A Metadata-Driven Framework for Classifying Software Supply Chain Risk in Python Package Ecosystems

Ramkumar Sundarakalatharan

Information Security Group Royal Holloway University of London
Egham, Surrey, TW20 0EX United Kingdom

A Metadata-Driven Framework for Classifying Software Supply Chain Risk in Python Package Ecosystems

Submitted as part of the requirements for the award of the

MSc in Information Security

at Royal Holloway, University of London.

Acknowledgements

I would like to express my sincere gratitude to **my Supervisor** for his invaluable guidance, thoughtful feedback, and steady encouragement and patience to answer all my illogical questions, throughout this work.

I am also grateful to **Dr Geraint Price** (*Senior Lecturer, Department of Information Security*) for his support and constructive advice at key stages of the project.

My thanks to **Dr Nguyen Khoi Tran** (*University of Adelaide*) for the generosity of his time and insight, freely shared despite the distance and time zones. His perspectives helped shape the early direction of this research.

I am indebted to members of the **Surrey Cybersecurity Cluster** for providing invaluable, industry-grounded input on the initial design and evaluation approach.

To **my co-founder and team at Zerberus**, thank you for affording me the time and flexibility to pursue this project alongside our work.

Finally, to my son**, Anithran Imayavaramban,** and **my wife, Bhuvana**: thank you for your patience and unwavering support in allowing me to pursue this ambition. Your confidence in me, never dismissing it as a mere midlife detour, made this possible.

I hope to do justice to your kindness.

All remaining errors are my own.

Abstract

The growing reliance on open-source software has amplified supply chain risks in ecosystems such as PyPI, where threats include typosquatting, package abandonment, and declared–installed version mismatches. Addressing these requires lightweight methods that do not depend on source code or executing untrusted packages.

This dissertation introduces **ZSBOM**, a metadata-driven Command Line Interface (CLI) tool that extracts and classifies package risks across five dimensions: CVEs, CWE mappings, version drift, package abandonment, and typosquatting. **ZSBOM evaluates both direct and transitive dependencies, ensuring risk coverage across entire package graphs.** It operates entirely in a black-box mode, generating and evaluating SBOMs without installing dependencies. An open-source implementation enables seamless integration into Continuous Integration/Continuous Deployment (CI/CD) pipelines.

Validation against 200 Python packages, benchmarked with Syft+Grype and Snyk, shows that ZSBOM achieves 100% coverage of ground-truth malicious cases while surfacing additional risks—particularly typosquatting and abandonment—that CVE-centric tools miss. Weighted scoring reduces noise from low-severity cases, ensuring warnings remain actionable without blocking pipelines. Runtime tests confirmed efficiency (231s vs 420s for Grype and 590s for Snyk).

These findings demonstrate that complementing CVE and CWE data with exploit-aware metadata signals can deliver reproducible, pre-installation triage for software supply chains. ZSBOM thus provides actionable, ecosystem-specific risk scores and lays the groundwork for expansion beyond Python..

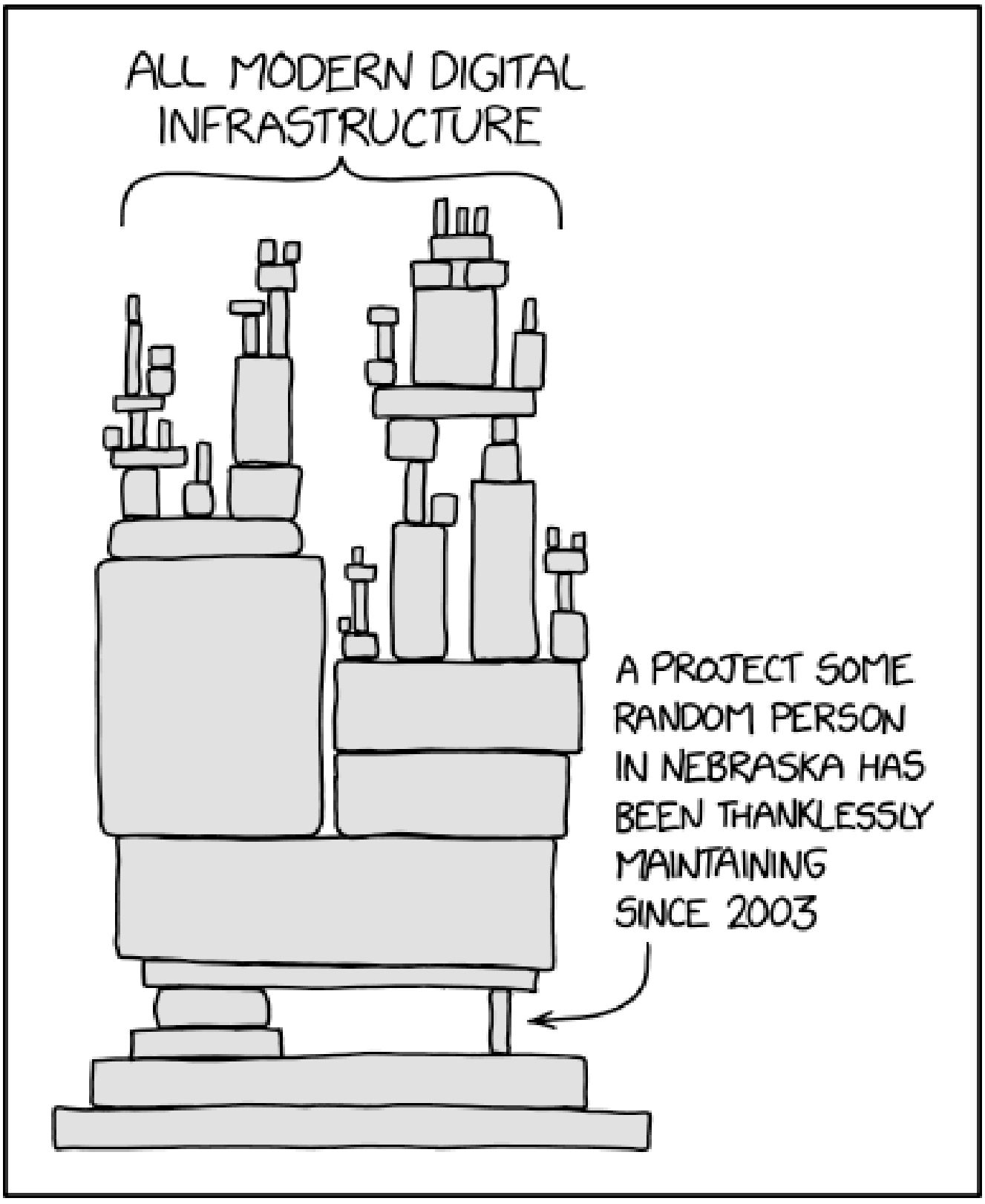


Figure 1: An XKCD Comic image depicting the Impact of Open Source Components in Modern Software Ecosystem (Source: XKCD)

Table of Contents

List of Figures

List of Tables

Index of Appendices

Abbreviations:

Term/Acronym	Full Phrase / Meaning	First Mention (Page No.)
API	Application Programming Interface	1
ASCII	American Standard Code for Information Interchange	31
ASGI	Asynchronous Server Gateway Interface	Appendix
AWS	Amazon Web Services	Appendix
C4	Context, Container, Component & Code (Modelling)	19
CI/CD	Continuous Integration / Continuous Deployment	Abstract
CLI	Command-Line Interface	Abstract
CVE	Common Vulnerabilities and Exposures	12
CVSS	Common Vulnerability Scoring System	18
CWE	Common Weakness Enumeration	14
FAR	False Acceptance Rate	65
FN	False Negatives	64
FP	False Positives	64
FRR	False Rejection Rate	65
GHSA	GitHub Security Advisory	45
HTTP	Hypertext Transfer Protocol	Appendix
ISO 8601	International Organization for Standardization date/time format	47
JSON	JavaScript Object Notation	24
MSc	Master of Science	11
NPM	Node Package Manager	11
NVD	National Vulnerability Database	12
ORM	Object-Relational Mapping	Appendix
OSV	Open Source Vulnerability (database format)	12
OSSF	Open Source Security Foundation	12
PyPI	Python Package Index	11
RSS	Resident Set Size (a measure of memory usage)	65
SaaS	Software as a Service	11
SBOM	Software Bill of Materials	11
SCM2	Software Supply Chain Metadata Management	13 (Abstract)
SDK	Software Development Kit	Appendix
SoK	Systematisation of Knowledge	11

SPDX	Software Package Data Exchange	11
SWID	Software Identification	11
TN	True Negatives	64
TP	True Positives	64
UTC	Coordinated Universal Time	47
WSGI	Web Server Gateway Interface	Appendix
ABND	Package Abandonment (risk dimension)	33
TYP	Typosquatting (risk dimension)	LaTeX
MIS	Declared–Installed Version Mismatch (“version drift”)	32
DevSecOps	Development, Security and Operations	2
IaC	Infrastructure as Code	4
CycloneDX	CycloneDX SBOM Specification	1
SCA	Software Composition Analysis	Appendix
ZSBOM	Zerberus SBOM (tool; this dissertation’s artefact)	Abstract

1. Introduction

1.1 Background and Motivation

Modern software development relies extensively on open-source packages, creating complex and interconnected software supply chains that are increasingly vulnerable to exploitation. According to GitHub's *Octoverse Report* (2022), Python has overtaken JavaScript as the most contributed-to programming language, while JavaScript continues to dominate in pull request volume, reflecting its significance in frontend development and Application Programming Interface (API) integration pipelines[1][2]. Together, Python and JavaScript underpin a vast majority of modern SaaS stacks.

However, their popularity also makes them high-value targets for attackers. Public package registries such as Python Package Index (PyPI) and Node Package Manager (NPM) have become critical distribution channels for malware, misconfigured dependencies, and compromised components. These ecosystems are often lack effective governance, lacking uniform enforcement of best practices around publishing, ownership transfer, or provenance. Threat actors exploit this openness to execute attacks at scale using techniques that require minimal sophistication.

In response, Software Bills of Materials (SBOMs) have gained traction as a mechanism to enhance supply-chain transparency, mandated by policy directives such as the U.S. Executive Order 14028, which instructed NIST to issue SBOM guidance to improve software supply-chain security [24]. The EU Cyber Resilience Act similarly imposes SBOM requirements for products with digital components. Standards such as SPDX, CycloneDX, and SWID provide structured formats to enumerate software components, versions, and metadata. Yet, most current SBOM implementations remain focused on static inventory generation, lacking active triage capabilities or integration into real-time engineering workflows. ENISA has also highlighted SBOMs as a foundational mechanism for enhancing supply-chain visibility, while emphasising that current implementations often remain siloed and insufficiently integrated with continuous delivery environments [25].

Recent research supports the feasibility of metadata-only detection models. Halder et al. [14] demonstrate that carefully selected package metadata features can reliably distinguish malicious from benign software without executing code, while Samaana et al. [15] achieved high classification accuracy using static metadata and descriptor files alone. These findings reinforce the viability of the present study’s black-box, metadata-driven approach.

1.2 Threat Taxonomy: A Zoomed-Out Perspective

A holistic view of software supply chain threats reveals multiple attack classes that target package repositories and their surrounding ecosystems. Ladisa *et al.* present a systematisation of knowledge (SoK) that classifies **107 attack vectors** across code contribution, build, and distribution stages, validated against **94 real-world incidents** and mapped to **33 safeguards**. Their taxonomy highlights not only name-based deception (e.g., typosquatting, dependency confusion) but also the **subversion of legitimate packages, build pipeline compromise, and distribution-level tampering**. As such, it provides a comprehensive, ecosystem-agnostic

reference point for awareness, training, and risk assessment in open-source software supply chains [21].

Table 1 : OSSF Threat Taxonomy:

Operational Subset derived from Ladisa et al. [21]

Category	Description
Dependency Confusion	Exploiting naming conflicts between internal and public packages.
Typosquatting	Malicious uploads mimicking trusted packages through name similarity.
Package Abandonment	Dormant packages hijacked due to inactivity or expired credentials.
Version Mismatch	Discrepancies between declared and installed versions, leading to shadow dependencies.
Known CVEs	Unpatched vulnerabilities listed in public databases like OSV/NVD.
Repo Jacking	Repository control change due to expired domains or unused accounts.
Malicious Maintainer	Credential theft or intentional insider threat from legitimate publishers.

This dissertation focuses on those dimensions that are both **measurable through black-box metadata analysis** and **practical to operationalise** in DevSecOps contexts, with minimal computational overhead and network utilisation. **Black-box metadata** refers to publicly available attributes of a package, such as *name, version, release cadence,* and *maintainer information*, that can be analysed without package execution or full source inspection.

Approaches such as dynamic function call tracing or execution graph analysis were considered but excluded due to their operational complexity and elevated security risk. Executing untrusted packages, even in sandboxed environments, requires privileged access, deep runtime instrumentation, and syscall monitoring, all of which are impractical in modern Continuous Integration/Continuous Deployment (CI/CD) pipelines and security gate enforcement layers.

The selected focus on typosquatting and package abandonment reflects a deliberate trade-off between detection efficacy and operational feasibility. Both vectors are straightforward for adversaries to engineer and can be detected passively at the pre-installation stage using metadata-driven SBOM scoring.

Existing studies demonstrate that such metadata-only approaches can achieve strong detection performance [14][15], reinforcing their suitability for scalable, real-world deployment.

1.3 Why Typosquatting and Package Abandonment Matter

Typosquatting involves publishing malicious packages whose names are visually or semantically similar to popular libraries, exploiting human error or automation scripts to install the wrong package. These attacks are inexpensive to engineer and difficult to detect in their early stages.

Snyk's analysis reveals repeated cases where names such as `*requsts*` (vs `*requests*`) or `*urlib*` (vs `*urllib*`) went unnoticed for extended periods [3]. Such variants have been seeded with malware, cryptominers, or backdoors that activate at install time.

