International Journal of Computer Trends and Technology
ISSN: 2231–2803 / https://doi.org/10.14445/22312803/IJCTT-V73I12P103

Volume 73 Issue 12, 13-21, December 2025
© 2025 Seventh Sense Research Group®

Original Article

AI-Assisted Verification of SBOM Accuracy and Drift in Software Supply Chains

Karthikeyan Thirumalaisamy

Independent Researcher, Washington, USA.

Corresponding Author : kathiru11@gmail.com

Received: 23 October 2025 Revised: 28 November 2025 Accepted: 08 December 2025 Published: 27 December 2025

Abstract - *Modern supply chain security depends on Software Bills of Materials (SBOMs), which provide transparency and compliance verification and component origin tracking for complex software systems. The fast-paced nature of DevSecOps operations causes SBOM accuracy to deteriorate rapidly. The actual software artifact contents differ from the declared SBOM because dependencies change, build environments transform, and automatic updates occur. The software supply chain becomes vulnerable to version spoofing, dependency confusion, and unintentional untrusted component usage because of this metadata accuracy decline. The paper presents an AI-based system that detects SBOM drift and measures the extent of discrepancies between declared and actual software content. The system uses machine learning to compare software build artifacts with dependency graphs and component metadata to detect any discrepancies between declared SBOM information and actual software components. The system evaluates each discrepancy through a four-level severity assessment, which ranges from Critical to Low. The system blocks deployment of builds that show High or Critical issues until developers fix the underlying problems. The system performs continuous automated checks to enhance supply-chain security while minimizing human inspection requirements and maintaining development pipeline compliance standards. The system enables organizations to maintain a dependable software supply chain that supports fast-paced development methods.*

Keywords - *SBOM, Software Bills of Materials, Vulnerability Management, Supply Chain Security, AI Security, DevSecOps.*

1. Introduction

The software supply chain has become more complex, which makes it difficult to track dependencies and build artifacts through their entire lifecycle. The Software Bill of Materials (SBOM) functions as a solution that addresses third-party library and external dependency problems in software systems. An SBOM contains complete information about software components, including their licensing details and origin points. Industry-recognized standards, including SPDX and CycloneDX, enable organizations to create and distribute SBOMs through standardized formats. Organizations can detect vulnerabilities more effectively through these standards, which also support their legal requirements.

The generation of SBOMs occurs at particular development stages, which restricts their ability to maintain accuracy because source code, dependencies, and container images change. The SBOM document loses its accuracy when SBOM drift occurs because it no longer reflects the actual software components. Security threats emerge from the combination of low SBOM accuracy, confidence, and SBOM drift because it enables dependency confusion, version mismatches, and hidden unknown components. The current review methods, based on rules and manual checks, do not identify major inconsistencies that occur in DevSecOps environments that undergo fast-paced changes.

The paper creates an Artificial Intelligence (AI) system that operates across the CI/CD pipeline to check SBOMs and identify drift events for achieving complete SBOM accuracy. The system employs artificial intelligence to analyze SBOM statements against build results and dependency lists and digital authentication evidence to detect any discrepancies. The system evaluates detected inconsistencies through a severity rating system, which classifies them as Critical, High, Medium, or Low. The system will prevent build and release operations from executing when it identifies any High or Critical severity discrepancies. The system allows organizations to verify software product authenticity and regulatory compliance while maintaining developer freedom.

Organizations can achieve instant software product composition verification through intelligent SBOM validation, which reduces manual verification work and strengthens their software supply chain defenses. The paper begins by discussing the importance of AI-based verification. It then outlines the system's design and shares test results that demonstrate how software can verify itself and enforce policies.

2. Background and Motivation

This section provides an overview of the foundational concepts related to SBOMs, outlines their significance within contemporary software development workflows, and identifies key challenges in ensuring their ongoing accuracy.

2.1. Definition and purpose of SBOMs

An SBOM is essentially a complete, accurate, machine-readable listing of every component, dependency, library, and other item included in software. The typical details include the component name, the version number, the name of the supplier, and additional information about the component, including licensing information and the results of the cryptographic hash. This information is useful to companies because it identifies the precise items of software being used and where, in those items, potential weaknesses are located.

