The *Component.Type.LIBRARY* is used for the packages of the software application listed under the document's *components* section. Each component includes essential package metadata such as *name*, *version*, *description*, and *supplier* information. Components are uniquely identified through *bom-ref* fields containing package URLs (PURLs) and include integrity verification via *hashes* arrays with cryptographic checksums. Additional package metadata includes *copyright* statements, *scope* definitions (required/optional), and *externalReferences* arrays that provide links to distribution sources, websites, and other relevant resources.

License Choice Structure and Configuration: License modeling employs *LicenseChoice* containing two representation options: *List<License> licenses* or *Expression expression* for SPDX License Expressions[3]. The *evidence* field serves as a repository listing every detected license within the package, encompassing both declared licenses and all identified concluded license references. When *includeLicenseTexts* is enabled, complete license texts are incorporated into the evidence structure. The license processing supports multiple configuration flags: *normalizeDeclaredLic* triggers normalization through a custom normalization method detailed in Subsection 6.4.3, while *exactLicTexts* controls formatting preservation versus JSON escaping, and *encodeLicTexts* enables Base64 encoding. The primary *license* field stores the constructed concluded license expression

[3]https://spdx.github.io/spdx-spec/v3.0.1/annexes/spdx-license-expressions/

derived from all the cleared discovered licenses, which may result in compound expressions using AND operators when multiple licenses are identified.

Vulnerability Integration with Comprehensive Class Hierarchy: Vulnerability processing utilizes the CycloneDX *Vulnerability* class hierarchy supporting multiple rating systems and analysis states. Essential fields populate unconditionally: advisory identifier and source attribution via *Vulnerability.Source* objects linking to GitHub Advisories, CVSS ratings through *Vulnerability.Rating* with version-specific handling (v2, v3.1, v4.0), and affected component mappings through *Vulnerability.Affect* objects.

Version Range Processing and OSV Integration: Affected component specification employs Package URL Version Range Specification (vers) format[4] construction. The implementation processes OSV event data through stream operations: *introduced*, *fixed*, *limit*, and *lastAffected* events generate version constraint strings, such as *vers:npm/>= 1.0.0 < 2.0.0*. CWE extraction from *databaseSpecific* fields utilizes Jackson ObjectMapper for JSON parsing, filtering *CWE-* prefixed identifiers into integer arrays after substring extraction. These extracted CWE identifiers can then be included in the output through the *includeVulnCwes* flag configuration.

Other optional fields include recommendation text from FIX-type references and advisory collections, creation, publication, and modification dates from OSV data, and analysis objects with state mappings to the CycloneDX *enum State* (IN_TRIAGE, EXPLOITABLE, NOT_AFFECTED, FALSE_POSITIVE).

Serialization and Schema Validation: Final output generation leverages *BomJsonGenerator* with version-specific schema selection through *Version* enumeration values. The library's type-safe API ensures specification compliance through compile-time type checking, while runtime validation prevents invalid constructs.

6.6 Frontend Integration

The frontend implementation creates intuitive user interfaces for SBOM configuration and generation through React-based components and TypeScript state management. The design prioritizes user experience through immediate feedback, progressive disclosure, and intelligent automation while maintaining technical robustness through centralized state management and comprehensive validation systems.

[4]https://github.com/package-url/vers-spec/blob/master/VERSION-RANGE-SPEC.rst

6.6.1 React Context State Management for Component Coordination

The frontend employs React Context for centralized state management within the SBOM configuration interface, creating a scoped state container that coordinates between configuration options, template selection, and generation controls. The *SbomProvider* component encapsulates all SBOM-related state using React's *useState* hooks, providing a clean separation between SBOM functionality and broader application state management.

The context-based approach enables seamless communication between configuration toggles, format selectors, template dropdowns, and generation buttons, obviating the need for prop drilling and complex component hierarchies. State updates trigger immediate re-renders of dependent components, ensuring users see the effects of their changes reflected consistently across all interface elements. The *useSbomContext* hook provides type-safe access to state and actions throughout the component tree, maintaining consistent behavior while preventing access errors.

Configuration state management centers around the *configMap* object, which maintains current option values. The *toggleFlag* function provides a simple interface for making boolean configuration changes. The system automatically derives initial configurations from the centralized *SbomConfigData* metadata, ensuring default values remain synchronized with option definitions without manual maintenance.

Project template integration implements automatic template loading through a two-stage process that ensures users always see their previously configured settings when navigating to project SBOM interfaces. The system first monitors project changes through the *project.sbomTemplateId* field, which can contain three distinct states: undefined for no template selection, a special default template UUID for organizational defaults, or a specific template identifier for custom organizational templates. When project data loads or changes, the context automatically updates the template selection state to reflect the project's stored preference.

The second stage involves asynchronous template configuration loading, where the system retrieves the actual template data and applies its configuration, format, and version settings to the current interface state. This process handles missing templates gracefully by falling back to default configurations while providing appropriate error handling for network or data issues. The template loading mechanism preserves user workflow continuity by automatically restoring their previous configuration choices without requiring manual template reselection each time they access the SBOM interface.

Template integration demonstrates sophisticated state coordination where template selection automatically updates format, version, and configuration options while preserving user context about the source of current settings. The source tracking indicates whether current configuration values originated from an organizational template, project defaults, or direct user customization, enabling appropriate governance and compliance feedback. The *selectedTemplateId* and *isDefaultTemplate* flags enable the interface to distinguish between organizational templates, project defaults, and custom configurations, providing appropriate visual feedback about modification status and governance compliance.

Version management illustrates reactive state relationships where format changes automatically trigger default version selection through the *availableVersions* computation and *useEffect* hooks. This intelligent behavior prevents invalid format-version combinations while maintaining continuity of user workflow during format switching.