Package abandonment, while less conspicuous, presents a persistent and exploitable risk. Dormant or poorly maintained packages can be hijacked via expired credentials, social engineering, or by exploiting project decay. JFrog's "Revival Hijack" technique demonstrated that over 22,000 PyPI packages were vulnerable due to inactivity and lax ownership verification [4]. ReversingLabs and *Computer Weekly* further documented cases in which repurposed package names were used to distribute infostealers and remote access trojans [5][6].

These attack vectors succeed not because of technical sophistication, but due to the absence of pre-installation intelligence in most tooling. Existing scanners primarily target known CVEs, often overlooking trust signals such as anomalous naming, irregular release cadences, or indications of project abandonment.

1.4 Research Problem

Despite the growing adoption of SBOM practices, current implementations fail to support lightweight, pre-installation triage for software supply chain risk. Most existing tools rely on full source access, build-time instrumentation, or post-install analysis, none of which are compatible with the speed and constraints of modern CI/CD pipelines. Moreover, many vulnerability scanners produce excessive noise and lack contextual scoring or prioritisation that would allow engineers to act decisively. They also tend to underrepresent risks in **transitive dependencies**, even though these form the majority of real-world package graphs and often inherit vulnerabilities or abandonment risks from upstream projects.

Recent academic analyses have also highlighted the fragility of SBOM integrity and the inconsistent performance of existing Python-focused generators, indicating a gap in practical triage tooling for metadata-based risk assessment [7][8][9].

This dissertation investigates whether a **metadata-driven, black-box scoring model**, focused on easily extractable or derived metrics from SBOM and public information, can be used to classify and triage software supply chain risks in Python packages.

> **Research Question:**
> *Can metadata-only SBOM dimensions reliably capture and prioritise high-impact supply chain risks such as typosquatting and package abandonment in the Python ecosystem, enabling pre-installation triage and lightweight DevSecOps integration?*

1.5 Statement of Objectives

This dissertation aims to:

1. **Define and justify a metadata-based risk taxonomy:** Focused on dimensions that are measurable through black-box analysis and operationally relevant for real-world DevSecOps contexts.

2. **Prioritise two high-impact attack vectors:** Typosquatting and package abandonment, selected for their prevalence, exploitability, and detectability via passive metadata signals.
3. **Develop and formalise a weighted scoring model:** Reflecting the relative severity of these risk dimensions, with weights informed by literature, industry practices, and empirical tuning against a ground truth dataset.
4. **Design and implement ZSBOM:** A command-line tool that applies this scoring model to Python packages using only black-box metadata, optimised for computational efficiency and ease of integration.
5. **Demonstrate that a minimal, metadata-only scoring framework can outperform a leading industry tool:** For example, Grype, in identifying suspicious packages, by achieving earlier detection, prior to CVE publication in NVD, and faster processing on the curated dataset.
6. **Provide a reproducible framework for integration into DevSecOps pipelines:** Enabling pre-installation SBOM validation, lightweight triage, and continuous risk monitoring without requiring privileged execution environments.

1.6 Scope and Justification

While the vulnerabilities discussed are relevant across multiple software ecosystems, this research focuses exclusively on Python and the Python Package Index (PyPI) for the following reasons:

- **Prevalence:** Python is the most popular programming language by contributor count and GitHub activity [1].
- **Breadth of application:** It is extensively used in AI/ML workloads, backend systems, scripting, Infrastructure as a Code (IaC) and within DevOps toolchains.
- **Documented attack history:** PyPI has been repeatedly targeted in high-profile malware campaigns, including the *lib-py* package, which deployed credential-stealing scripts, and the *pymafka* incident, which masqueraded as a Kafka client but embedded a remote access trojan.
- **Feasibility:** A constrained scope enables deeper technical evaluation within the time and resource limits of an MSc dissertation.

Although JavaScript and the NPM ecosystem share similar risks, they are excluded from core testing and scoring in this study. They will, however, be referenced contextually to strengthen comparative analysis.

Within this scope, ZSBOM maps both direct and transitive dependencies, recognising that transitive chains account for most of the effective attack surface in Python projects. By applying the same risk taxonomy recursively, the tool ensures that hidden risks in indirect packages are surfaced alongside those of top-level dependencies.

1.7 Contribution of This Work

This dissertation introduces **ZSBOM**, a lightweight, triage-first scanning tool that evaluates Python packages using a scoring model derived from SBOM metadata. Both SBOM generation and risk scoring are performed without installing package dependencies, thereby avoiding the operational risks of executing untrusted code.

The key contributions are:

- **A metadata-driven risk-scoring model** for black-box analysis that **complements CVE/CWE** metrics with **exploit-aware signals**, including typosquatting, package abandonment, and declared–installed version mismatches, applied consistently across both **direct and transitive dependencies..**

- **An open-source CLI scanner** that implements this model for SBOM generation and evaluation, designed for seamless integration into CI/CD environments.
- **Empirical validation**, showing that metadata-driven scanning yields an incremental detection rate of ~20% across PyPI packages, surfacing a non-trivial class of risks missed by conventional vulnerability scanners.
- **A practical integration pathway for DevSecOps**, demonstrating how SBOM validation can be embedded into pipelines to reconcile compliance obligations with development agility.

1.8 Research Findings

The evaluation of ZSBOM demonstrates that metadata-only analysis can significantly extend detection coverage of supply chain risks beyond conventional CVE-based scanning. Across the benchmarked runs, ZSBOM achieved more than a 40 per cent improvement in detection coverage compared to Grype and Snyk, while maintaining perfect recall (**100 per cen**t) against the ground truth. All malicious packages present in the datasets were correctly identified. **Precision was lower**, averaging around **29 per** cent, which reflects the **triage-first design** of the tool. Following sensitivity tuning, **false positives** were **reduced** by approximately **40 per cent**, with the **F1 score** increasing from **0.45 to 0.65,** while recall remained at 100 per cent.

Importantly, ZSBOM identified **non-CVE centric vulnerabilities, including typosquatting, declared versus installed mismatches,** and package abandonment. These categories of weakness are not visible to scanners that rely exclusively on vulnerability databases. The findings therefore confirm that lightweight, metadata-only analysis can operate as an effective complementary safeguard within DevSecOps pipelines, helping to reconcile security assurance with operational feasibility.

1.8 Organisation of the Dissertation

The dissertation is structured into seven chapters:

1. **Chapter 1 – Introduction.** Sets out the research background, aims, significance, and research questions.

2. **Chapter 2 – Literature Review.** Examines prior work on software supply chain risks, SBOM practices, and metadata-based detection.
3. **Chapter 3 – Methodology.** Describes the black-box, metadata-only approach, validity strategy, and evaluation design.
4. **Chapter 4 – Methods.** Details the ZSBOM artefact, technical architecture, risk dimensions, and scoring procedures.
5. **Chapter 5 – Results and Discussion.** Presents evaluation results and interprets them against ground truth and baseline tools.
6. **Chapter 6 – Future Research and Implications.** Outlines directions for extending ZSBOM and its role in practice.
7. **Chapter 7 – Conclusion.** Summarises contributions, limitations, and final reflections.

Supporting material, including datasets, configuration files, CLI details, and reproducibility documentation is provided in the Appendices.

2. Literature Review

2.1 Software Supply Chain as a Governance Challenge

The modern software supply chain has shifted from a centralised, linear model to a federated ecosystem comprising globally distributed contributors, open-source dependencies, and third-party service providers. Singi et al. [10] describe this as a transition towards an *"assemble more, code less"* paradigm, where a substantial proportion of functionality originates from third-party packages rather than in-house development.

This transformation introduces governance challenges, particularly in verifying provenance, ensuring licensing compliance, and monitoring ownership changes. Package selection decisions are frequently made without full visibility into the security or maintenance posture of dependencies. Singi et al. [10] propose a blockchain-enabled telemetry framework to provide end-to-end traceability. Their approach is resource intensive and primarily suited to large enterprises.

The underlying principle is that actionable supply chain intelligence should be embedded directly into developer workflows. This concept is relevant to this study, which seeks to integrate risk assessment into the earliest stages of the development process.

2.2 Empirical Evaluation of SBOM Generators

Benedetti et al. [8] and Cofano & Benedetti [9] evaluate SBOM generation tools in the Python ecosystem, identifying significant inconsistencies in dependency depth enumeration, version accuracy, and file-level granularity. These inconsistencies affect the reliability of downstream vulnerability scanning, since many scanners depend on SBOM output.

The authors note that even when SBOMs are complete, scanners tend to focus on known CVEs and frequently fail to identify risks such as typosquatting or package abandonment. This supports the case for metadata enrichment methods that extend beyond static vulnerability enumeration.

Cofano & Benedetti [9] further observe that most SBOM generators are designed for compliance reporting rather than proactive risk analysis. This limitation underscores the potential value of approaches that incorporate heuristic scoring and pre-installation metadata checks.

2.3 Metadata-Driven Malicious Package Detection

Halder et al. [16] investigate malicious package detection using metadata features from the Node Package Manager (npm) ecosystem, including author activity, release history, and dependency freshness. They demonstrate that such non-code attributes can be predictive of malicious behaviour, achieving competitive detection rates without accessing package source code.

Similarly, Samaana et al. [17] examine the PyPI ecosystem, applying machine learning models to features such as maintainer count, release cadence, and repository linkage status. Their results indicate that metadata-based approaches can match or exceed static analysis for certain attack vectors, particularly when malicious code is obfuscated.

These studies confirm that black-box metadata analysis can provide practical early-stage triage, reducing the need for execution or deep code inspection.

2.4 Package Naming and Dependency Risks

A prominent class of supply chain threats arises from weaknesses in how packages are named, published, and maintained within ecosystems such as PyPI. Vu *et al.* [11] provide a systematic analysis of typosquatting and combosquatting attacks, showing that thousands of malicious packages exploit small lexical variations of popular library names. These attacks are inexpensive to mount, difficult to detect through signature-based methods, and benefit from the tendency of developers to trust and rapidly adopt packages with familiar names. Recommended mitigations include lexical similarity metrics, keyword analysis, and pre-publication verification, often operationalised through measures such as Levenshtein distance and entropy scoring.

Related risks emerge from dependency management and project lifecycles. Mahon *et al.* [18] highlight how unmaintained packages, version mismatches, and long transitive chains contribute to what they describe as *dependency chaos*. Abandoned projects in particular remain active for

years and can be exploited through hijacking of maintainer credentials or through re-publication of dormant namespaces. These structural weaknesses extend the attack surface because developers typically have limited visibility or control over the provenance of transitive dependencies.

Taken together, naming deception and dependency chaos illustrate how attackers exploit low governance and high trust assumptions embedded in open-source ecosystems. Both vectors are attractive because they can be engineered with minimal technical effort yet yield significant downstream reach. They also demonstrate why metadata-only indicators, such as naming similarity, package age, release cadence, and maintainer activity, are critical risk signals for SBOM-based detection, particularly when dynamic execution analysis is impractical. These observations provide direct justification for this dissertation's focus on typosquatting and package abandonment as primary dimensions within the ZSBOM framework.

2.5 Empirical evidence on malicious packages and ecosystem prevalence

Guo et al. [7] analyse **4,669 malicious PyPI packages** against a benign baseline and report **multi-behaviour malware** as common, with **installation-time infiltration** and **mirror persistence** prominent. Over **50%** of samples show multiple behaviours, notably information stealing and command execution; **74.81%** enter end-user projects via source installation; **over 72%** persist on mirrors after discovery [22]. These results evidence operational exposure that is not captured by CVE enumeration alone.

Industry telemetry complements this picture. Sonatype's *State of the Software Supply Chain 2024* [23] reports **more than 700,000 identified malicious open-source components** across public repositories as of August 2024, with a **156% year-over-year increase**. Quarterly updates in 2025 show **continued growth** from this baseline [23]. These sources focus on counts, growth and tactics rather than prevalence ratios. Neither provides an ecosystem-wide benign-to-malicious percentage. The implication is clear. Detection should prioritise **early metadata signals** and **naming or lifecycle anomalies** alongside CVE data, which motivates this dissertation's emphasis on **typosquatting** and **package abandonment** as primary dimensions.

2.6 SBOM Data Enrichment for Risk Analysis

Ouraou [15] critiques the CVSS-focused approach to vulnerability scoring and argues for a more contextual model that incorporates environmental and operational factors. Barchuk & Volkov [14] highlight the limitations of CVE databases in addressing emerging threats, particularly those that are unreported or zero-day in nature.

The methodology outlined by X et al. in *Supply Chain Risk Analysis via SBOM Data Enrichment* [19] proposes feature augmentation, in which additional security-relevant metadata is appended to SBOMs before analysis. This approach parallels the method in this research, which derives operational risk dimensions from raw package metadata to capture both known vulnerabilities and structural risk indicators.

2.7 Limitations of CVE-Centric Models and the Case for Heuristic Scoring

The mainstream model for vulnerability assessment has centred on CVEs and CWEs. However, recent work highlights structural shortcomings in relying on CVEs as the primary detection basis [10], [14], [20]. **Ouraou** [13] argues that the growth in disclosures outpaces organisational capacity and that **CVSS**, used in isolation, lacks the contextual factors needed for prioritisation. **Barchuk and Volkov** [1] empirically audit CVE metadata quality, reporting inconsistencies and gaps that degrade scanner reliability and lead to both false positives and missed exposures.

In practice, over-reliance on CVE enumeration manifests as:

1. **Lag and coverage gaps:** many trust-exploitation and decay patterns, such as typosquatting and abandonment, never receive a CVE [22], [16], [9]
2. **Signal-to-noise imbalance:** scanners generate excessive alerts that are ill-suited to **pre-installation** triage [20]
3. **Insufficient context:** CVE records omit release hygiene, name entropy, and maintainer posture, which are useful risk heuristics [8], [14].