The main goal of SBOM is to improve the transparency and traceability in the software supply chain. With an accurate and current inventory of the components of the software, developers and security professionals can easily find and assess known vulnerabilities, check on licensing requirements, and quickly react to new threats. In addition, SBOMs are also a key part of regulatory and compliance environments, in which companies need to prove they have knowledge and control over third-party and open-source software used in their products.

In summary, the SBOM provides a digital footprint for software composition, so there is visibility into the process from the time the developer begins building through the time the software is deployed. A correct SBOM enables the basis for managing vulnerabilities, assessing risk, and establishing good practices for the governance of secure software.

2.2. SBOM Generation Tools Across Languages

Tools to generate a Software Bill of Materials (SBOM) are available in all ecosystems, but the way these tools obtain dependency data differs by the specific build system and package manager for each language ecosystem. The overall objective remains the same: to determine the software composition of your application, though the tools you will use, the formats they produce, and the degree of support they have for integrating with your development workflow depend upon which ecosystem you work in, how mature that ecosystem is, and what metadata is available within that ecosystem.

Below is a list of some of the most popular SBOM generation tools per language ecosystem:

Table 1. SBOM Generation Tools Across Languages

Language	Tools	Formats	Notes
.NET / C#	Microsoft SBOM Tool, CycloneDX .NET, Syft	SPDX, CycloneDX	Integrated with Azure DevOps, it supports NuGet and DLL metadata.
Java / JVM	CycloneDX Maven/Gradle Plugin, Syft, ORT (OSS Review Toolkit)	SPDX, CycloneDX	Reads POM and Gradle metadata, supporting JAR and WAR packaging.
Python	Syft, CycloneDX Python, pip-audit, FOSSA CLI	SPDX, CycloneDX	Extracts dependency trees from requirements.txt and virtual environments.
JavaScript / Node.js	CycloneDX Node, Syft, npm audit	SPDX, CycloneDX	Integrates with package-lock.json and yarn.lock for full dependency maps.
C/C++	Syft, FOSSology, SCANCODE Toolkit	SPDX	Parses Make/CMake files and library manifests; requires more manual configuration.
Go (Golang)	Syft, CycloneDX Go, Go SBOM Generator	SPDX, CycloneDX	Supports Go modules; can analyze static binaries for embedded dependencies.
Container Images	Syft, Anchore, Tern, Docker SBOM (BuildKit)	SPDX, CycloneDX	Analyzes OS packages and embedded language layers inside container images.

3. The Challenge of SBOM Accuracy and Drift

Modern SBOM generation tools deliver excellent software component visibility, yet organizations face major difficulties when it comes to keeping their SBOMs up to date. The process of generating SBOMs happens during build or release operations to record dependency information at that particular time.

The rapid changes in development environments create SBOM drift because the declared SBOM no longer matches the actual software components, and the following sections will explain this in detail.

3.1. Automatic Dependency Updates

The current default mode of operation in modern software package managers is to obtain the most recent stable and secure versions of packages and libraries. Therefore, most package managers will automate the resolution of dependencies, especially when it comes to patch level and minor updates. Although there are some benefits to automating the dependency resolution with regard to security and performance, this automation can lead to unintentional SBOM drift.

SBOMs are typically created at a single point in time. Typically, this occurs either during the initial build or release process of an application. When the SBOM captures a dependency version such as Library X v1.2.0, and the package manager then subsequently resolves Library X v1.2.3, the SBOM does not reflect the components that make up the final artifact. The difference between the SBOM and the actual components used in the final artifact may remain undetected due to several reasons:

- Automatic updates of patch levels occur without the developers making conscious decisions about what updates are made.
- Many build processes have "floating" versions that are specified in the dependency declaration, e.g., ^1.2 or 1.2.x.
- Many lock files are not consistently updated or committed.
- Different build agents may resolve dependencies differently due to different cache states or states of their repositories.