6.6.2 Single Source of Truth for Maintainable User Interface

The configuration system derives its entire user interface from a centralized metadata structure that defines labels, tooltips, validation rules, and behavioral relationships for each option. The *SbomConfigEntry* interface establishes the comprehensive schema that drives all configuration UI generation:

```
export interface SbomConfigEntry {
  key: keyof Schema["SbomConfig"];
  label: string;
  defaultValue: boolean;
  category: string;
  isParent: boolean;
  dependsOn?: keyof Schema["SbomConfig"];
  tooltip?: string;
  disabledFor?: Array<
    | {
        format: Schema["SbomFormat"];
        version?: Schema["SbomVersion"];
      }
    | Schema["SbomFormat"]
  >;
  disclaimer?: string;
  disclaimerWhenEnabled?: boolean;
}
```

This metadata-driven approach ensures a consistent presentation across all inter-

face components, while enabling the rapid addition of new configuration options without requiring interface redesign. Users benefit from uniform interaction patterns where similar configuration types behave identically regardless of their position within the interface hierarchy. The *key* field establishes type-safe linkage to the actual configuration schema, while the *label* and *category* properties facilitate automatic UI organization and grouping.

The dependency system, as defined by the *isParent* and *dependsOn* fields, creates logical hierarchies that automatically manage option availability and state cascading through parent-child relationships. When users interact with parent options, the system automatically updates dependent children without requiring manual intervention, thereby reducing cognitive load while maintaining configuration consistency.

Format-specific option visibility, handled through the *disabledFor* array, simplifies the interface by automatically disabling unsupported configurations based on the selected SBOM format and version. Users working with SPDX v2.3 see CycloneDX-specific options rendered in light grey with explanatory tooltips indicating format incompatibility, providing clear visual feedback about option availability without completely hiding functionality. This filtering creates focused interfaces tailored to current needs while maintaining awareness of functionality across all supported formats. This allows users to understand the full scope of available options even when they cannot access them with their current format selection.

The integrated *tooltip* properties provide contextual guidance, while *disclaimer* and *disclaimerWhenEnabled* fields enable proactive warnings about potentially problematic configuration combinations. Users receive direct guidance on configurations, enabling them to make informed decisions when selecting configuration options.

Default value management through the *defaultValue* field ensures an initial configuration, which is required for new projects, where no template has been selected.

6.6.3 Intelligent Configuration Cascading and Three-State Logic

Component interaction design in the configuration menu centers around intelligent cascading logic that manages configuration relationships through a sophisticated three-state parent switch system. Parent options, identified by the *isParent* flag in the *SbomConfigEntry* interface, utilize switches with three distinct states: all options enabled, all options disabled, and a custom middle state indicating mixed configuration.

Three-State Parent Switch Logic: Parent switches implement a cyclic three-

state system that transitions through the two states, all or none, on click. The "all" and "none" states represent complete category enablement or disablement, while the "custom" state emerges when users selectively modify individual options within a category. This cycling pattern enables users to quickly access standard configurations while preserving granular customization capabilities through the middle state.

Hierarchical Dependency Management: The cascading system demonstrates sophisticated awareness of option hierarchies by implementing selective propagation rules based on dependency depth. When the parent switches activate "all" mode, the system enables only first-level dependencies, while deliberately excluding second-level dependencies that primarily represent formatting or encoding enhancements. This selective approach prevents overwhelming users with specialized options they may not require while ensuring essential functionality remains instantly accessible.

Bidirectional dependency relationships handle individual option interactions, where enabling child options automatically activates required parent options, and disabling parent options cascades to dependent child options. For example, enabling Base64 encoding (available only in CycloneDX) automatically enables license text inclusion, as encoding requires the presence of text, while turning off license text inclusion disables the encoding options. This bidirectional management prevents invalid configuration states while minimizing user cognitive load about technical prerequisites.

User Experience Optimization: The visual distinction between essential options (activated by parent switches) and enhancement options (requiring explicit activation) guides users toward appropriate configuration choices without overwhelming them with technical dependency relationships. The system strikes a balance between automation convenience and customization flexibility, allowing users to quickly achieve standard configurations while maintaining control over specialized features that serve specific use cases.

Furthermore, this dependency management eliminates frustrating blocked-action scenarios where users encounter disabled options without understanding prerequisites. Instead of requiring users to discover dependency relationships through trial and error, the system proactively enables required options and clearly communicates relationships through visual cues and state changes.

6.6.4 Template Management Interface

The template management interface features a sophisticated, permission-based design that adapts functionality based on user organizational roles, while maintaining intuitive workflows for all permission levels. The system employs the *useResourcePermissions* hook to dynamically determine user capabilities, with

hasPermission("manage_sbom_templates") controlling creation, editing, and deletion operations, while *hasPermission("view_sbom_templates")* enables template browsing and selection.

Users without management permissions are presented with a streamlined interface that displays only template selection capabilities and informational dialogs, eliminating visual clutter from unavailable actions while preserving access to essential functionality. The permission system operates transparently, hiding management buttons and dialogs rather than showing them in disabled states, creating clean interfaces tailored to user capabilities without suggesting restricted functionality. Screenshots showing the UI and the update dialog can be found in Appendix B.

Template Operations Architecture: The *useSbomTemplateOperations* hook encapsulates all template manipulation logic, providing centralized validation, error handling, and API communication. This hook implements comprehensive name validation that checks for empty names and duplicate names within the same template format across an organization, providing immediate feedback near the input fields.

The operations architecture separates template configuration updates from metadata changes through distinct API endpoints. This design allows template settings and descriptive information to be modified independently, supporting flexible workflows and improving API clarity and efficiency.

State Management and Dirty Tracking: The template workspace implements sophisticated dirty state detection that monitors configuration, format, and version changes relative to the selected template. The system calculates this dirty state by comparing the current interface settings against stored template values, accounting for legacy templates that lack version specifications, and provides visual feedback when users deviate from saved configurations.