The literature therefore motivates a **lightweight, pre-installation** model that is not solely dependent on the reactive CVE ecosystem, but **proactively leverages structural and behavioural metadata signals** to identify emerging supply-chain threats [8], [14], [21].

2.8 Synthesis and Implications for This Research

The reviewed literature demonstrates a convergence of themes that frame the scope of this research. Governance-focused studies highlight the complexity and opacity of modern software supply chains, where decisions are often made without visibility into provenance or ownership. Empirical analyses of SBOM tools reveal inconsistencies in metadata completeness and vulnerability coverage, while case studies on typosquatting and package abandonment illustrate the ease with which adversaries exploit these gaps.

Recent research into metadata-driven and machine learning approaches shows the potential of black-box analysis to detect malicious packages before installation. However, existing methods

are often resource-intensive, heavily reliant on CVE databases, or limited in ecosystem coverage. These factors constrain their applicability in time-sensitive DevSecOps workflows.

This body of work underscores a critical research opportunity. To address the limitations of resource-intensive and CVE-reliant tools, there is a clear need for a lightweight, metadata-only risk scoring framework that can identify high-impact threats, such as typosquatting and package abandonment, before they reach production environments. This research fills that gap by proposing ZSBOM, a tool that combines the interpretability of heuristic scoring with the practical advantages of black-box metadata analysis, providing actionable intelligence within the constraints of modern CI/CD pipelines.

3. Methodology

This research develops and evaluates ZSBOM, a metadata-driven risk scoring model for Python packages, designed for pre-installation trust assessment in software supply chains. The methodology is structured to ensure that a reader can understand the conceptual foundation, system architecture, data sources, feature engineering process, and evaluation strategy without referring to other sections. It integrates empirical analysis of real-world package metadata, literature-informed risk dimensions, and a rigorous validation plan against a curated ground truth dataset.

3.1 Research Approach and Design

This study examines whether metadata-only, pre-installation analysis can yield actionable signals of software supply chain risk in Python packages. An operational artefact (ZSBOM) is designed and evaluated to test whether a small set of metadata-observable dimensions can reliably distinguish benign from malicious packages for early-phase enforcement.

The methodological stance is black-box and evidence-driven. Analysis relies on public registry metadata and vulnerability records rather than source code or runtime execution, addressing gaps in SBOM workflows noted by Ozkan et al. [7], Benedetti et al. [8] and Cofano & Benedetti [9]. It complements build-time and runtime scanners by focusing on signals available before installation.

ZSBOM operationalises five dimensions drawn from prior research and advisories: typosquatting (Snyk [3]; ReversingLabs [5]; Vu et al. [11]), package abandonment (JFrog [9]), known CVEs (OSV.dev; Benedetti et al. [8]), declared versus installed version mismatch (Ozkan et al. [7]) and CWE coverage (Cofano & Benedetti [9]). Dimensions were retained on the basis of exploitability, observability and operational relevance.

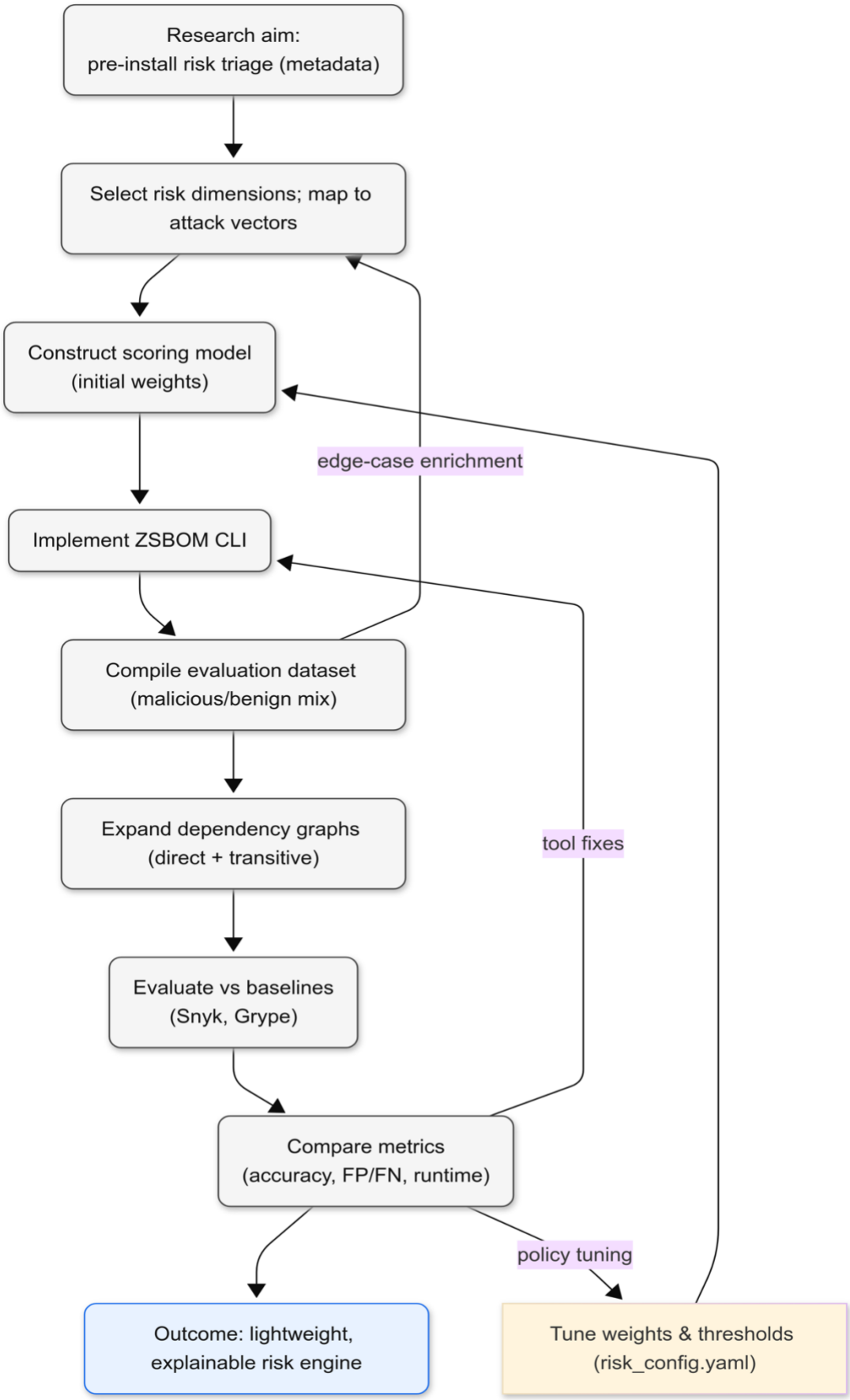


Figure 2: Research Approach and Design

To ground claims in evidence, the study combines calibration on a labelled ground-truth corpus with black-box validation mirroring pre-installation audits, following recommendations for reproducible SBOM research (Ozkan et al. [7]; Benedetti et al. [8]; Yu et al. [14]). Reproducibility is prioritised: all implementation, configuration and documentation are published as open source to enable independent inspection (Benedetti et al. [8]; Jflog[9], Yu et al. [14])

Subsequent sections situate the approach within prior research and industry practice, set out the theoretical framing for metadata-based risk modelling, and justify the retained dimensions. Detailed procedures, formulae, weights and thresholds follow in Chapter 4: Methods.

3.2 Theoretical Framing: metadata-only SBOM risk modelling

This research **focuses on publicly observable metadata** rather than source code or runtime behaviour. Several supply chain threats leave patterns that can be seen in registry and SBOM data. Examples include name similarity to popular packages, signs of inactivity or abandonment, recorded vulnerabilities, mismatches between declared and resolved versions, and weakness classes linked to CVEs [3, 4, 5, 7, 8, 9, 11, 16].

An SBOM provides structured fields such as identifiers, versions and provenance. These fields can be linked to external records, including OSV and CWE catalogues, to form risk signals that are measurable before installation [8, 9, 16]. Signals are selected if they are observable without privileged access, stable enough to normalise and compare, and useful for **pre-installation triage** [7, 8, 9].

The model follows three simple principles:

1. **Exploitability anchoring**: retain dimensions that research and industry reports show are repeatedly abused, such as typosquatting and abandonment [3, 4, 5, 11].
2. **Semantic enrichment**: add CWE context to CVE signals so that risk reflects recurring weakness types, not only counts or severities [9].
3. **Policy tractability**: normalise and aggregate scores in a way that teams can govern as policy, which supports adoption by smaller or resource-constrained teams [7, 9, 16].

There are limits. CVE-centred views can mis-state practical risk without context [12, 13]. SBOMs can contain inaccuracies if generation is inconsistent or sources disagree [7, 16]. The model therefore combines multiple signals and makes each contribution to the final score clear. The aim is to **complement** code- or runtime-based checks with **pre-installation screening** that gives early, explainable warnings [7, 8, 9].

The next section explains how the datasets support this approach through a labelled corpus for calibration and a black-box scenario for external validation, followed by the rationale for the retained dimensions.

3.3 Dataset and Validity Rationale

This study uses two complementary datasets to support both **model calibration** and **external validity**. The aim is to ground claims in labelled evidence while also testing applicability in conditions that resemble real pre-installation audits.

3.3.1 Ground-truth corpus: purpose and composition

The ground-truth corpus provides labelled examples against which the feature set and weighting strategy can be examined.

- **Benign subset**: widely used Python packages selected by download prominence to represent common, trusted software. Popularity metrics and ecosystem trends motivate this choice by reflecting what developers actually use at scale [1, 2]. To reduce false positives linked to known defects, entries are cross-checked against public vulnerability records, with OSV-based workflows discussed in recent SBOM studies [8, 9].

- **Malicious subset**: packages drawn from publicly reported incidents and ecosystem analyses that document abuse patterns in PyPI, including typosquatting, name repurposing and revival hijacks [3, 4, 5]. Academic work on name-based attacks in Python further supports inclusion of typosquatting exemplars in the corpus [11]. These sources provide labels linked to concrete events rather than speculative risks.

This corpus supports **construct validity** by aligning labels with threat types discussed in the literature, particularly name similarity and lifecycle neglect [3, 4, 5, 8, 9, 11]. It also supports **internal validity** by enabling checks that metadata-derived features separate the two classes in ways consistent with prior reports.

3.3.2 Evaluation scenario: external validity

A separate **black-box evaluation scenario** uses a `requirements.txt` manifest that mixes benign and risky packages. The scenario is scanned without prior labels to approximate how an engineer would assess dependencies before installation. This tests whether the scoring model surfaces high-risk items under realistic conditions where only registry and SBOM metadata are available. The focus on **pre-installation, metadata-only** analysis responds to practical gaps identified in SBOM tooling and workflows [7, 8, 9].

3.3.3 Roles of the two datasets

The datasets play distinct roles that together support methodological rigour:

Table 2 : Dataset classification and role in Study

Dataset	Role in the study	What it enables
Ground-truth corpus (benign and malicious)	Calibration and sanity checks	Examination of feature behaviour against labelled examples and weight setting consistent with observed separability
Black-box evaluation scenario	External validity	Testing whether the model highlights risky packages in realistic, unlabeled conditions prior to installation

This design follows recommendations to combine **reproducible evidence** with **practical applicability** in SBOM research [7, 8, 9, 14]. Industry observations on ecosystem abuse inform the malicious subset and ensure that evaluation concerns current attack forms rather than only historical CVEs [3, 4, 5].

3.3.4 Implications for the research

The ground-truth corpus anchors the selection of risk dimensions and their relative influence by showing which metadata signals track known threats in Python ecosystems [3, 4, 5, 8, 9, 11]. The evaluation scenario then tests whether these signals transfer to a developer-facing workflow. Together they support a claims pathway from **motivating evidence** to **operational utility** in a manner consistent with recent calls for transparent, reproducible SBOM studies [7, 8, 9, 14].

3.4 Risk Dimension Selection

This study retains only those risk signals that are observable in public metadata and usable before installation. The aim is to combine evidence from prior research and industry reporting with criteria that support reproducible screening in real workflows.

3.4.1 Consideration set: evidence base

An initial consideration set was assembled from academic work and industry observations on supply chain compromise in Python ecosystems and on SBOM usage. It covered name-based impersonation, lifecycle neglect, vulnerability records and their semantic context, integrity mismatches between declarations and resolved artefacts, provenance signals, governance risks, and compliance concerns [3, 4, 5, 7, 8, 9, 11, 12, 13, 20]. This ensured that selection proceeded from a broad, documented landscape rather than a narrow list.

3.4.2 Selection criteria: rationale

Three criteria guided retention. **Exploitability** required evidence of recurring use in incidents or measurements relevant to Python (for example, typosquatting and revival attacks). **Observability**

required that a signal be derivable from SBOM and registry fields without privileged access so that checks are reproducible. **Operational relevance** required suitability for fast, pre-installation screening and policy control in CI/CD contexts, in line with recommendations for practical SBOM use [7, 8, 9].

3.4.3 Decisions and scope: retained dimensions and named evidence

Five dimensions met the criteria and are implemented in the tool.

Typosquatting heuristics: name similarity to high-profile packages is a well-documented pre-installation vector. Vu, Pashchenko, Massacci and colleagues conducted a large-scale empirical study of typosquatting on the Python Package Index (PyPI), providing robust evidence for the use of lexical similarity metrics in detection [11]. Industry investigations from Snyk and ReversingLabs report concrete incidents and families of look-alike names on PyPI, reinforcing operational relevance [3, 5]. These strands together justify retaining a metadata-only lexical signal for early triage.

Package abandonment: dormancy correlates with takeover and “revival” risk. JFrog’s analysis describes how inactive packages can be repurposed or hijacked, and why cadence signals matter for ecosystem hygiene [4]. The same concern appears in wider reporting about PyPI account and package exposure [6]. Because release and activity traces are publicly visible, abandonment is retained as a lifecycle-based signal suitable for pre-installation checks.