The continued occurrence of these silent and frequent updates results in the declared SBOM being out of sync with the dependency tree that was actually built. Although the updates were minor, they may introduce new vulnerabilities, change how components interact, or create

Licensing issues that the outdated SBOM does not identify. Mechanisms to accurately provide visibility require methods to detect and reconcile the automated version shifts as part of ongoing build and validation processes.

3.2. Transitive Dependency Changes

The problem with transitive dependencies (i.e., packages that are pulled in as part of another package) is that while most applications rely upon both the direct and indirect library packages, many of these indirect packages do not remain static – they tend to evolve based on updates made to the direct package and/or the package that provided the original library. Since the primary application is typically unaware of the transitive dependencies that were established during build time, this means there will likely be times when transitive dependencies are updated (either automatically during the build process or as a result of the maintainer of one of the transitive dependencies updating their dependency list). Yet, the SBOM has not been refreshed to reflect those changes. There are several common ways in which SBOM drift occurs as a result of this dynamic:

- As a result, the developer of an upstream package chooses to update the version of the library being used by that package.
- As a result of security patches being applied to lower-level libraries.
- As a result of the replacement or removal of a component from a third-party package.
- Due to the fact that the build-time resolution of different transitive versions of libraries may differ as a result of caching or environmental conditions.

Since these updates to transitive dependencies occur independently of the application's own source code, developers are often not aware that these updates have occurred and therefore may not manually refresh the SBOM. As such, SBOM drift can lead to vulnerabilities or outdated sub-components being masked, thereby hindering the ability to gain a comprehensive view of the entire dependency tree of the application.

Since many projects' total number of components will be comprised primarily of transitive dependencies, failure to monitor changes to these dependencies accurately will diminish the trustworthiness and comprehensiveness of the SBOM.

3.3. Environment-Specific Builds

The software components included in a build may be influenced by the environment in which the build is performed. The operating system, the container base image, the build agent, and the system packages installed on the build server are all examples of how the environment can influence the software components included in a build, with varying results in terms of the final artifact created.

However, the source code for the application has not been changed. Differences in the build environment commonly cause what is referred to as environment-specific drift. Environment-specific drift typically occurs because of one or more of the following factors:

- OS-level packages differ. Examples of this would include an Alpine-based container image versus a Debian-based container image, both of which contain differing system libraries.
- Differing compilers or runtimes result in additional native dependencies being introduced. Examples of this would include updated versions of .NET, Java, or Python runtimes.
- Tools, SDKs, or cached packages are locally installed by the build agent and therefore differ between build servers.
- Resolution of platform-dependent packages may result in different binaries or libraries being selected by the package manager based upon the specific platform (architecture) the binary or library is intended to support.

As a result of using SBOM generation in a singular environment while executing builds across multiple environments and/or stages, the SBOM may not be entirely representative of the dependencies contained within the final output. Inaccurate representations of the dependencies contained within the final output can lead to oversight of various components, inconsistency of vulnerability scan results, and diminished reliance on the SBOM's ability to describe the software dependencies used in the development process completely.

Given the fact that modern CI/CD pipelines frequently utilize distributed and ephemeral build environments, it is imperative to address the issue of environment-specific drift to ensure the accuracy and reliability of the SBOM throughout the entire software life cycle.

3.4. Cached and Layered Builds

While many build tools today use caching and layering to accelerate build times and streamline CI/CD pipelines, both of which offer significant advantages in terms of efficiency, they can contribute to an accumulation of non-declared, but still present, components in the build, which ultimately contributes to drift.

The way many container-based builds operate is by using layers that are built upon other layers (previous builds), and will reuse layers from previous builds if there has been no detection of change to those layers. Layers reused from previous builds will contain all libraries and OS packages that existed at the time those layers were created, regardless of how old they may be.

Build tools such as Docker, Gradle, Maven, and the .NET incremental compiler are also designed to utilize cached artifacts, and in doing so, do not re-resolve nor verify dependencies against the original source(s).

Caching of this nature results in several common examples of drift, including:

- Layers from previous container-based builds are being used to create new containers, with older packages/libraries included in the new containers.
- Components (i.e., modules) that were previously compiled (e.g., C#) and subsequently cached are rebuilt, and newer binary images are produced, while the cached image remains unchanged.
- Cached local packages (e.g., NuGet, npm, pip) resolve to dependencies that have not changed since the last version was installed and resolved.
- CI/CD cache restore mechanisms restore older versions of artifacts during pipeline execution.