Suppose the configuration is in a dirty state. In that case, the reset functionality enables users to quickly revert to saved template configurations, while the save option becomes available, which offers two distinct options for the user. Option one is to save it as a new template, whereas option two updates the selected template to this configuration.

Dialog-Based Interaction Design: Template management employs modal dialogs for complex operations, providing focused interfaces that guide users through creation, editing, and deletion workflows. The creation dialogs emphasize the uniqueness of the name and the completeness of the configuration, while the editing dialogs focus on showing the updates compared to the current version.

The dialog system implements proper lifecycle management where opening dialogs prepopulate fields with relevant existing data, while closing dialogs reset form states to prevent confusion during subsequent operations. Delete dialogs

provide explicit confirmation interfaces that display template names and affected configurations, helping users make informed decisions about template removal without accidental deletions.

Template Selection and Application: The template selection UI provides intuitive dropdown functionality, handling selection changes that consider the dirty state. When the template is in a dirty state, the dropdown menu offers an additional option to restore the original template state by displaying a 'Reset Selected Template' option.

The template application demonstrates seamless integration between frontend interface updates and backend persistence through automatic project template assignment. When users select templates, the system updates the local interface configuration simultaneously. It saves the template preference to the project, ensuring consistency between user experience and persistent storage without requiring explicit save actions.

The rendering system provides a contextual display of template names, clearly indicating configuration status through visual cues. It shows 'Configuring manually' for unsaved changes and template names with styling variations for modified configurations. This contextual feedback clarifies the origin of configurations while informing the user about saving, resetting, or proceeding with their current settings.

6.6.5 Flexible Configuration and Generation Workflows

The frontend implementation realizes the template-generation separation described in Section 5.4.2 through two distinct workflows that serve different user needs while sharing the same generation endpoint.

Template-Based Workflow: Users select organizational templates that automatically populate configuration options, format, and version settings. When a template is selected, the system loads the template's stored configuration into the current interface state. It saves the template association to the project via the *PUT /projects/{projectId}/sbom-template* endpoint. This workflow supports standardized organizational practices where teams use predefined configurations for consistent SBOM generation across projects.

Direct Configuration Workflow: Users manually adjust configuration options without template constraints, creating custom configurations that exist only in the current session. These modifications do not affect stored templates or require the creation of new templates, enabling one-time customizations for specific analysis requirements. Users can load template configurations as starting points and then modify them freely without persistence concerns.

Unified Generation Process: Both workflows converge at the generation

stage, where the system extracts the current configuration state and passes it to the SBOM generation API. The *GET /ver-units/{verUnitId}/sbom* endpoint receives configuration parameters as JSON regardless of whether they originated from templates or direct editing. This approach ensures the backend generation system remains independent of template storage while providing consistent generation behavior across all workflow types.

This implementation successfully serves both organizational governance and individual customization needs through a single, straightforward interface that maintains backend simplicity.

6.6.6 Scan Status Communication and Process Transparency

The scan status component appears during legal data scanning, displaying circular progress indicators with completion percentages and visual feedback on the status. It supports active scanning and failure state for scan interruptions or failures.

The component informs the user about a strategic compromise by enabling SBOM generation during active scanning, allowing users to download documents immediately rather than waiting for complete legal data analysis to be completed. This early-generation approach trades data completeness for availability: SBOMs generated during scanning include only discovered license information for already-processed components. In contrast, unscanned components contain solely the declared license data from package metadata.

This transparency enables users to decide whether to proceed with immediate generation or wait for more comprehensive data, based on their specific needs.

6.7 Implementation Summary

This chapter demonstrated the systematic translation of architectural principles into concrete implementation through established software engineering patterns and modern development practices. The implementation successfully balances the competing requirements of organizational governance and individual flexibility while maintaining integration with the existing SCA Tool ecosystem.

The template management system provides persistent configuration storage with backward compatibility mechanisms that ensure seamless system evolution without requiring data migration. The SBOM generation engine employs appropriate design patterns—Factory for runtime selection, Strategy for CycloneDX version handling, and Template Method for SPDX structural differences—reflecting the

varying compatibility characteristics of different standards and their evolution patterns.

The data processing pipeline optimizes performance through bulk aggregation and selective processing, while format-specific implementations demonstrate sophisticated integration with specialized Java libraries for SPDX and CycloneDX document construction. Internal caching mechanisms and hash-based license mapping provide efficient resource utilization during large-scale SBOM generation operations.

The frontend implementation realizes the template-generation separation through React Context state management and metadata-driven UI generation, creating intuitive user workflows that accommodate both structured organizational processes and exploratory individual analysis. The permission-aware template management interface adapts functionality based on user roles while maintaining consistent user experience patterns.

Together, these implementation components establish a robust foundation for configurable SBOM generation that integrates seamlessly with existing SCA Tool workflows while introducing new capabilities for template-based standardization and multi-format document export. The system's effectiveness and performance characteristics under real-world usage conditions are evaluated in the following chapter.

7 Evaluation

This chapter presents the evaluation of the configurable SBOM export system against the requirements defined in Chapter 4. Each requirement is examined and classified as fulfilled, partially fulfilled, or not fulfilled.

The evaluation combines manual feature testing, automated end-to-end tests for SPDX v3.0.1 and CycloneDX v1.6 formats, standards compliance verification using official format documentation, library integration, and official validation tools, as well as performance measurement through direct timing of system operations. Some usability requirements remain unvalidated due to the absence of formal user testing protocols.

7.1 Functional Requirements Evaluation

FR-1: SPDX v3.0.1 compliance has been fully achieved through implementation based on official documentation, utilizing SPDX Java libraries for object creation, document creation, and serialization. The implementation successfully validates through the SPDX validation tool[1]. Automated Playwright end-to-end tests integrated into the GitHub CI workflow validate the complete download workflow and verify document structure, including required metadata fields, root package identification, license inclusion, and dependency graph representation. All validation and testing were conducted with configuration options enabled that ensure maximum content inclusion and thorough verification coverage.