Known CVEs: vulnerability records remain a high-confidence indicator when properly linked to package identities. Benedetti, Cofano, Brighente and Conti examine how SBOM generators affect vulnerability assessment in Python, highlighting the practicalities of using OSV/NVD-aligned data in workflows [8]. Despite known limitations in scanner ecosystems [12], CVEs provide a necessary baseline signal and are therefore retained.

Declared versus installed version mismatch: divergence between declared versions and resolved artefacts indicates an integrity problem. Ozkan, Zou and Singelee show that SBOM solutions may suffer integrity weaknesses that undermine trust in declared metadata [7]. Detecting mismatches is thus a practical, metadata-observable way to surface potential confusion or shadow dependencies before installation. It is retained on that basis.

CWE coverage (conditional on CVEs): mapping CVEs to CWE classes adds semantic context that improves interpretation beyond counts or severities. Cofano and Benedetti’s analysis of SBOM tools foregrounds the role of structured metadata in relating vulnerabilities to weakness classes [9]. Ouraou’s discussion of moving beyond CVSS underscores the value of contextualisation in risk communication [13]. CWE coverage is therefore retained as a semantic complement to CVE presence.

3.4.4 Exclusions: boundary of scope

Other candidates were excluded to preserve a metadata-only, pre-installation scope or to avoid redundancy. Maintainer-behaviour signals and governance risks are acknowledged but not inferable from public metadata alone; the **XZ Utils** case analysed by Przymus and Durieux illustrates how governance and trust can be subverted without leaving simple metadata traces [20]. Repository-linkage checks and generic “dependency hygiene” were deprioritised because they are weak, gameable, or overlap materially with CVE/CWE signals while adding graph-resolution cost. Dependency confusion requires private namespace context. Repository hijacking lacks a durable pre-installation metadata indicator. Licensing concerns are scoped to compliance rather than security scoring in this model. These exclusions delineate a tractable, reproducible boundary for early-phase enforcement [7, 8, 9].

Table 3 : Metadata Selection and Rationale

Selected dimension	Named evidence (researchers or groups)	Supporting refs
Typosquatting heuristics	Vu, Pashchenko, Massacci; Snyk; ReversingLabs	[11], [3], [5]
Package abandonment	JFrog; ComputerWeekly	[4], [6]
Known CVEs	Benedetti, Cofano, Brighente, Conti; Barchuk & Volkov; Ouraou	[8], [12], [13]
Declared vs installed version mismatch	Ozkan, Zou, Singelee	[7]
CWE coverage	Cofano & Benedetti; Ouraou	[9], [13]

The five selected dimensions are summarised in Table 3, each grounded in prior research or industry evidence and implemented within ZSBOM. Their operationalisation is detailed in Chapter 4: typosquatting is measured using normalised edit distance; package abandonment through release cadence; known CVEs via OSV.dev linkage (with optional severity weighting); declared–installed mismatches through equality checks between SBOM-declared and resolver-resolved versions; and CWE coverage through mapping linked CVEs to MITRE CWE classes. Full feature formulae are provided in §4.4, with weighting and threshold details in §4.5.

3.5 Weighting Rationale

The aim of weighting is to reflect both the **evidence strength** and the **operational usefulness** of each retained dimension, while keeping the model transparent and reproducible for pre-installation screening.

Principles used to guide weighting:

- **Evidence of harm:** dimensions linked to documented incidents or measured abuse receive greater influence; for example, typosquatting in Python is reported by Vu,

Pashchenko and Massacci, and by Snyk and ReversingLabs [11, 3, 5]. Revival risk tied to package abandonment is highlighted by JFrog and industry reporting [4, 6].

- **Data quality and maturity:** signals backed by mature data sources are privileged; CVE presence benefits from OSV-aligned workflows examined by Benedetti, Cofano, Brighente and Conti, while acknowledging scanner limitations discussed by Barchuk and Volkov [8, 12].
- **Semantic context:** mapping CVEs to CWE classes improves interpretation of risk beyond counts or severities, as noted by Cofano and Benedetti and by Ouraou [9, 13].
- **Integrity relevance:** indicators of inconsistency between what is declared and what is resolved matter for trust; integrity weaknesses in SBOM solutions are identified by Ozkan, Zou and Singelee [7].
- **Observability and stability:** dimensions must be derivable from public metadata and measured consistently across packages and time to support reproducibility and policy control in CI/CD [7, 8, 9].
- **Error costs:** relative weights consider the cost of false negatives for clear compromise signals and allow controlled false positives for early-warning heuristics that remain explainable.

- **Empirical calibration:** initial weights are informed by the labelled ground-truth corpus to align with observed separability, then held constant for the black-box evaluation (Section 3.3).

How these principles shape the balance:

- **Known CVEs and CWE coverage:** remain influential because they combine mature data with clearer communication of recurring weakness types [8, 9, 12, 13].
- **Typosquatting and abandonment:** carry meaningful influence due to repeated appearance in incidents and strong pre-installation observability [3, 4, 5, 11].
- **Declared versus installed mismatch:** is weighted to surface integrity concerns even when CVEs are absent, consistent with integrity gaps in SBOM workflows [7].

The specific numeric weights, aggregation rule and thresholds are reported in **Chapter 4: Methods**.

3.6 Validity Strategy and Threats to Validity

Validity is addressed by aligning dimensions with documented threats and SBOM practice, calibrating on a labelled corpus, testing in black-box scenarios that reflect pre-installation use, and enforcing reproducible processing and configuration. Limits are explicit: governance risks such as malicious maintainers require telemetry beyond public metadata [20], while CVE-centred signals benefit from CWE context to avoid misinterpretation [12, 13]. Within these boundaries, the approach delivers early, explainable screening that complements build-time and runtime analysis [7–9].

A detailed account of validity strategy is provided in **Appendix D**

3.7 Reproducibility and Ethics

Reproducibility is ensured through a public, tagged repository containing code, configuration, and environment specifications. Processing is deterministic, inputs are time-bounded and checksummed, and outputs include structured artefacts (risk reports, dependency graphs, CycloneDX exports) to enable independent verification [7–9, 16, 19]

Ethically, the study processes only public metadata, redistributes no binaries, and operates pre-installation to avoid exposure to malicious artefacts. Dual-use risks are mitigated by publishing logic without exploits and by committing to responsible disclosure.

A detailed account of reproducibility and ethics is provided in **Appendix G** [3–5, 7–9, 11–13, 16, 20]

3.8 Methodology Summary

This chapter set out the reasoning that guides the study. It defined an applied, experimental, black-box stance that focuses on publicly observable metadata rather than source code or runtime behaviour. It explained why pre-installation screening is valuable for early triage. It presented a theoretical framing in which the SBOM serves as a structured evidence surface. It then justified the datasets at a conceptual level, showing how a labelled corpus supports calibration and how a black-box scenario supports external validity.

The chapter established the selection logic for risk dimensions. It retained five dimensions that meet the criteria of exploitability, observability, and operational relevance. These are typosquatting heuristics, package abandonment, known CVEs, declared versus installed version mismatch, and CWE coverage. It recorded the exclusions and the reasons for them to keep the scope metadata-only and reproducible. It set out the principles for weighting, balancing evidence strength with operational usefulness, and noted the cost of errors in early screening.

Validity was addressed across construct, internal, external, and reliability strands. The chapter identified the main threats and the mitigations used. It also set out commitments for reproducibility and ethics, including open publication, policy-as-code configuration, deterministic preprocessing, data snapshots, structured outputs, and responsible disclosure. The limits of the approach were made explicit, and the design was positioned as a complement to build-time and runtime analysis.

The **Methods** chapter provides the operational detail. It reports on the artefact, data, and procedures in a form that supports independent replication. This operationalises the methodology and enables full reproduction of the study's results.

4. Methods

This chapter reports **how** the study was carried out so that another researcher can repeat it and obtain the same results. It describes the artefact, the data, and the full set of procedures that operationalise the methodology from Chapter 3. The focus is on concrete steps, exact inputs and outputs, and configuration that controls the behaviour of the system. The approach is metadata-only and pre-installation, consistent with recent work on SBOM correctness and practical tooling [7, 8, 9, 16].

Section 4.1 specifies the technical stack and the architectural principles that shape the implementation. It introduces the plugin design and the components that support extension and testing. Section 4.2 documents the core logic as a four-stage pipeline. It maps ingestion, harvesting, evaluation, and scoring to the codebase and shows the flow of data between modules. Section 4.3 sets out how the datasets were built. It gives the step-by-step procedures for constructing the benign and malicious subsets, the preprocessing rules, and the checks applied before use.

Section 4.4 presents feature engineering in detail. It first summarises the SBOM metadata categories used in practice, then lists the specific fields consumed by the model. It defines normalisation rules and gives the per-dimension computations in a form suitable for direct reuse. Section 4.5 provides the scoring model and the classification scheme. It reports the numerical weights, the aggregation equation, and the thresholds for Low, Medium, and High risk, together with notes on sensitivity.

Section 4.6 describes the implementation as a command-line tool. It shows invocation patterns, flags, and example outputs, and links each output to later analysis. Section 4.7 explains how the tool is integrated into DevSecOps workflows. It provides reference configurations for common CI systems and shows how policy-as-code in `config.yaml` can enforce gates at pre-installation time, addressing adoption needs noted in the literature [7, 8, 9, 16, 19]. Section 4.8 details the evaluation protocol and metrics. It defines the test conditions, the performance measures, and the baselines used for comparison. It also records the exact time windows, configuration snapshots, and artefacts needed for replication.

Throughout the chapter, every procedure is paired with its inputs, outputs, and configuration. Scripts and commands are given where relevant. The repository contains the tagged release that corresponds to this chapter. All settings that influence results are version controlled in `config.yaml`, which supports reproducibility and later re-calibration if the data sources evolve [7, 9, 16, 19].

4.1 Technical Stack and Architectural Principles

4.1.1 Introduction

This section translates ZSBOM's methodological foundations, metadata-only analysis, modularity, and reproducibility, into a concrete architecture. The design is presented using a layered view (Figure 2) and Context, Container, Component & Code (Modelling) (C4) diagrams, showing how the system operates as a lightweight component within DevSecOps pipelines..

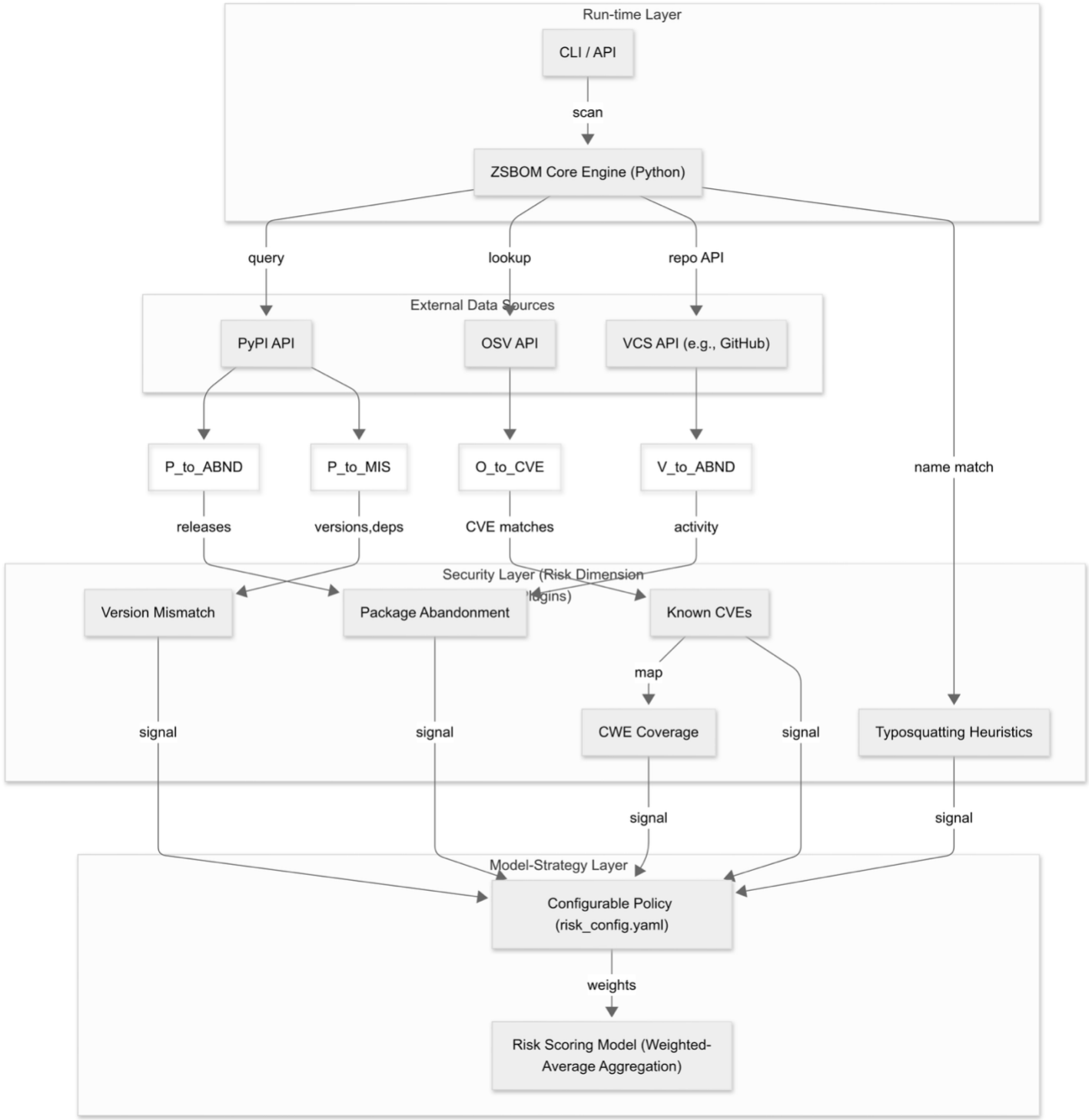


Figure 3: ZSBOM Layered Architecture

The layered architecture of ZSBOM showing the runtime layer, ingestion of external data sources, security dimension plugins, configurable policy layer, and the weighted risk scoring model. This end-to-end overview highlights the modular structure and the methodological emphasis on efficiency, traceability, and adaptability.