Since the SBOM is typically generated from the declared dependency files (not the actual cached artifact), it will likely exclude any component that has accumulated through the caching process.

Therefore, the disconnect between what should be represented in the SBOM and what is actually represented in the deployed application can result in vulnerabilities/deprecated dependencies that would otherwise be identified based on build processes that trigger rebuilds due to changes that are ignored because of their potential impact on performance.

Over time, in high-frequency release pipelines, these caching mechanisms can lead to SBOMs that are technically correct (in terms of compliance with regulations) but represent an inaccurate representation of what exists in the deployed application.

3.5. Manual or Ad-Hoc Changes

In many cases, especially in rapidly evolving projects, an organization will make updates outside of its normal build or dependency management process. When they do so, it is common for them to cause SBOM Drift as the component has been added, updated, or removed from the system, but there was no trigger to have the entire SBOM rebuild. Examples of such practices include:

- Directly patching or hotfixing deployed binaries.
- Replacing DLLs or Libraries manually while performing debugging or troubleshooting.
- Adding temporary dependencies into your test environment that do not get added back into your project files.
- Using local development overrides that utilize untracked local packages or custom libraries.
- Making emergency fixes within a Production Environment by bypassing the standard pipeline.

These types of manual changes are typically done without creating an automated record of those changes. Therefore, security and compliance teams do not know about undocumented components that are included in the final product.

As these "small" exceptions continue over time, they begin to greatly affect the accuracy and dependability of the SBOM. The only way to prevent this type of drift is through establishing governance and through automated processes that ensure the actual software contents being used, and not just the declared dependencies.

4. AI-Assisted Verification Framework

As SBOM Drift occurs increasingly in modern pipeline environments, we are at a point where Static generation alone will be insufficient to provide for the required reliability and Trust. Organizations must therefore have mechanisms that continually verify that the stated SBOM matches the actual software being built by comparing it against the actual Build output at every build step. In this Section, an Artificial Intelligence-assisted verification framework will be introduced. The Framework utilizes Artificial Intelligence (AI) to automatically determine build Artifacts, dependency manifests, and component metadata to identify any differences between them; classify the differences by severity and ensure that the SBOM remains aligned to the current state of the software through the entire development lifecycle.

4.1. Architecture Overview

The AI-assisted verification framework is a modular platform to be integrated into current CI/CD frameworks, and will use an automated method (AI) to compare the actual build outputs of a software application to its stated SBOM (Software Bill of Materials). The Framework has four primary modules with AI incorporated into both the verification module and the risk assessment module; see the following illustration for a high-level overview of how all of the modules interact with each other.