However, the SPDX v2.3 implementation presents more complex validation challenges. While the implementation follows official documentation and utilizes SPDX Java libraries for object creation and serialization, validation through the SPDX validation tool succeeds for some documents but encounters failures for others due to an *org.spdx.core.InvalidSPDXAnalysisException*. The error indicates that extracted license information object URIs do not follow the required SPDX V2.X pattern. Extensive analysis was conducted using multiple SBOM documents with different degrees of data inclusion and for various projects.

[1] https://tools.spdx.org/app/validate/

The demo project successfully passed validation, demonstrating that the implementation can generate valid SPDX v2.3 documents. However, validation failures occur inconsistently for other projects, producing the same URI pattern exception even when the problematic elements are not included in the generated SBOM. This inconsistent behavior, where identical implementation logic succeeds for some projects but fails for others, suggests that an incorrect exception is being raised by the validator or even a potential bug in the validation tool, rather than systematic non-compliance. This validation inconsistency will be analyzed as part of future work. *Partially Fulfilled.*

FR-2: CycloneDX v1.6 and v1.5 compliance has been achieved using official CycloneDX documentation and library implementations. The implementation successfully validates through the CycloneDX Web Tool[2]. All validation and testing were conducted with configuration options enabled that ensure maximum content inclusion and thorough verification coverage. Comprehensive Playwright end-to-end tests for the latest version execute within the GitHub CI workflow to validate the complete SBOM download process and verify core document components, including required metadata fields, root package structure, license information, and dependency relationships. *Fulfilled.*

FR-3a: Consistent core component metadata is guaranteed across all supported formats and versions through standardized data processing mechanisms and inclusion in each format and version. *Fulfilled.*

FR-3b: Each package maintains identical license information across all formats and versions through common implementation patterns (see Section 6.5.1). The used approach ensures consistent input data processing and uniform license representation. *Fulfilled.*

FR-3c: Vulnerability data consistency has been achieved across all supported formats. Core vulnerability information, including vulnerability ID, source attribution, and CVSS vector data, is included in all formats when present in the source data. However, format-specific limitations exist, particularly in SPDX v2.3, which provides limited support for vulnerabilities compared to newer specifications. The complete vulnerability data can still be included, though it utilizes the comment field. *Fulfilled.*

FR-4: Integration with existing SCA Tool services has been successfully implemented, as documented in the architecture chapter and Section 6.4. The system seamlessly interfaces with established service layers and data processing workflows. *Fulfilled.*

FR-5: Uniform configuration application is ensured through the common implementation approach detailed in Section 6.5.1. This guarantees consistent *sbom-*

[2]https://cyclonedx.github.io/cyclonedx-web-tool/validate

Config data processing across all supported formats and versions. *Fulfilled.*

FR-6: Template management operations have been fully implemented, providing comprehensive template creation, modification, and deletion capabilities as described in Section 6.2. *Fulfilled.*

FR-7: Role-based access control has been implemented using the existing *AuthorizationService* for permission validation and permission-based frontend component rendering. This ensures that access control is implemented appropriately throughout the system. *Fulfilled.*

FR-8: Template validation is implemented through dynamic option filtering mechanisms detailed in Section 6.6.2. The system presents only valid configuration options and validates the name within the creation/edit dialog, as detailed in Section 6.6.4, in real-time. *Fulfilled.*

FR-9: Dependency scope configuration has been implemented as described in Section 6.5.1, allowing flexible control over which dependency types are included in generated SBOMs. *Fulfilled.*

FR-10: All specified configuration options for component metadata and license inclusion are supported by the system, enabling comprehensive customization of SBOM content. *Fulfilled.*

FR-11: Vulnerability data inclusion is supported across all formats with format-appropriate implementations. Enhanced vulnerability support is available in CycloneDX and SPDX v3.0.1, while SPDX v2.3 utilizes comment fields for vulnerability information due to format limitations (see Section 6.5.3). *Fulfilled.*

FR-12: System extensibility has been achieved through separate generator classes implementing common interfaces with format-specific custom logic. New generators can be integrated through the *SbomGeneratorFactory* and invoked by controllers. Template configurations maintain backward compatibility when new options are introduced. *Fulfilled.*

FR-13: Legal notice integration has been implemented with the same license inclusion mechanisms in all generators as detailed in Section 6.5. Only the objects representing the legal data differ in each format, as required by the specification. *Fulfilled.*

7.2 Non-Functional Requirements Evaluation

NFR-1: SBOM generation performance evaluation was conducted through empirical testing with projects of varying component counts. For smaller projects, SBOM generation completes rapidly within acceptable timeframes. Initial testing

with larger projects, including SCA Tool itself, encountered 503 errors indicating internal server errors during processing. To facilitate systematic analysis of these failures, comprehensive logging through Sentry was implemented to capture detailed error information and performance metrics.

Subsequent testing successfully validated performance requirements for large-scale SBOM generation. A comprehensive test with approximately 2300 components completed in under 2 minutes, significantly exceeding the defined threshold of 5 minutes for a 2000-component SBOMs. The generated SBOM included complete dependency information, all available license data with full license texts, vulnerability information, and checksums, demonstrating full SBOM capability. One exception was identified in SPDX v2 format generation, which fails with an exception. While this issue has been addressed in the codebase, verification testing could not be completed within the thesis timeframe, as it requires deployment of the main branch. *Fulfilled.*

NFR-2: Template management operations were tested and demonstrated successful completion within the specified 3-second threshold under normal operational conditions. All standard template operations, including creation, modification, deletion, and retrieval, consistently perform within acceptable response times during typical usage scenarios.