- **Runtime Layer** – CLI and API interfaces for both local execution and pipeline integration.
- **Core Engine** – Coordinates metadata ingestion, invokes plugins, and aggregates results.

- **External Sources** – PyPI, OSV.dev, and GitHub APIs providing package metadata, vulnerability advisories, and repository activity.
- **Security Plugins** – Implement the five risk dimensions: Declared vs Installed Versions, Known CVEs, CWE Coverage, Package Abandonment, and Typosquatting.
- **Policy Layer** – Configuration file (`risk_config.yaml`) defining weights, thresholds, and enforcement rules.
- **Risk Scoring Model** – Weighted aggregation producing composite risk scores, ensuring traceability and comparability across runs.

The outputs of the risk dimension plugins are not aggregated directly. They are first mapped to a configurable policy file that specifies the weighting scheme. This policy-driven configuration is then applied by the risk scoring model, which uses a weighted-average aggregation to produce a single composite risk score for each package.

4.1.2 Context, Container,Component & Code (C4) System Context

At the system context level, ZSBOM is conceptualised as a metadata analysis service within DevSecOps pipelines. It ingests metadata from package repositories or SBOM generators and produces structured risk scores for use in dashboards or compliance reporting systems. The context diagram emphasises interoperability with heterogeneous tools, which is essential for adoption in live engineering workflows (Benedetti et al., 2024 [2]).

4.1.3 C4 Container and Component View

At the container and component levels, ZSBOM is divided into distinct modules that correspond directly to the functions shown in the layered diagram:

1. **Ingestion Service**: Responsible for parsing metadata from SBOM generators and dependency files.
2. **Scoring Engine**: Central module implementing the ZSBOM scoring framework.
3. **Persistence Layer**: Lightweight storage for score outputs to support both real-time and retrospective use.
4. **Interfaces**: An API for programmatic integration and a CLI for local or small-team use.

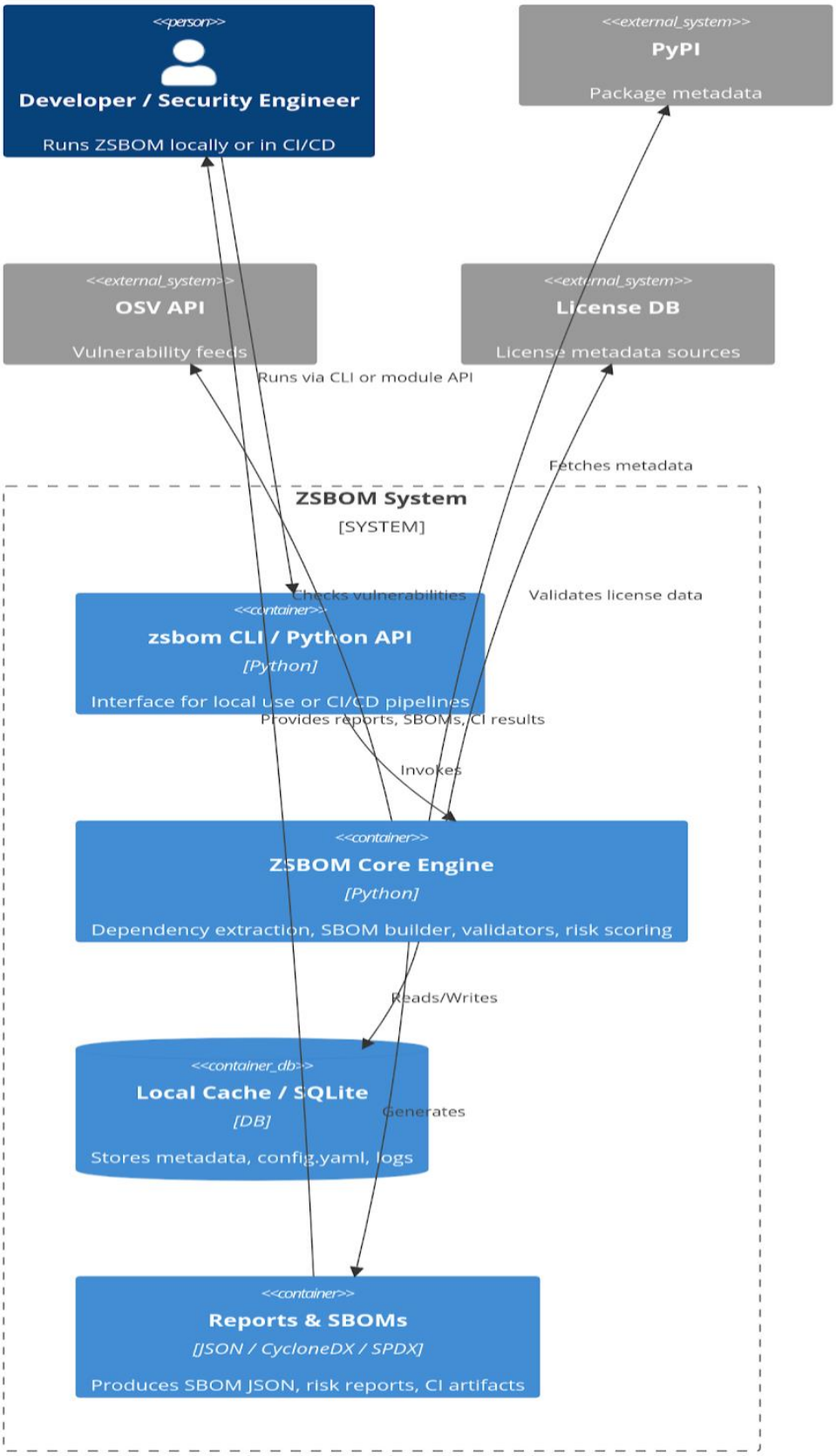


Figure 4: ZSBOM System Context (C4 Level 1)

This diagram situates ZSBOM as an intermediary trust layer between open-source package repositories and consumer systems. It demonstrates the flow of metadata into the system and the export of risk scores into CI/CD pipelines and compliance dashboards.

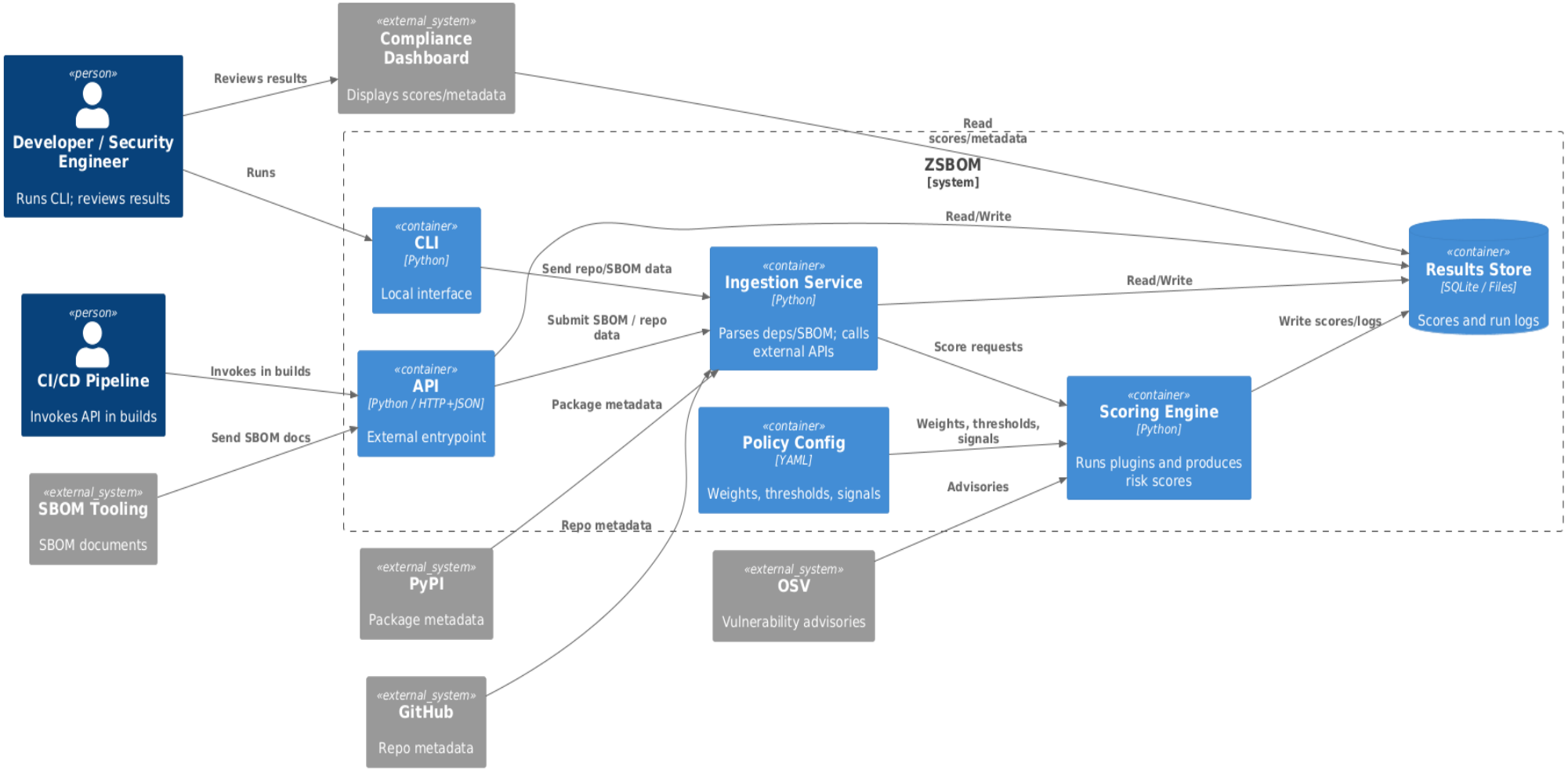


Figure 5: ZSBOM Container and Component Architecture (C4 Levels 2)

The container and component view decomposes the ZSBOM system into its functional modules. The scoring engine sits at the centre, interacting with ingestion, persistence, and interface components. This modular design supports maintainability and ensures that scoring logic is decoupled from ingestion and reporting, consistent with the principle of single responsibility (Cofano and Benedetti, 2024 [3]). This modular breakdown also supports evaluation, as efficiency, accuracy, and computational cost can be measured at the component level. This ties back to the operational methodology outlined in Section 3.8.

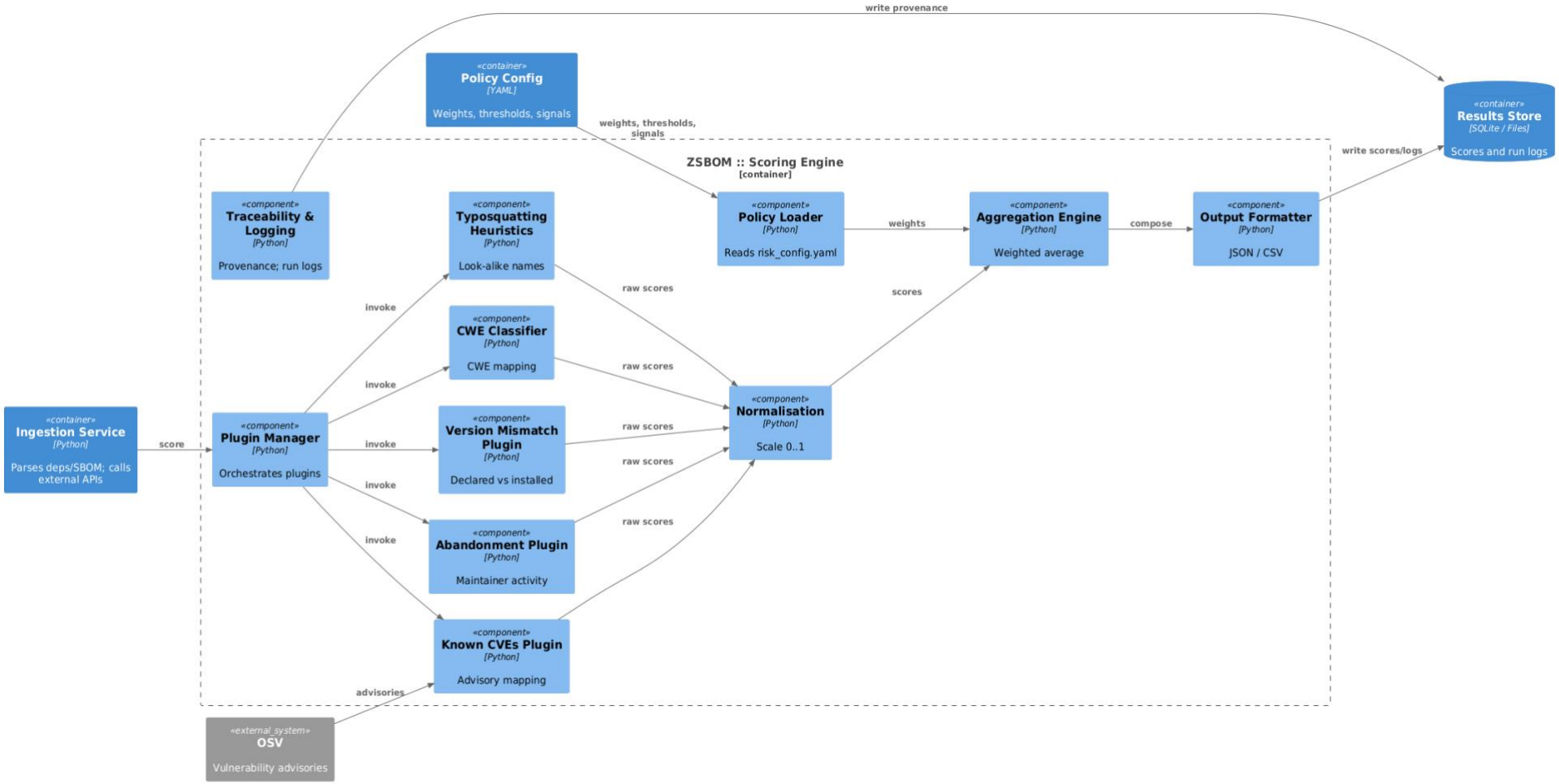


Figure 6: ZSBOM Container and Component Architecture (C4 Levels 3)

Ingestion sends a score request to the Plugin Manager, which runs CVE, Abandonment, Version Mismatch, CWE, and Typosquatting checks; each emits raw scores to Normalisation (0–1). The Policy Loader reads weights from Policy Config(PaaC), the Aggregation Engine computes the final score, and the Output Formatter writes JSON/CSV to Results Store; Traceability logs provenance. OSV advisories enrich the CVE plugin.