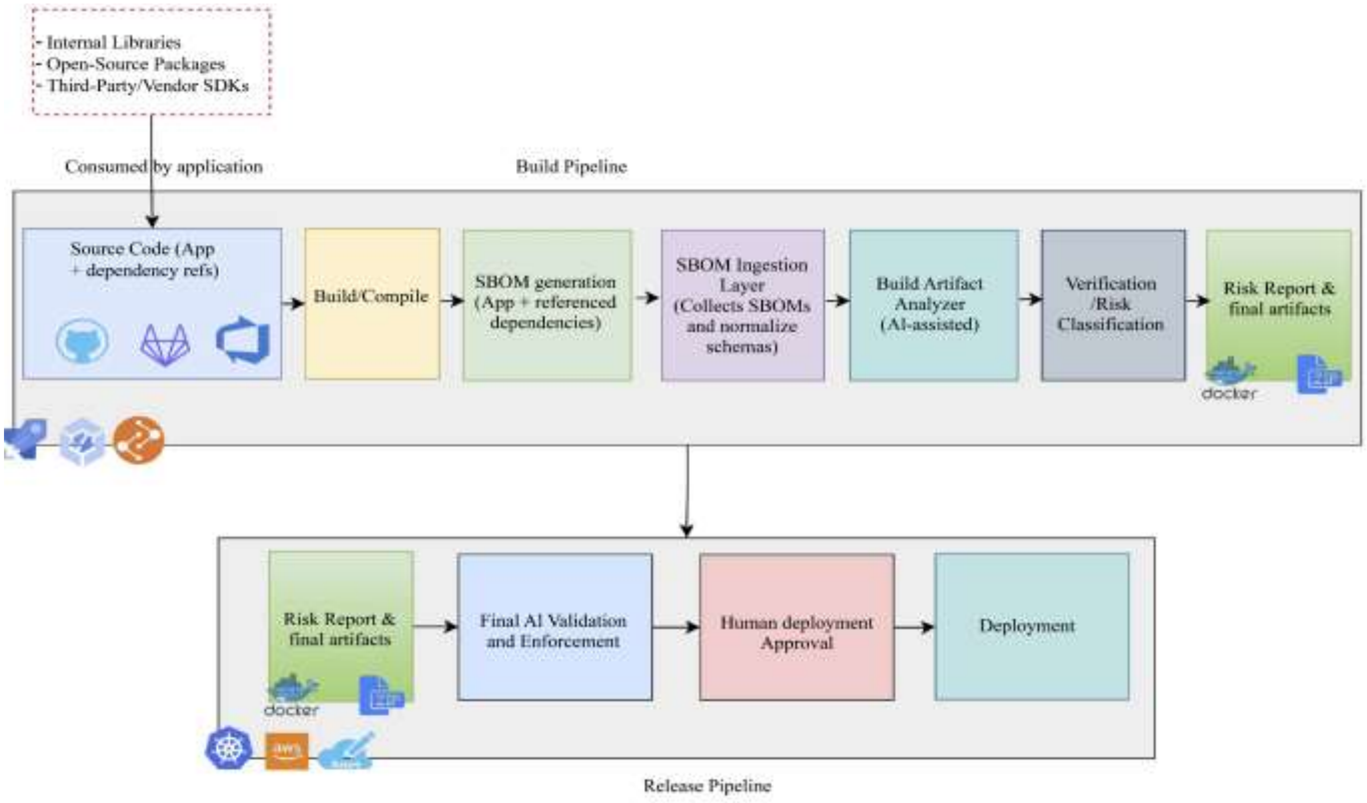
- Internal Libraries
- Open-Source Packages
- Third-Party/Vendor SDKs
Consumed by application
Build Pipeline
Source Code (App + dependency refs)
Build/Compile
SBOM generation (App + referenced dependencies)
SBOM Ingestion Layer (Collects SBOMs and normalize schemas)
Build Artifact Analyzer (AI-assisted)
Verification /Risk Classification
Risk Report & final artifacts
docker
Risk Report & final artifacts
docker
Final AI Validation and Enforcement
Human deployment Approval
Deployment
aws
Release Pipeline

Fig. 1 Illustration of AI-driven Trust Graph architecture

4.2. Compile Source Code

The pipeline process starts with the compilation of the application's source code to create binary objects, libraries, or container images. The build process will include the entire dependency graph necessary to generate the final product through the retrieval of all dependencies declared in the application's project files by the build tool/package manager; and also including any other transitive dependencies; and/or platform specific objects/components and/or any runtime libraries that were automatically included during the build process or when the build tool/framework was installed.

Depending on how many different environments are involved (local builds, CI servers, containerized builds, etc., it may cause different dependency resolution due to differences in available dependency caches, base images, or runtime versions. Due to these differences, the build/compile phase is typically the first place where minor differences between the final set of components can be introduced. Maintaining accuracy and consistency at this level is crucial.

4.3. SBOM Generation

Once the application's source code was successfully compiled by the system, the system produces a Software Bill of Materials (SBOM), which details the components the build process believes make up the application. Typically, the SBOM contains all of the direct dependencies associated with the application; the internal libraries used by the application; and metadata pulled from project configuration files, package manifest files, or other files used in conjunction with the build process.

Normally, SBOM tools capture the detailed information about each component included in the application, including, but not limited to, name, version, license, and supplier information. However, because these SBOMs are created prior to post-build operations such as building containers, creating binaries, or injecting run-time components into an application, the SBOM may represent the intended dependency list rather than the actual dependency list.