However, performance evaluation was not conducted for extreme edge cases, such as systems containing millions of templates, as these scenarios do not represent practical or realistic use cases for the template management feature. The testing focused on realistic operational loads that reflect actual deployment environments and user workflows. As part of future work, implementing organizational limits on template creation could be considered to prevent potential resource exhaustion attacks through excessive template creation. *Fulfilled.*

NFR-3: Standards compliance has been achieved through implementation based on official SPDX and CycloneDX documentation, utilizing respective Java libraries for document construction and serialization. The generators implement format-specific requirements according to published specifications, ensuring adherence to SPDX v2.3, SPDX v3.0.1, and CycloneDX v1.5/v1.6 standards.

However, one implementation decision creates a potential conflict with the SPDX v2.3 specification. The current implementation utilizes *extractedLicensingInfo* for both custom licenses and listed licenses when the option is enabled, whereas the specification indicates that it should be reserved exclusively for custom or non-listed licenses. This design choice prioritizes user functionality by enabling the inclusion of license text for all license types, but introduces non-compliance with the specification. The decision to maintain this option versus adhering strictly to the specification remains under consideration. It may be resolved as part of future work to ensure the generation of only specification-compliant SBOMs. *Partially*

Fulfilled.

NFR-4: Generated documents successfully pass validation against official format schemas for CycloneDX v1.5/v1.6 and SPDX v3.0.1, with successful validation through respective official validation tools. However, as already stated in the evaluation of FR-1, SPDX v2.3 documents encounter validation failures for some documents due to issues with the extracted license information URI pattern compliance, which prevents full automated compliance checking for this format. *Partially Fulfilled.*

NFR-5: Semantic consistency for essential package information is maintained across SPDX and CycloneDX outputs through shared data processing pipelines that ensure identical package identifiers, versions, suppliers, checksums, and dependency relationships. However, format-specific capabilities introduce some semantic variations, such as CycloneDX's support for affected version ranges in vulnerability data and detailed risk scoring, while SPDX's JSON-LD serialization constraints limit certain vulnerability metadata representations, such as the *Double* datatype of the risk score. These differences reflect format strengths and weaknesses rather than consistency failures. *Fulfilled.*

NFR-6: Seamless integration with existing SCA Tool infrastructure has been achieved through consistent application of established controller and service layer patterns, utilization of existing data provider services, and integration with current user management and authorization systems. *Fulfilled.*

NFR-7: Comprehensive error handling has been implemented through appropriate exception throwing, SLF4J logging integration for operational monitoring, and Sentry error reporting for issue tracking across all deployments. The system generates descriptive error messages and detailed stack traces through Sentry, enabling efficient debugging and root cause analysis. *Fulfilled.*

NFR-8: Template creation and modification interfaces provide intuitive user workflows designed for operation without external documentation requirements. The intelligent configuration cascading and permission-aware interface design enable template management through self-explanatory interface elements and contextual guidance. The interface design suggests this requirement should be fulfilled based on the implemented usability features. However, formal testing with actual novel users was not performed to validate the intuitive operation assumption. *Not Fulfilled.*

NFR-9: Backward compatibility for existing templates has been achieved through frontend default value application, API request deserialization, and database entity mapping with automatic default handling. *Fulfilled.*

NFR-10: Modular architecture implementation successfully enables independent evolution of format-specific generation components through clear separation

of concerns and factory pattern generator selection. The architecture supports future SBOM standard additions through established extension patterns, without demanding modifications to existing implementation code. *Fulfilled.*

8 Conclusion

This thesis developed and implemented a configurable SBOM export system that addresses critical gaps in software composition analysis tooling. The system delivers multi-format SBOM generation capabilities supporting SPDX v2.3, SPDX v3.0.1, CycloneDX v1.5, and CycloneDX v1.6, integrated with a comprehensive template management framework that bridges the accessibility gap between technical and non-technical stakeholders.

Out of 23 defined requirements, 19 were fulfilled, three were partially fulfilled, and one was not fulfilled. The not-fulfilled requirement (NFR-8) concerns usability validation: while the system was designed for intuitive operation, formal user testing protocols were not conducted within the thesis timeframe. As a result, usability remains unverified until future deployments of the SCA Tool provide feedback from first-time users. The partially fulfilled cases reflect validation inconsistencies in SPDX v2.3 compliance and the trade-off between strict specification adherence and user functionality.

The modular architecture ensures extensibility for future SBOM standards while maintaining seamless integration with existing SCA Tool infrastructure. Key technical contributions include the implementation of consistent cross-format data processing, organizational template management, and comprehensive integration of legal notices within SBOM documents. The system successfully validates against official format specifications for CycloneDX formats and SPDX v3.0.1, demonstrating compliance with the latest industry standards.

Performance evaluation confirmed the system's capability to handle large-scale projects efficiently. Testing of the SCA Tool project itself, which contains approximately 2,300 components with all dependencies, legal, and vulnerability data included, resulted in the generation of an SBOM, completing in under two minutes. This demonstrates the system's ability to process substantial projects while maintaining acceptable performance thresholds.

Nevertheless, some limitations remain. SPDX v2.3 validation encounters inconsistencies due to apparent issues with the validator tool rather than systematic implementation defects. An identified exception in SPDX v2.3 format generation

remains unverified due to deployment constraints within the thesis timeframe. Finally, the trade-off between specification compliance and user functionality, particularly regarding SPDX v2.3 license text inclusion, highlights ongoing challenges in balancing adherence to standards with practical usability.

Due to time constraints, several enhancements and issue resolutions remain for future development. Performance optimization constitutes a primary focus area: implementing SBOM caching mechanisms would further improve generation time, while direct filtering would replace the current bulk request-then-filter approach.

Specification compliance issues demand further investigation. SPDX v2.3 validation inconsistencies require analysis to determine whether problems originate from validator implementation defects or necessitate Uniform Resource Identifier (URI) generation logic adjustments. The compliance conflict regarding *extractedLicensingInfo* usage needs resolution to ensure the generation of fully compliant documents.