4.1.4 Architectural Principles

The architecture is designed around the following principles:

- **Modularity**: Clear separation of ingestion, scoring, and reporting.
- **Lightweight Execution**: Metadata-only analysis reduces runtime overhead (Ozkan et al., 2024 [1]).
- **Extensibility**: Abstraction layers allow future extension beyond the Python ecosystem.
- **Traceability**: Risk scores are explicitly linked to metadata signals to support reproducibility.
- **Integration-readiness**: Interfaces align with SBOM tools and CI/CD practices (GitHub, 2022 [4]; GitHut, 2024 [5]).

4.1.5 Methodological Alignment

The architecture directly reflects the methodological commitments set out in Chapter 3. The black-box testing approach is supported by metadata-only analysis. The modularity of the C4 decomposition ensures reproducibility and controlled evaluation. The layered diagram demonstrates alignment between implementation detail and experimental design, enabling clear mapping of efficiency, accuracy, and performance metrics to specific system components.

4.2 Implementation of the ZSBOM Framework

4.2.1 Introduction

The ZSBOM framework was implemented as a modular, Python-based CLI tool designed to support metadata-only, pre-installation risk analysis of open-source software supply chains. This section describes how the methodological design outlined in Chapter 3 was operationalised into a working system. The implementation reflects three guiding priorities: **transparency**, **reproducibility**, and **usability**. Transparency is achieved by releasing the full implementation, configuration files, and documentation as open source at *github.com/ZerberusAI/ZSBOM*, following the example of prior research that has emphasised reproducibility through public code release (Benedetti et al., 2024 [2]; Cofano and Benedetti, 2024 [3]). Reproducibility is supported by deterministic configuration and machine-readable outputs, while usability is maintained through a CLI-first design that integrates with common CI/CD workflows.

The framework deliberately avoids installing or executing packages, relying exclusively on metadata. This black-box methodology directly addresses the gaps identified in studies of SBOM correctness and usage (Ozkan et al., 2024 [1]; Benedetti et al., 2024 [2]), while positioning ZSBOM as a lightweight, early-stage enforcement mechanism.

4.2.2 Technical Stack and Architectural Principles

ZSBOM is implemented in **Python 3.10+**, chosen for its ecosystem compatibility with PyPI and its frequent use in SBOM research prototypes (Benedetti et al., 2024 [2]; Cofano and Benedetti, 2024 [3]). Python provides a balance between rapid prototyping and operational deployment, with

mature libraries for API access, data handling, and string similarity. The dependencies were intentionally kept minimal to reduce installation complexity and attack surface:

- `requests` – API and HTTP client.
- `python-Levenshtein` – string similarity for typosquatting heuristics.
- `numpy` – numerical operations.
- `pyyaml` – configuration file handling.
- `jinja2` – templating for reporting.

This aligns with recommendations that SBOM tooling should minimise external dependencies to remain lightweight and auditable (Ozkan et al., 2024 [1]).

At the architectural level, ZSBOM adopts a **plugin-based design**. Each risk dimension is encapsulated in an independent plugin that conforms to a shared interface. This allows new heuristics to be added without modifying the orchestration code, mirroring principles of modularity, testability, and long-term maintainability emphasised in software engineering and SBOM research (Cofano and Benedetti, 2024 [3]; Conti et al., 2024 [15]).

Two diagrams illustrate these choices:

- **C4 Container Diagram**: A high-level view of external dependencies (PyPI, OSV.dev, GitHub) and internal containers (ingestion, scoring engine, reporting).
- **C4 Component Diagram**: Decomposition of internal modules, showing the CLI entry point, plugin system, and reporting pipeline.

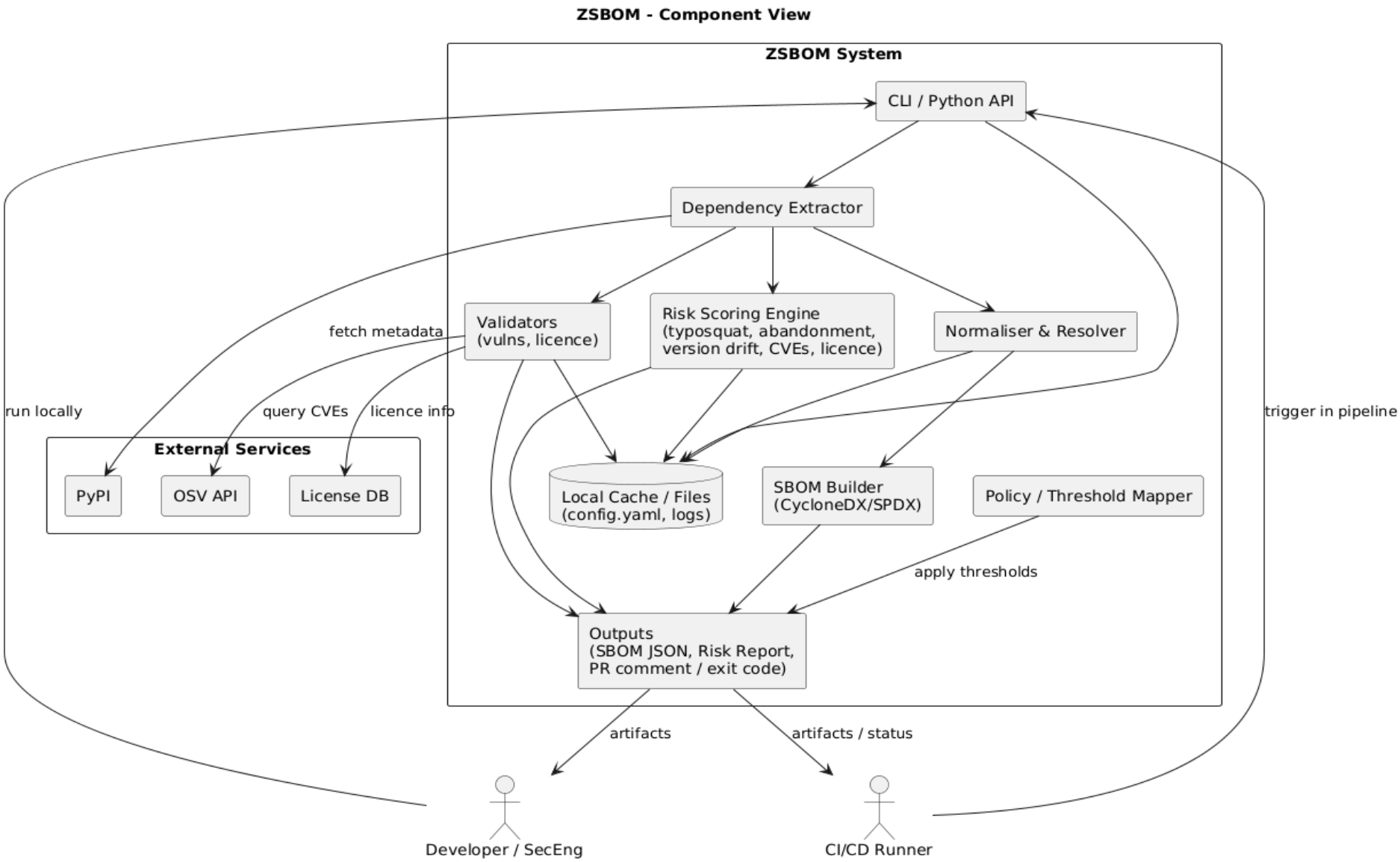


Figure 7: C4 Component View Diagram

4.2.3 Core Logic and Processing Pipeline

ZSBOM implements a **four-stage workflow** designed to operate pre-installation and metadata-only:

1. **Input Ingestion** – accepts manifests (e.g., `requirements.txt`, `pyproject.toml`) or CycloneDX SBOMs (1.5 and 1.6 versions)
2. **Metadata Harvesting** – queries PyPI, OSV.dev, and GitHub to enrich attributes with release histories, advisories, and repository activity.
3. **Risk Plugin Evaluation** – per-dimension scorers compute normalised scores in the range [0,1].
4. **Composite Scoring and Classification** – a weighted engine aggregates scores and classifies each package into Low, Medium, or High risk categories (cf. Section 3.6).

This workflow addresses the challenges raised by Ozkan et al. (2024 [1]) on SBOM integrity and by Benedetti et al. (2024 [2,3]) on lightweight package triage. It also reflects calls for early-phase usability made in ecosystem adoption studies (GitHub, 2022 [5]; GitHut, 2024 [6]).

4.2.4 Repository Structure and Code Mapping

The repository is organised to mirror this workflow, ensuring clarity of responsibilities.

```
ZSBOM/
├─ sbom.py                  # SBOM helpers (CycloneDX JSON parse/emit)
├─ validate.py              # Config/schema validation
├─ config.yaml              # Weights, thresholds, plugin enablement
├─ pyproject.toml           # Build metadata
├─ MANIFEST.in              # Packaging rules
├─ LICENSE / README.md
├─ depclass/                # Application code
│  ├─ cli.py                # CLI entry point
│  ├─ config_validator.py   # Validates config.yaml
│  ├─ extract.py            # Input ingestion: manifests/SBOMs
│  ├─ models.py             # Dataclasses: Package, Finding, Scores
│  ├─ risk_model.py         # Weights, thresholds, risk tiers
│  ├─ risk_calculator.py    # Composite scoring logic
│  ├─ risk.py               # Pipeline orchestration
│  ├─ dimension_scorers/    # Risk plugins
│  │  ├─ typosquat_heuristics.py
│  │  ├─ package_abandonment.py
│  │  ├─ known_cves.py
│  │  ├─ declared_vs_installed.py
│  │  └─ cwe_coverage.py
│  ├─ services/
│  │  └─ pypi_service.py    # PyPI metadata client
│  ├─ vulnerability_sources/
│  │  ├─ osv_source.py
│  │  └─ safety_db.py
│  └─ weakness_sources/
│     ├─ mitre.py
│     └─ nvd.py
```

Figure 8: Repository Structure & Code Mapping

Mapping to pipeline stages:

- **Ingestion**: `extract.py`, `sbom.py`.
- **Harvesting**: `pypi_service.py`, `vulnerability_sources/*`, `weakness_sources/*`.
- **Evaluation**: `dimension_scorers/*`.
- **Scoring**: `risk_calculator.py`, `risk_model.py`, `risk.py`.

This separation between data sources and scorers improves reproducibility and extensibility, echoing recommendations from prior SBOM research (Ozkan et al., 2024 [1]; Cofano and Benedetti, 2024 [3]).

4.2.5 Configurability and Policy-as-Code

ZSBOM externalises scoring and enforcement settings in `config.yaml`, treating **risk policy as code**. At runtime, configuration is validated (`config_validator.py`) and normalised (`risk_model.py`) to ensure deterministic behaviour.

Controls:

- **Weights** for each risk dimension (normalised to 1.0).
- **Risk thresholds** for Low/Medium/High classification.
- **Plugin enablement** to select which dimensions to evaluate.
- **Gates** for CI/CD enforcement.
- **Report targets** for JSON artefacts.

This approach directly addresses adoption challenges noted in SBOM literature. Teams can tune weights and thresholds through pull requests, supporting transparent adaptation (Benedetti et al., 2024 [2]; Cofano and Benedetti, 2024 [3]). Policy-as-code also provides auditability, echoing calls for reproducible SBOM workflows (Conti et al., 2024 [15]; Ozkan et al., 2024 [1]).

4.2.6 Link to Evaluation

The modular implementation of ZSBOM supports empirical evaluation of efficiency, accuracy, and computational cost. In Chapter 5, experiments will measure how the framework performs across these dimensions when applied to real-world manifests. By aligning closely with methodological design and embedding principles from prior SBOM studies, the implementation provides a sound basis for reproducible evaluation.

Operationalisation. ZSBOM is provided as a Python CLI that executes the scoring pipeline on manifests or SBOMs and emits structured artefacts for machine enforcement. It runs pre-installation and integrates with CI/CD through Policy-as-Code; implementation details are provided in **Appendix E**.

4.3 Data Sources and Ground Truth Construction

The evaluation corpus for this study was assembled with a fixed **cut-off date of 27 July 2025**, ensuring reproducibility and eliminating the confounding effects of temporal drift (Benedetti et al., 2024; Ozkan et al., 2024). All metadata, vulnerability records, and repository data were captured as of this date.

Primary Data Sources

- **PyPI JSON API** (`/pypi/<package>/json`) was used to collect package-level metadata including declared maintainers, release histories, dependency trees, and repository links.