Therefore, the SBOM produced at this time is referred to as the declared SBOM and is used as the primary documentation of what the developer declares the application contains. The developer must ensure the declared SBOM accurately identifies the components contained within the application, as the declared SBOM will be used as the basis for later verification processes in the pipeline.

4.4. SBOM Ingestion Layer

The SBOM Ingestion Layer serves as the standardization and preparation phase for verification. Since SBOMs are created using various formats, i.e., SPDX, CycloneDX, etc., and/or language-specific metadata files, the ingestion layer creates a standardized version of the SBOM. As a result, the various analyses that occur after ingestion can operate on a common model, regardless of the environment/tools used to produce the SBOM.

As part of the ingestion process, the system verifies the SBOM structure, extracts key elements (i.e., component name, version, hash, license) from the SBOM, and resolves reference(s) to the dependency graph. In addition, the ingestion layer identifies any potential errors (i.e., incomplete metadata, missing field(s), inconsistent formatting) within the SBOM that may affect the accuracy of verification. Through the standardization/validation of the declared SBOM during ingestion, the ingestion layer provides a reliable foundation upon which comparisons will be made between the declared SBOM and the actual build artifacts. Therefore, the ingestion layer represents the transition from an unformatted raw SBOM to a normalized model from which accurate changes can be detected.

4.5. Build Artifact Analyzer (AI-Assisted)

The Build Artifact Analyzer examines build process outputs to determine which components actually exist in the final software product. The analyzer detects actual content within compiled binaries, packages, and container images, whereas the SBOM shows what the build system planned to include.

The analyzer checks different types of artifacts based on the technology stack used in the application.

- The analyzer checks .NET assemblies together with their associated NuGet packages.
- The analyzer checks Java applications through their JAR and WAR file formats.
- The analyzer checks Python applications through their wheel packages and distribution folders.
- The analyzer checks Node.js applications through their npm package directories.
- The analyzer checks native binary files and compiled modules.
- The analyzer checks container images and their operating system-level layers.

The analyzer extracts component names and versions, file hashes, nested dependencies, and runtime libraries and configuration files, and all additional modules that entered the build environment through implicit inclusion from these artifacts.

The system achieves better results at this stage through AI implementation, which enables it to:

- The system detects dependencies that third-party build tools bundle with other libraries through renaming or merging operations.
- The system detects non-standard build methods that third-party build tools implement.
- The system uses artificial intelligence to detect hidden dependencies that lack official documentation.
- The system detects unusual files that differ from standard build operations.
- The system handles situations where metadata exists in incomplete, missing, or inconsistent states.

The analyzer produces an exact actual build Composition which shows all components from the final artifact, regardless of their inclusion method. The information gathered during this phase enables the verification process to compare the declared SBOM with the actual dependency set. The analyzer produces an exact actual build composition, which shows all components from the final artifact regardless of their inclusion method. The information gathered during this phase enables the verification process to compare the declared SBOM with the actual dependency set.

4.6. Verification and Risk Classification

After receiving both inputs, the declared SBOM that was created as part of the build process and the actual build composition, which is derived from the final build output, the system will then perform an organized and thorough validation of these two items. The organization validates the declared composition (SBOM) of the software with its actual composition at this time. This is critical for validating the integrity, completeness, and accuracy of the SBOM before moving further into the pipeline.

The organization identifies discrepancies between the declared and actual composition of the software through this validation process; discrepancies in this regard could be due to unaccounted-for dependencies, versions that do not match, and/or files not accounted for in the declaration of the software's component parts.

Any such discrepancies would identify a deviation in how the software has been configured or developed and/or potentially expose the user to supply chain risks. Therefore, this validation phase is the basis of all subsequent phases of the organization, including the validation of the risk score assigned to the software and any resultant enforcement actions.