Additional technical enhancements would strengthen system robustness and functionality. SPDX v3.0.1 risk score support, currently constrained by JSON-LD serialization limitations, becomes feasible through the recently merged fix in the SPDX Java v3 JSON-LD store library. Implementation of organizational template limits would prevent resource exhaustion attacks through excessive template creation. Asynchronous polling mechanisms for generation and download processes would improve user experience during large SBOM generation tasks.

These enhancements would strengthen the system's capability to serve as a comprehensive solution for organizational SBOM generation while maintaining the balance between accessibility and technical flexibility established by this research.

Appendices

A SPDX v3.0.1 License Relationship Example

This appendix demonstrates SPDX v3.0.1's graph-based license relationship model, showing how a single package (axios) connects to the declared and one concluded licenses with full license text through explicit relationship elements.

```
{
  "@context" : "https://spdx.org/rdf/3.0.1/spdx-context.jsonld",
  "@graph" : [ {
    "@id" : "_:creationInfo_0",
    "type" : "CreationInfo",
    "specVersion" : "3.0.1",
    "createdBy" : [ "https://spdx.scatool.com/.../Agent/Creator" ],
    "createdUsing" : [ "https://spdx.scatool.com/.../Tool/SCATool" ],
    "created" : "2025-09-18T15:04:10Z"
  },
  ...
  {
    "spdxId" : "https://spdx.scatool.com/.../Package/pkg:npm/axios@1.3.5",
    "type" : "software_Package",
    "suppliedBy" : "https://spdx.scatool.com/.../Agent/npmjs.org",
    "software_copyrightText" : "Copyright (c) 2016 Alex Indigo",
    "software_downloadLocation" :
        "https://registry.npmjs.org/axios/-/axios-1.3.5.tgz",
    "software_packageVersion" : "1.3.5",
    "verifiedUsing" : [ {
      "type" : "Hash",
      "algorithm" : "sha512",
      "hashValue" : "..."
    } ],
    "software_homePage" : "https://axios-http.com",
    "description" : "Promise based HTTP client for the browser and node.js",
    "name" : "axios",
    "software_packageUrl" : "pkg:npm/axios@1.3.5",
    "creationInfo" : "_:creationInfo_0"
  },
  ...
  {
    "spdxId" : "https://spdx.scatool.com/.../Relationship/pkg:npm/axios@
        1.3.5/hasDeclaredLicense/MIT",
    "type" : "Relationship",
    "relationshipType" : "hasDeclaredLicense",
    "to" : [ "https://spdx.scatool.com/.../LicenseExpression/declaredLic
        ense/MIT"
        ],
    "from" : "https://spdx.scatool.com/.../Package/pkg:npm/axios@1.3.5",
    "creationInfo" : "_:creationInfo_0"
```

```
  },
  ...
  {
    "spdxId" : "https://spdx.scatool.com/.../Relationship/pkg:npm/axios@1.3.5/hasListedLicense/MIT#1722701872",
    "type" : "Relationship",
    "relationshipType" : "hasConcludedLicense",
    "to" : [ "https://spdx.scatool.com/.../ListedLicense/MIT#1722701872" ],
    "from" : "https://spdx.scatool.com/.../Package/pkg:npm/axios@1.3.5",
    "creationInfo" : "_:creationInfo_0"
  },
  ...
  {
    "spdxId" : "https://spdx.scatool.com/.../ListedLicense/MIT#1722701872",
    "type" : "expandedlicensing_ListedLicense",
    "simplelicensing_licenseText" : "Permission is hereby granted, free
        of charge, to any person obtaining a copy\nof this software and
        associated documentation files (the \"Software\"), to deal\nin
        the Software without restriction, including without limitation
        the rights\nto use, copy, modify, merge, publish, distribute,
        sublicense, and/or sell\ncopies of the Software, and to permit
        persons to whom the Software is\nfurnished to do so, subject to
        the following conditions:\n\nThe above copyright notice and this
        permission notice shall be included in all\ncopies or substantial
        portions of the Software.\n\nTHE SOFTWARE IS PROVIDED \"AS IS\",
        WITHOUT WARRANTY OF ANY KIND, EXPRESS OR\nIMPLIED, INCLUDING BUT
        NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,\nFITNESS FOR A
        PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
        THE\nAUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM,
        DAMAGES OR OTHER\nLIABILITY, WHETHER IN AN ACTION OF CONTRACT,
        TORT OR OTHERWISE, ARISING FROM,\nOUT OF OR IN CONNECTION WITH
        THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE\nSOFTWARE.",
    "name" : "MIT",
    "creationInfo" : "_:creationInfo_0"
  },
  ...
  {
    "spdxId" : "https://spdx.scatool.com/.../LicenseExpression/declaredLicense/MIT",
    "type" : "simplelicensing_LicenseExpression",
    "simplelicensing_licenseExpression" : "MIT",
    "comment" : "Declared License",
    "creationInfo" : "_:creationInfo_0"
  },
  ... ]
}
```