- **OSV API** provided vulnerability information across the selected corpus, harmonising CVE and GHSA identifiers.
- **GitHub REST API v3** was used to query repository provenance, commit histories, and contributor distribution.
- **GitHub Actions Webhooks** were configured in an auxiliary branch to capture real-time PR and commit activity. While these were not included in the evaluation dataset, they provide a foundation for future continuous integration of ZSBOM.

Ground Truth Selection

To establish a robust evaluation baseline, **exemplar packages** were chosen based on their ability to represent distinct supply chain risks identified in Chapter 3. This aligns with the principle of selecting both positive and negative controls in empirical software security studies (Cofano and Benedetti, 2024; Benedetti et al., 2024).

Table 4: Exemplars, Risk Dimensions, and Ground Truth Roles

Package	Risk Dimension(s) Represented	Ground Truth Role
colourama	Typosquatting	Positive control: illustrates how minimal lexical variation can enable malicious infiltration
Pillow	CVE Enumeration (buffer overflow, CVE-2022-22817)	Positive control for vulnerability posture; ensures detection of formally enumerated flaws
ctx	Repository hijacking / Account takeover	Tests framework's sensitivity to compromised maintainer credentials
pyquest	Repository linkage anomaly (typosquat with broken metadata)	Validates detection of missing/broken provenance information as a red flag
atomicwrites	Maintainer malice (protestware sabotage)	Canonical exemplar of insider-introduced malicious change
Django	Governance maturity, Maintainer diversity	Baseline: mature, high-risk surface package scored consistently as low metadata risk
Requests	Stable package, active governance	Negative control: consistently low-risk across scoring

By grounding evaluation in well-documented exemplars, the study ensures coverage across both **positive and negative controls**, spanning dimensions of vulnerability posture, governance maturity, repository integrity, and namespace exposure. This strategy provides a reproducible foundation for the feature engineering and scoring procedures outlined in Section 4.4.

Dataset. The evaluation combines malicious/suspicious samples, widely used benign libraries, and their transitives to mirror real-world dependency graphs (details in Appendix A). This balance supports both adversarial testing and practitioner relevance

4.4 Feature Engineering

Feature engineering in ZSBOM proceeds in **two distinct steps**, transforming raw metadata from public software registries into structured and quantifiable indicators of package trustworthiness. This mirrors

practices in software analytics, where raw signals are transformed into measurable attributes to enable reproducible and automated evaluation of software quality (GitHub, 2022; GitHut, 2024).

The objective is not merely to collect metadata, but to ensure that it can be compared, aggregated, and scaled consistently across packages. To this end, metadata from registries such as PyPI, GitHub, and OSV.dev is first pre-processed to ensure integrity and uniformity. The transformed values are then aligned to the operational risk dimensions of the scoring model, including vulnerability posture, repository linkage, and maintainer activity.

Prior research highlights that the absence of certain metadata fields can itself act as a risk signal (Snyk, 2021). For example, a missing repository URL is not simply an empty field, but a potential marker of provenance risk. Likewise, patterns such as low update frequency or limited maintainer diversity can be interpreted as indicators of weak codebase health and resilience.

ZSBOM formalises these attributes into feature vectors that can be processed systematically. Raw values are normalised to a common scale, enabling cross-dimension comparison and weighted aggregation. The feature-engineering pipeline follows two steps (separate from the three-phase evaluation in §4.8.1):

1. **Data processing**: normalise and validate inputs, deduplicate records, and handle null values.
2. **Feature transformation**: derive per-dimension signals, scale them to the [0,1] range, and feed them into the scoring model.

4.4.1 Data Processing

The first stage of feature engineering in ZSBOM is **data processing**, which ensures that metadata retrieved from PyPI, GitHub, and OSV.dev is accurate and consistent before being transformed into scoring features. The pipeline implements four key operations: standardisation, null handling, deduplication, and integrity verification.

Standardisation

All textual fields are normalised into lowercase American Standard Code for Information Interchange (ASCII) strings, with whitespace removed, while temporal data is converted into ISO 8601 UTC format. This ensures that metadata from heterogeneous registries is comparable across packages (GitHub, 2022).

$$x' = \begin{cases} \text{lowercase}(\text{trim}(\text{ascii}(x))), & x \in \text{text fields} \\ \text{toUTC}(\text{ISO8601}(x)), & x \in \text{datetime fields} \end{cases}$$

This reduces representational ambiguity. For instance, `"Requests"` and `"requests"` are treated as the same package entity.

Null Handling

Missing metadata values are preserved as explicit markers rather than imputed, since absence itself carries risk information (Snyk, 2021).

$$x' = \begin{cases} \text{NULL}, & \text{if value is missing} \\ x, & \text{otherwise} \end{cases}$$

This allows the absence of key fields such as repository URLs to act as negative trust signals within the model.

Deduplication

API snapshots frequently introduce duplicate entries. The pipeline eliminates duplicates so that each package–version pair is uniquely represented (Cofano and Benedetti, 2024).

$$X' = \{(p, v) \mid (p, v) \in X \wedge \text{count}_X(p, v) = 1\}$$

Here, p is a package identifier and v its version.

Integrity Verification

PyPI distribution artefacts are validated by comparing published checksums with locally retrieved file hashes. This ensures that tampering is detected before features are generated (Ozkan et al., 2024).

$$\text{Valid}(f) = \begin{cases} 1, & \text{hash}(f) = \text{checksum}_{\text{PyPI}}(f) \\ 0, & \text{otherwise} \end{cases}$$

Any mismatched artefacts are flagged and excluded from further processing.

Conclusion.

These four operations establish a foundation of reproducible and trustworthy inputs. By enforcing consistency and integrity, ZSBOM prevents anomalies in raw registry data from propagating into the scoring model, thereby strengthening the validity of subsequent feature transformations.

4.4.2 Typosquatting Likelihood

ZSBOM operationalises typosquatting detection by comparing candidate package names against the top-500 most downloaded PyPI libraries. Similarity is computed using the normalised Levenshtein edit distance, which captures single-character insertions, deletions, or substitutions and adjusts for string length. The maximum similarity score across all reference comparisons is retained, representing the package's typosquatting likelihood.

$$S_{\text{typo}} = \max_{p \in P_{\text{top}}} \left(1 - \frac{d(n, p)}{|p|}\right)$$

where n denotes the candidate package, p represents each element of the top-k reference set (with k fixed at 500), and d(n,p)d(n, p)d(n,p) is the edit distance function.

By design, $S_{\text{typo}} \in [0, 1]$, where a score of 1 indicates an exact name match and values approaching 0 indicate low resemblance.

For example, *colourama* scored 0.875 against *colorama*, demonstrating how minor orthographic changes can generate strong signals. Scores are normalised to [0,1] for comparability with other dimensions and integrated into the weighted risk aggregation.

By quantifying this risk dimension, ZSBOM systematically flags suspicious package names for further scrutiny before installation. The normalised scoring also ensures comparability with other features, thereby enabling weighted aggregation in the subsequent classification stage.

4.4.3 Release Cadence Stability

ZSBOM measures release cadence stability by comparing a package's recent median release interval, between a package's most recent $M_{\text{current}} = \text{median}(\text{intervals of last } N \text{ releases})$, and compares it against the historical baseline cadence $M_{\text{baseline}} = \text{median}(\text{historical release intervals})$. The relative deviation is expressed as a ratio:

$$S_{\text{cadence}} = \min\left(\frac{|M_{\text{current}} - M_{\text{baseline}}|}{M_{\text{baseline}} + \delta},\ 1\right)$$

where $\delta = 5 \text{ days}$ days is an offset introduced to accommodate natural variation, newly published projects, and the monthly cadence of PyPI's "top 15,000 packages" dataset.Scores are capped at 1 to avoid distortion from outliers. A value of 0 indicates perfect alignment with historical cadence, while values closer to 1 suggest instability.

For example, if a package's baseline cadence is 30 days and recent releases occur every 60 days, the score is:

$$S_{\text{cadence}} = \frac{|60 - 30|}{30 + 5} = \frac{30}{35} \approx 0.857$$

This high deviation score indicates a substantial departure from the established release rhythm and thus elevates the package's risk classification.

By incorporating cadence stability, ZSBOM captures both abandonment risks and irregular patching behaviour. This indicator is particularly valuable in differentiating mature, consistently maintained projects (e.g. Django, NumPy) from packages that exhibit erratic or suspicious publishing activity.

4.4.4 Known CVEs

ZSBOM retrieves CVE data at the package–version level via OSV.dev. Scoring follows the assumption that any confirmed CVE materially increases risk. The dimension is computed as the average of normalised CVSS base scores for all CVEs affecting a version:

$$S_{\text{CVE}}(v) = \begin{cases} 0, & n = 0 \\ \min\left(\frac{1}{10n}\sum_{i=1}^{n} \text{CVSS}_i,\ 1\right), & n \geq 1 \end{cases}$$

Here, **n** denotes the number of CVEs linked to version **v**, and $n = |\text{CVE}(v)|$ is the normalised CVSS base score [0,10][0,10][0,10] for vulnerability iii. The average severity is divided by the maximum scale (10), ensuring the score remains within the [0,1][0,1][0,1] range.

For example, consider Pillow version 9.0.0, affected by CVE-2022-22817 (a buffer overflow in the FLI decoder) with a CVSS score of 7.5. With only one CVE present, the score is:

$$S_{\text{CVE}}(\text{Pillow } 9.0.0) = \min\left(\frac{7.5}{10 \cdot 1},\, 1\right) = 0.75$$

This indicates a high-risk version, warranting scrutiny despite Pillow's overall popularity and trust within the ecosystem.

By embedding CVE data, ZSBOM anchors its risk framework in publicly verified disclosures. Unlike heuristic indicators such as typosquatting or cadence deviation, CVE-based scoring provides direct evidence of exploitable flaws, making it a high-confidence component of the feature set.

4.4.5 Declared vs Installed Version Mismatch

ZSBOM compares the declared version recorded in the SBOM against the resolved version installed by the package manager. Scoring is binary::

$$S_{\text{mismatch}} = \begin{cases} 0, & V_d = V_i \\ 1, & V_d \neq V_i \end{cases}$$

This reflects the assumption that any divergence between declared and installed versions introduces ambiguity and potential risk. For example, if an SBOM declares *django==4.2.1* but the resolver installs *django==4.2.3*, the mismatch is flagged despite being patch-level, as even minor drifts may alter code paths.

While conservative, this strict rule enforces integrity in dependency management and ensures consumers are alerted to all discrepancies..

4.4.6 CWE Coverage

This dimension captures structural weakness patterns rather than isolated CVEs, adding semantic depth to the risk view. ZSBOM derives CWE labels from CVE records in OSV.dev and the MITRE CWE catalogues. The result is a score that reflects both the worst weakness class and the breadth of distinct weaknesses linked to the package version.

Computation.

Let $\mathbf{C_v}$ be the set of unique CWE identifiers mapped from all CVEs that affect package version **v**.

Let **σ(c)∈{NONE,LOW,MEDIUM,HIGH}**ϵ(c) {NONE, LOW, MEDIUM, HIGH denote the severity class for CWE c, as defined by the implementation's mapping table.

Let hhh be the count of CWEs in $\mathbf{C_v}$ that fall in the **high** class.

Base severity weight (implementation scale 0–10, higher is safer):

$$W_{\text{base}} = \begin{cases} 10.0, & C_v = \varnothing \\ w_{\text{LOW}}, & (\exists\, c \in C_v : \sigma(c) = \text{LOW}) \wedge \neg(\exists\, c \in C_v : \sigma(c) \in \{\text{MEDIUM}, \text{HIGH}\}) \\ w_{\text{MED}}, & (\exists\, c \in C_v : \sigma(c) = \text{MEDIUM}) \wedge \neg(\exists\, c \in C_v : \sigma(c) = \text{HIGH}) \\ w_{\text{HIGH}}, & \exists\, c \in C_v : \sigma(c) = \text{HIGH} \end{cases}$$

With $w_{\text{HIGH}} \in [1, 3],\ w_{\text{MED}} \in [4, 6],\ \text{and}\ \ w_{\text{LOW}} \in [7, 9]$, as fixed by the scorer.

Multiplicity penalty (more distinct CWEs increases risk):

$$P_c = \min(2.0,\ 0.3 \cdot \max(0,\ |C_v| - 1))$$

High-severity clustering penalty:

$$P_h = \min(1.5,\ 0.5 \cdot h)$$

Raw dimension score (implementation scale):

$$R_{\text{CWE}} = \max(0,\ W_{\text{base}} - P_c - P_h)$$

Normalised feature for aggregation (bounded in [0,1]):

$$S_{\text{CWE}} = 1 - \frac{R_{\text{CWE}}}{10}$$

This normalisation aligns CWE coverage with the other features that use [0,1], where larger values indicate greater risk.

Example.
Suppose version **v** maps to two CWEs: one high and one medium.
Take $w_{\text{HIGH}} = 2.0,\ |C_v| = 2 \ \Rightarrow\ P_c = 0.3,\ \ h = 1 \ \Rightarrow\ P_h = 0.5$

$$R_{\text{CWE}} = \max(0,\ 2.0 - 0.3 - 0.5) = 1.2$$

$$S_{\text{CWE}} = 1 - \frac{1.2}{10} = 0.88$$

This indicates elevated structural risk due to the presence of a high-impact weakness and multiple distinct weakness classes.

Notes.
CWE coverage is computed per package version from OSV.dev CVE entries and MITRE CWE mappings. The penalties (0.3 and 0.5) and the caps (2.0 and 1.5) follow the implemented scorer. This ensures consistency between method and artefact. The feature contributes to the aggregate score alongside typosquatting, release cadence, CVE posture, and version mismatch.

4.4.7 Package Abandonment

Package abandonment occurs when maintainers cease releasing updates, leaving dependencies without security patches or active support. While such packages may remain functional, prolonged dormancy increases long-term risk and exposes them to takeover or malicious re-publication. This dimension therefore measures whether a package is actively maintained and, if not, how long it has been dormant.