4.6.1. Verification Process

The verification tool will compare every item on the declared SBOM with the items found during the artifact analysis and verify the following:

- Missing Items: Declared items missing from the analyzed build.
- Added Items: Undeclared items in the build artifact.
- Version Drift: The versions of declared items do not match the build artifact's version of the item (for example, if a vendor has updated an item's patch level automatically).
- Unexpected Transitives: Upstream dependencies nested within other items were not accounted for in the SBOM.
- Environment Specific Differences: Items have been added due to variations of the Operating System used, how the application was run-time, or base image layers used.
- Anomalous or Suspicious Items: Build artifacts deviating from past build practices, vendor standards, or expected dependency diagrams.

A drift report is generated summarizing all discrepancies between the declared SBOM and the actual build items.

4.6.2. Risk Classification

The Risk Classification module evaluates each of the identified anomalies and assigns a risk level based on the severity level of the anomaly as follows:

Critical: Evidence of tampering; Malicious additions to the package; Inaccurate binary files; Missing essential core component(s). Releases are blocked.

High: Significant versions of software in use by the application; SBOM entries missing; Unexpectedly added dependencies with security risks. Release will require intervention prior to being released.

Medium: Minor errors that do not pose security risks but may cause the SBOM to be less accurate. (Example: Metadata drift.)

Low: The issue is cosmetic in nature, does not represent a security concern, and has no material impact.

Risk Report: These values form the Risk Report that is generated at the end of the validation phase. This Risk Report is used for input into the build pipeline (to provide developer feedback) and for final decision-making in the release pipeline.

4.7. Release Pipeline: Final Validation and Deployment

Once the build pipeline verifies the initial build results and creates the risk report and artifacts, it moves to the Release Pipeline for final and authoritative validation of the software to be released or distributed. The purpose of this stage is to ensure the software being released/distributed was completely validated, contains no unknown/unintended components, and matches exactly the SBOM provided as part of the release.

4.7.1. Final AI Validation

Once the artifact leaves the build pipeline and has produced the first risk report, the artifact then moves into the release pipeline. At this point, there is one last, definitive validation process in place to validate the final artifact, as well as the SBOM declared, to ensure that no items have been altered (drifted), replaced, or added unexpectedly from the time the artifact was built through packaging, signing, containerization, or staging. Final AI validation does another full scan of the artifact being prepared for release, and then makes a comparison between what was declared by the SBOM versus the actual contents of the artifact, and then also correlates all discrepancies related to internal, open source, and third-party dependencies.

The system then strictly enforces policies based on these findings; if critical or high-level discrepancies exist within the SBOM, the system will not allow the release of the artifact. If medium-level discrepancies exist, they will be identified for follow-up action. If low-level discrepancies exist, they will be documented for informational purposes only. Once an artifact has completed the final validation process and has met the requirements to move forward to production, a validated SBOM will be released with the artifact to ensure that all artifacts deployed to production include only components that have been validated and trusted.

4.7.2. Deployment with Verified SBOM

Following a successful final validation process with no blocks identified, the artifact is formally cleared for release into Production. The release pipeline ensures that:

- The accurate and verified SBOM accompanies the artifact in deployment
- Risk reports generated during validation are logged in internal audit records
- Any downstream Systems (e.g., Registries, Repositories, etc.) will be provided with validated components only

Thus, it is ensured that the version of software moving into production contains an SBOM that accurately reflects its components and that each of those component dependencies has been individually validated.

5. Conclusion

In order to provide a secure foundation for today's Software Supply Chain Security (SSCS), as more and more application development relies on internal components, open-source libraries/dependencies, and third-party libraries/dependencies, it has become imperative to ensure the accuracy and reliability of Software Bills of Materials (SBOM). Ensuring a reliable and accurate understanding of what exactly is contained within a release of an application is becoming more difficult as well as more important than ever. The paper creates a structured system to verify the declared components in SBOM against the actual content of build and release outputs.

The Framework operates through AI-based artifact analysis and multiple dependency source verification steps and risk-based severity assessment to build an automated system that detects build discrepancies and deployment risks of unverified software. The release pipeline validation of SBOM completeness and accuracy confirms that every deployment contains a complete and accurate SBOM, which supports software life cycle integrity.

Organizations must implement verification-based SBOM best practices into their current frameworks because these practices will help them achieve better supply chain transparency and fulfill emerging SSCS standards compliance while strengthening their systems against undetected compromised components.