B Configuration Menu UI

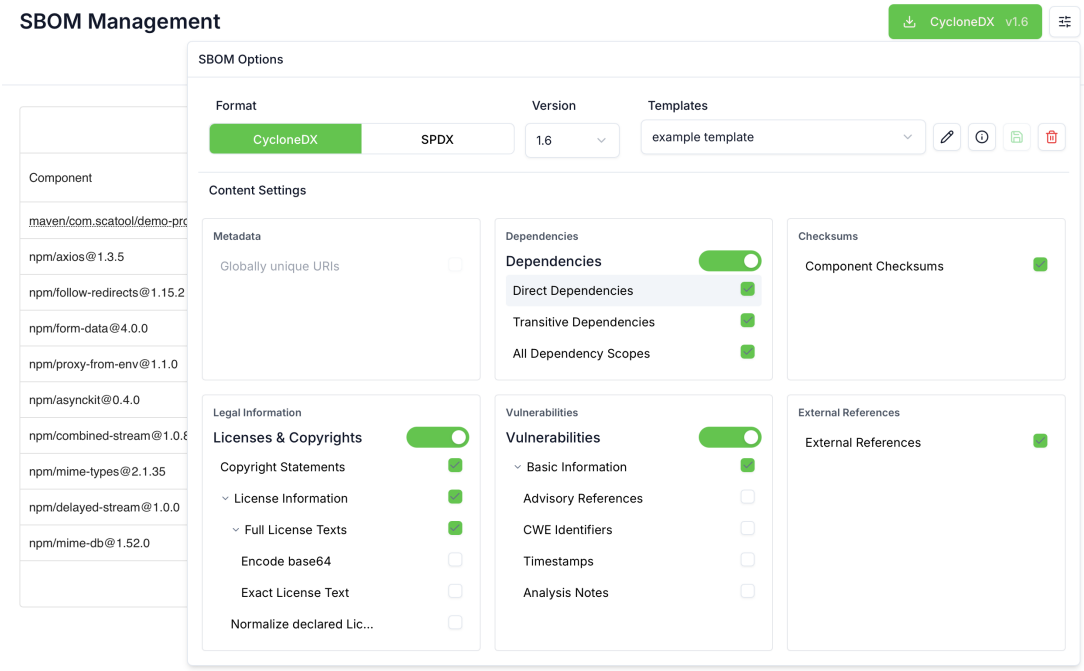

Figure 1: Configuration Tab with selected Template and the template management permission

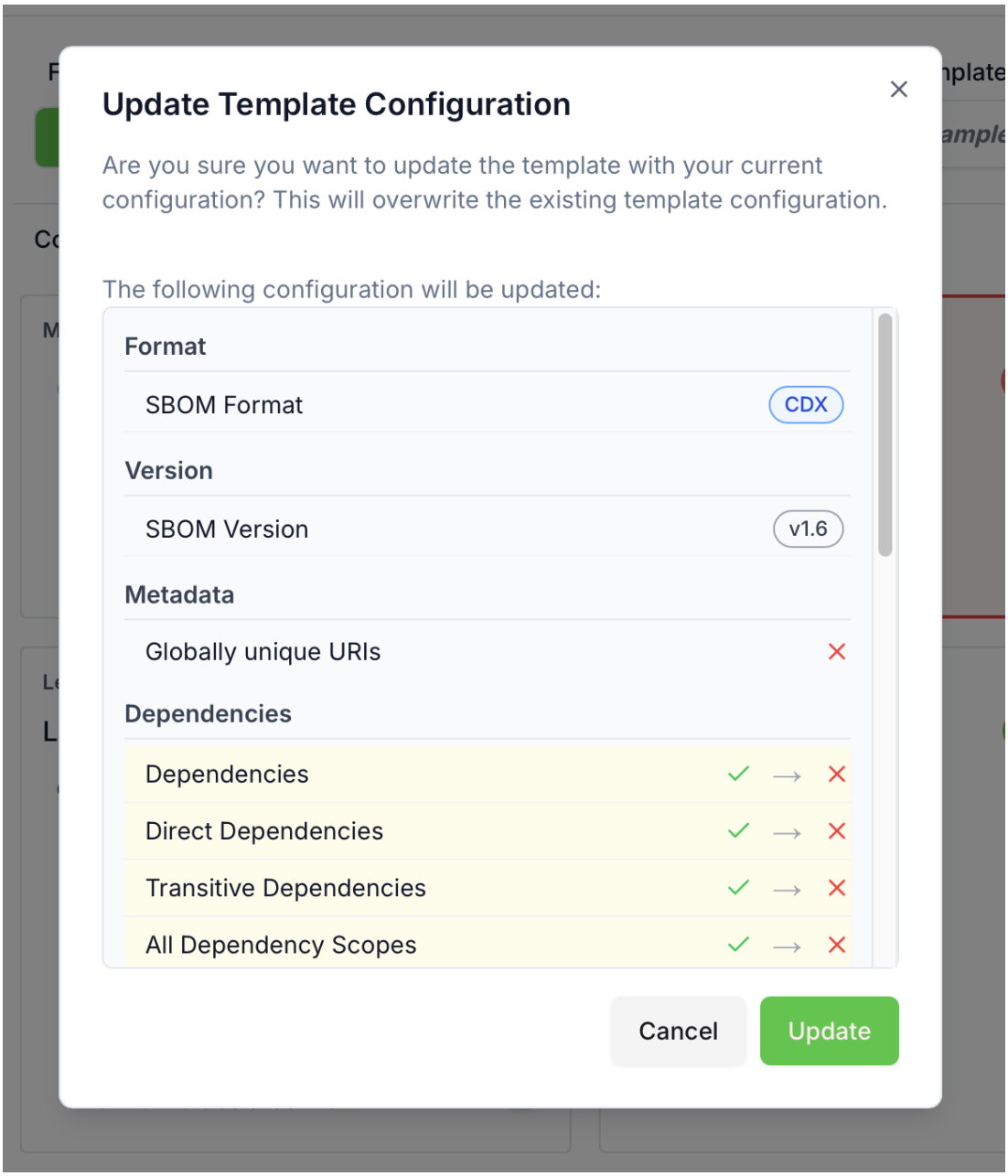

Figure 2: Template Update Dialog with format, version and configuration preview

References

Alrich T., Clark C., Dwyer P., Gandhi R., Weinrich A., Gates C., Herz J., Kruszewski D., Manion A., Martin B., Nandakumaraiah C., O'Connor B., O'Neall G., Sparrell D., Springett S., Stewart K., Walsh T., Waltermire D. & Winslow S. (2021). *Survey of Existing SBOM Formats and Standards* (tech. rep.). NTIA. https://www.ntia.gov/sites/default/files/publications/sbom_formats_survey-version-2021_0.pdf