References

[1] Lennard Helmer, Lisa Fink, and Maximilian Poretschkin, "Utilizing SBOM for Transparent AI Risk Communication," *Proceedings of the AAAI Symposium Series*, vol. 7, no. 1, pp. 185-189, 2025. [CrossRef] [Google Scholar] [Publisher Link]

[2] Rio Kishimoto et al., "A Dataset of Software Bill of Materials for Evaluating SBOM Consumption Tools," *IEEE/ACM 22nd International Conference on Mining Software Repositories*, Ottawa, ON, Canada, pp. 576-580, 2025. [CrossRef] [Google Scholar] [Publisher Link]

[3] Wataru Otoda et al., "SBOM Challenges for Developers: From Analysis of Stack Overflow Questions," *IEEE/ACIS 22nd International Conference on Software Engineering Research, Management and Applications*, Honolulu, HI, USA, pp. 43-46, 2024. [CrossRef] [Google Scholar] [Publisher Link]

[4] Menghan Wu et al., "More Than Meets the Eye: On Evaluating SBOM Tools in Java," *ACM Transactions on Software Engineering and Methodology*, pp. 1-28, 2025. [CrossRef] [Google Scholar] [Publisher Link]

[5] Serena Cofano, Giacomo Benedetti, and Matteo Dell'Amico, "SBOM Generation Tools in the Python Ecosystem: An In-Detail Analysis," *IEEE 23rd International Conference on Trust, Security and Privacy in Computing and Communications*, Sanya, China, pp. 427-434, 2024. [CrossRef] [Google Scholar] [Publisher Link]

[6] Hamed Okhravi, Nathan Burow, and Fred B. Schneider, "Software Bill of Materials as a Proactive Defense," *IEEE Security & Privacy*, vol. 23, no. 2, pp. 101-106, 2025. [CrossRef] [Google Scholar] [Publisher Link]

[7] Priyanshu Anand, Why Automated SBOM & Continuous Validation are Mission-Critical, 2025. [Online]. Available: https://technologymatch.com/blog/why-automated-sbom-continuous-validation-are-mission-critical

[8] Swaroop Sham, SBOMs: The Foundation of Software Supply Chain Security, 2025. [Online]. Available: https://www.wiz.io/academy/software-bill-of-material-sbom

[9] Joshua Burgin, The Top 6 Open-Source SBOM Tools. [Online]. Available: https://www.upwind.io/glossary/the-top-6-open-source-sbom-tools

[10] Flashpoint, SBOM 102: How to Operationalize SBOM Data into Real-Time Vulnerability Management, 2025. [Online]. Available: https://flashpoint.io/blog/sbom-operationalize-vulnerability-management/

[11] Microsoft, Code Integrity Checking, 2023. [Online]. Available: https://learn.microsoft.com/en-us/windows-hardware/drivers/devtest/code-integrity-checking

[12] ZJYLWL, Understanding SBOM Drift: What it is, Why it Matters, and How to Address it, 2025. [Online]. Available: https://zjylwl.com/archives/153

[13] interlynk-io, sbomqs: The Comprehensive SBOM Quality & Compliance Tool. [Online]. Available: https://github.com/interlynk-io/sbomqs?tab=readme-ov-file

[14] Josh Bressers, Fast and Furious: Doubling Down on SBOM Drift, 2022. [Online]. Available: https://thenewstack.io/fast-and-furious-doubling-down-on-sbom-drift/

[15] Scribe, Identifying Vulnerabilities with a Software Bill of Materials: Ensuring Security, Transparency, and Compliance. [Online]. Available: https://scribesecurity.com/blog/recent-software-supply-chain-attacks-lessons-and-strategies/
[16] Finite State, SBOM The Best SBOM Generation Tools Compared (& How to Pick the Right One), 2024. [Online]. Available: https://finitestate.io/blog/best-tools-for-generating-sbom
[17] Amazon, Amazon Inspector SBOM Generator. [Online]. Available: https://docs.aws.amazon.com/inspector/latest/user/sbom-generator.html