Bi, T., Xia, B., Xing, Z., Lu, Q., & Zhu, L. (2024). On the Way to SBOMs: Investigating Design Issues and Solutions in Practice. *ACM Transactions on Software Engineering and Methodology*, *33*(6), 1–25. https://doi.org/10.1145/3654442

Bradner, S. O. (1997, March). *Key words for use in RFCs to Indicate Requirement Levels* (Request for Comments No. RFC 2119) (Num Pages: 3). Internet Engineering Task Force. https://doi.org/10.17487/RFC2119

BSI. (2024, September). *Technical Guideline TR-03183: Cyber Resilience Requirements for Manufacturers and Products* (tech. rep.). Federal Office for Information Security. https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/TechGuidelines/TR03183/BSI-TR-03183-2-2_0_0.pdf?___blob=publicationFile&v=3

CISA. (2025). *2025 Minimum Elements for a Software Bill of Materials (SBOM)* (tech. rep.). Cybersecurity and Infrastructure Security Agency, Department of Homeland Security. https://www.cisa.gov/sites/default/files/2025-08/2025_CISA_SBOM_Minimum_Elements.pdf

Dalia, G., Visaggio, C. A., Di Sorbo, A., & Canfora, G. (2024). SBOM Ouverture: What We Need and What We Have. *Proceedings of the 19th International Conference on Availability, Reliability and Security*, 1–9. https://doi.org/10.1145/3664476.3669975

Haddad, I. (2024, August). *Strengthening License Compliance and Software Security with SBOM Adoption: A Definitive SBOM Guide for Enterprises* (tech. rep.). The Linux Foundation. https://doi.org/10.70828/VHIN7583

Hannah, J. & Wilczynski, A. (2023, November). *Leveraging automotive SBOMs to enhance security and traceability* (tech. rep.). SBD. https://insight.sbdautomotive.com/rs/164-IYW-366/images/2200h-23%20SBD%

20Explores%20-%20Automotive%20SBOMs%20and%20Cybersecurity.pdf

Hiesgen, R., Nawrocki, M., Schmidt, T. C., & Wählisch, M. (2022, June). The Race to the Vulnerable: Measuring the Log4j Shell Incident. https://doi.org/10.48550/arXiv.2205.02544

Linux Foundation. (2022). SPDX Specification 2.3.0. Retrieved September 24, 2025, from https://spdx.github.io/spdx-spec/v2.3/

Linux Foundation. (2023). SPDX Specification 3.0.1. Retrieved September 24, 2025, from https://spdx.github.io/spdx-spec/v3.0.1/

Martínez, J. & Durán, J.M. (2021). Software Supply Chain Attacks, a Threat to Global Cybersecurity: SolarWinds' Case Study. *IIETA*, *Vol. 11*(No. 5), 537–545. https://doi.org/10.18280/ijsse.110505

Mirakhorli, M., Garcia, D., Dillon, S., Laporte, K., Morrison, M., Lu, H., Koscinski, V., & Enoch, C. (2024, February). A Landscape Study of Open Source and Proprietary Tools for Software Bill of Materials (SBOM). https://doi.org/10.48550/arXiv.2402.11151

NTIA. (2021, July). *The Minimum Elements for a Software Bill of Materials (SBOM)* (tech. rep.). United States Department of Commerce. https://www.ntia.gov/sites/default/files/publications/sbom_minimum_elements_report_0.pdf

OWASP Foundation. (2023). CycloneDX v1.5 JSON Reference. Retrieved September 24, 2025, from https://cyclonedx.org/docs/1.5/json/

OWASP Foundation. (2024, April). OWASP_cyclonedx-Authoritative-Guide-to-SBOM-en. Retrieved September 24, 2025, from https://cyclonedx.org/guides/OWASP_CycloneDX-Authoritative-Guide-to-SBOM-en.pdf

OWASP Foundation & Ecma International. (2024, April). CycloneDX v1.6 JSON Reference. Retrieved September 24, 2025, from https://cyclonedx.org/docs/1.6/json/

Riehle, D. & Harutyunyan, N. (2019). Open-source license compliance in software supply chains. In *Towards Engineering Free/Libre Open Source Software (FLOSS) Ecosystems for Impact and Sustainability* (pp. 83–95). Springer Singapore. https://doi.org/10.1007/978-981-13-7099-1

Sonatype. (2021). *The 2021 State of the Software Supply Chain Report* (tech. rep.). Sonatype. https://www.sonatype.com/hubfs/Q3%202021-State%20of%20the%20Software%20Supply%20Chain-Report/SSSC-Report-2021_0913_PM_2.pdf

Stalnaker, T., Wintersgill, N., Chaparro, O., Di Penta, M., German, D. M., & Poshyvanyk, D. (2024). BOMs Away! Inside the Minds of Stakeholders: A Comprehensive Study of Bills of Materials for Software Systems. *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 1–13. https://doi.org/10.1145/3597503.3623347

Stewart, K. (2022, October). SPDX and Software Bill of Materials ISO/IEC 5962L 2021. In A. Brock (Ed.), *Open Source Law, Policy and Practice*

(2nd ed., pp. 145–163). Oxford University Press. https://doi.org/10.1093/oso/9780198862345.003.0007

Wagner, M. (2023, June). *JavaScript User Interface License Compliance Best Practices* [Master's thesis, Friedrich-Alexander-Universität Erlangen Nürnberg].

Xia, B., Bi, T., Xing, Z., Lu, Q., & Zhu, L. (2023). An Empirical Study on Software Bill of Materials: Where We Stand and the Road Ahead. *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2630–2642. https://doi.org/10.1109/ICSE48619.2023.00219