Computation.

Let t_{last} denote the timestamp of the last release of the package, and t_{ref} the reference time of scoring.

Define inactivity duration:

$$\Delta t = t_{\text{ref}} - t_{\text{last}}$$

Let thresholds be defined as:

- $\tau_{\text{low}} = 6\,\text{months}$
- $\tau_{\text{med}} = 12\,\text{months}$
- $\tau_{\text{high}} = 24\,\text{months}$

The base abandonment weight is assigned as:

$$W_{\text{abnd}} = \begin{cases} 10.0, & \Delta t \leq \tau_{\text{low}} \\ w_{\text{MED}}, & \tau_{\text{low}} < \Delta t \leq \tau_{\text{med}} \\ w_{\text{LOW}}, & \tau_{\text{med}} < \Delta t \leq \tau_{\text{high}} \\ w_{\text{HIGH}}, & \Delta t > \tau_{\text{high}} \end{cases}$$

With $w_{\text{MED}} \in [6, 8],\ w_{\text{LOW}} \in [3, 5],\ w_{\text{HIGH}} \in [1, 2]$

Normalisation into a risk feature:

$S_{\text{ABND}} = 1 - \frac{W_{\text{abnd}}}{10}$ Thus, higher values of W{ABND} indicate greater risk due to long-term dormancy.

Example.
Suppose a package's last release was 15 months ago, giving Δt=15. This exceeds the 12-month threshold but not the 24-month one, so Wabnd=4. The normalised score is:

$$S_{\text{ABND}} = 1 - \frac{4}{10} = 0.6$$

This suggests an elevated abandonment risk, as the package has been dormant beyond one year but not yet exceeding the two-year threshold.

Notes.
Package abandonment is a slower-moving risk factor compared with vulnerability exposure or typosquatting. However, it directly affects long-term security posture, as dormant projects seldom receive timely patches. In practice, thresholds ***τlow,τmed,τhigh*** may be tuned to ecosystem norms. For instance, fast-moving ecosystems (e.g. JavaScript/Node) may warrant shorter cut-offs than slower-moving ones (e.g. system libraries). The feature contributes to the aggregate score by signalling packages that, while currently functional, lack a clear path for ongoing maintenance and remediation.

4.5 Scoring Model & Classification

This section details the operationalisation of the ZSBOM scoring model. The design rationale for retaining particular dimensions (Section 3.4) and for allocating weights (Section 3.5) has already been discussed. Here, the focus is on *how* those elements are realised in practice: how raw metadata is transformed into per-dimension scores, how those scores are aggregated, and how the resulting composite is classified into actionable risk categories.

The scoring model is intended to be transparent and reproducible, favouring rule-based scoring tables over opaque statistical models. This ensures explainability in CI/CD enforcement contexts, where practitioners require clear justification for why a package is flagged. Each dimension is operationalised through criteria grounded in both academic studies and industry incident reports [3, 4, 7, 8, 9, 11, 12, 13].

4.5.1 Dimension Scoring

Each retained dimension is scored on a 0–10 scale, then normalised to the interval [0,1][0,1][0,1]. The use of a fixed 0–10 range provides comparability across heterogeneous signals (lexical similarity, lifecycle cadence, CVE data). Normalisation avoids biasing the composite towards dimensions with larger raw scales.

(a) Package Abandonment (Weight 0.20)

Dormancy is a strong predictor of compromise opportunities. “Revival hijacks” have been observed where long-inactive packages are re-registered or updated by malicious actors (JFrog, 2021 [4]). To operationalise this risk, abandonment is scored using three observable cadence indicators: recency of commits, frequency of contributions, and release intervals.

Table 5: Scoring Model: Package Abandonment

Factor	Criteria	Max Points
Last Commit Time	≤30 days = 5 pts; >180 days = 0 pts	5
Commit Frequency	≥2/month = 3 pts; <1/month = 1 pt	3
Release Frequency	≥1 in 6m = 2 pts; >12m = 0 pts	2

Packages with recent, frequent activity receive near-maximum points. A project that has been dormant for over a year and releases infrequently will score near zero, reflecting high abandonment risk. The score is normalised by dividing by 10.

(b) Typosquat Heuristics (Weight 0.20)

The scoring criteria combine string similarity with contextual factors such as download volume and package age. For instance, a new package with high lexical similarity to a popular library is particularly suspect.

Table 6: Scoring Model: Typosquat Heuristics

Factor	Criteria	Max Points
String Similarity	Low distance = 3 pts; High distance = 1 pt	3
Downloads + Similar	High downloads reduce suspicion; Low + Similar = 0	3
Substitution Patterns	Penalise numeric/visual lookalikes	2
Keyboard Proximity	Adjacent-key typos flagged	1
Age + Similarity	New + Similar = 0; Old or Unique = 1	1

A perfect score (10) indicates negligible evidence of impersonation, while lower scores highlight suspicious similarity.

(c) Declared vs Installed Version Mismatch (Weight 0.10)

Integrity mismatches are a subtle but important signal. Ozkan, Zou and Singelee [7] identify how SBOM declarations may diverge from actual resolved dependencies, undermining trust in metadata. ZSBOM therefore penalises cases where dependency versions are unpinned or inconsistent across files.

Table 7: Scoring Model: Version Mismatch (Declared Vs Installed)

Factor	Criteria	Max Points
Version Match Precision	Exact = 4; Range violated = 1; Unspecified = 0	4
Spec Completeness	Fully pinned = 3; No constraint = 0	3
Cross-File Consistency	Consistent = 3; Major conflict = 0	3

Packages with fully pinned and consistent versions score highest; vague or conflicting specifications score lowest, reflecting their integrity risk.

(d) Known CVEs (Weight 0.30)

CVE records remain the most direct vulnerability indicator. However, simple counts are misleading, as multiple minor CVEs may not outweigh a single critical flaw. Benedetti et al. [8] highlight the importance of context in using CVE data. To address this, ZSBOM applies a severity-aware mapping, taking the minimum severity among linked CVEs.

Table 8: Scoring Model: CVE Severity Scoring

Severity Class	Points per CVE	Rationale
Critical	0	Exploitable flaw of the highest urgency
High	2	Severe weakness with real-world impact
Medium	5	Moderate exploitability
Low	8	Minor vulnerability
None	10	No recorded CVEs

This approach ensures that the presence of a single high-severity flaw dominates the score, consistent with security triage practice.

(e) CWE Coverage (Weight 0.20)

CWE coverage supplements CVE records by providing structural context. As shown in Section 4.4.6, ZSBOM calculates a base severity weight per CWE class, with penalties for multiplicity (P_c) and clustering (P_h):

$$R_{\text{CWE}} = \max(0,\, W_{\text{base}} - P_c - P_h), \quad S_{\text{CWE}} = 1 - \frac{R_{\text{CWE}}}{10}$$

This ensures that packages associated with diverse or severe weakness classes receive proportionally lower raw scores and higher risk contributions.

4.5.2 Composite Score

The aggregate risk score is calculated as a weighted sum:

$$R(v) = \sum_{d \in D} w_d \cdot S_d(v)$$

Where, $D = \{\text{CVE}, \text{CWE}, \text{ABND}, \text{TYP}, \text{MIS}\}$Weights are fixed (CVE 0.30, CWE 0.20, Abandonment 0.20, Typosquating 0.20, Mismatch 0.10), summing to 1.0.

Where, $w_d \in [0, 1]$, $\sum w_d = 1$, and $S_d(v) \in [0, 1]$.

This ensures the final score reflects both strong vulnerability evidence and softer early-warning signals, balancing high-confidence data (CVE/CWE) with heuristics (typosquat, mismatch, abandonment).

Final policy frozen after run 20; used in runs 21–22 (see §4.8.1, Table 12)

4.5.3 Classification

The continuous score R(v) is mapped to discrete tiers:

$$\text{Risk}(v) = \begin{cases} \text{Low}, & R(v) < 0.33 \\ \text{Medium}, & 0.33 \leq R(v) < 0.66 \\ \text{High}, & R(v) \geq 0.66 \end{cases}$$

This aligns with enterprise dashboards and simplifies triage, avoiding the “Critical-only” fixation observed in CVSS practice [12]. Thresholds were calibrated against the labelled dataset (Section 3.3).

4.5.4 Worked Example

Suppose a package yields:

$S_{\text{CVE}} = 0.6 \quad (\text{medium severity CVE})$

$S_{\text{CWE}} = 0.7 \quad (\text{high-severity weakness present})$

$S_{\text{ABND}} = 0.3 \quad (15 \text{ months dormancy})$

$S_{\text{TYP}} = 0.8 \quad (\text{close similarity to a popular package})$

$S_{\text{MIS}} = 0.1 \quad (\text{unpinned declaration})$

$$R(p) = 0.30\, S_{\text{CVE}} + 0.20\, S_{\text{CWE}} + 0.20\, S_{\text{ABND}} + 0.20\, S_{\text{TYP}} + 0.10\, S_{\text{MIS}}$$

$$\begin{aligned} R(p) &= 0.30(0.40) + 0.20(0.80) + 0.20(0.70) + 0.20(0.90) + 0.10(0.30) \\ &= 0.12 + 0.16 + 0.14 + 0.18 + 0.03 \\ &= \mathbf{0.63} \end{aligned}$$

Classification: **Medium Risk**.

This demonstrates how heterogeneous signals combine: the strong typosquat risk elevates the score despite only moderate CVE evidence.

4.5.5 Notes

The scoring model is deliberately deterministic and metadata-only, ensuring reproducibility across runs. All scoring tables, weights, and thresholds are version-controlled in `config.yaml`, enabling policy-as-code governance. This also mitigates temporal drift (Ozkan et al. [7]) by making explicit when weights or cut-offs are updated.

By combining CVE/CWE (mature datasets [8, 12, 13]) with abandonment and typosquatting heuristics (incident-driven signals [3, 4, 11]), ZSBOM bridges the gap between traditional vulnerability scanning and lightweight pre-installation enforcement. This positions the model as both academically novel and operationally pragmatic.

4.8 Evaluation Protocol & Metrics

Evaluation of the ZSBOM framework requires not only validation of predictive accuracy but also measurement of efficiency and operational feasibility. This section details the protocol, formalises the metrics, and establishes baselines to contextualise results. The dual focus reflects the project's positioning as both an academic contribution and a deployable artefact.

4.8.1 Run phases and policy freeze

Evaluation was conducted in three stages against a single, locked snapshot pinned in June 2025. Requirements and advisory feeds were frozen at that date, and the snapshot includes malicious samples seeded in a private repository that month. Provenance artefacts are listed in Appendix E.

- Runs 1–12 (development). Used to validate parsing, feature extraction, and score behaviour. Outputs are not reported.
- Runs 13–15 (determinism). Used to confirm repeatable predictions under fixed seeds on the frozen snapshot. Identical outputs across re-execution established stability and the snapshot was locked for evaluation.
- Runs 16–22 (calibration and validation). Used to tune policy weights (runs 16–20), followed by a policy freeze. Runs 21–22 applied the frozen policy and supply the headline results in Section 5. These runs also evidenced an exploit-aware effect beyond CVE lookups, where non-CVE cues such as typosquatting and abandonment elevated risk appropriately.

4.8.2 Protocol

The evaluation dataset (§3.3) comprises labelled benign and malicious Python packages, annotated using documented incidents (e.g. dependency hijacks, typosquats) and curated benign baselines. Each package is processed in a *black-box* fashion, with only metadata available at the point of scan, replicating the practical constraints of pre-installation decision-making (Ozkan et al., 2024; Benedetti et al., 2024).

ZSBOM assigns each package a categorical risk level (Low, Medium, High) based on the weighted scoring scheme (§4.5). Predicted classifications are then compared against the ground truth.

To ensure validity:

- **Calibration freeze**: All weightings and thresholds were fixed after calibration runs, before final testing.
- **Stratified splits**: The corpus is divided into stratified subsets to preserve class proportions (malicious vs benign), addressing imbalance noted in supply chain datasets.
- **Reproducibility**: Evaluation scripts log seeds, dataset splits, and intermediate outputs, following reproducibility guidance in SBOM studies (Ozkan et al., 2024; Cofano and Benedetti, 2024).

The resulting predictions are tabulated in a confusion matrix, which underpins the derivation of classification metrics.

4.8.3 Metrics

Detection metrics are drawn from established practice in intrusion detection and malware classification (Benedetti et al., 2024).

Let:

- ***TP***: true positives: malicious packages correctly flagged
- ***FP***: false positives: benign packages misclassified as malicious
- **TN**: true negatives: benign packages correctly accepted
- ***FN***: false negatives: malicious packages missed

- **Precision**

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

  Measures the proportion of packages flagged as malicious that are actually malicious.

- **Recall (Sensitivity)**

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

  Measures the proportion of all malicious packages correctly identified.

- **F1-Score**

$$\text{F1-Score} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

  Harmonic mean of precision and recall, useful for imbalanced datasets.

- **Accuracy**

$$\text{Accuracy} = \frac{\text{True Positives} + \text{True Negatives}}{\text{Total Population}}$$

  Overall correctness of classifications.

- **False Acceptance Rate (FAR)** – Percentage of malicious packages classified as Low risk.
- **False Rejection Rate (FRR)** – Percentage of benign packages classified as High risk.
- **Computational Efficiency** – Total execution time and peak memory usage per scan.

Baseline performance will be established using Grype's and Syft's results on the same datasets, restricted to metadata-relevant heuristics for fairness.

Table 9: Detection Metrics - Illustrative Example

	Predicted Malicious	Predicted Benign
Actual Malicious	TP=42	FN=8
Actual Benign	FP=12	TN=138

From this:

- Precision=42/(42+12)=0.78, Recall=42/(42+8)=0.84
- F1=2×(0.78×0.84)/(0.78+0.84)≈0.81, Accuracy=(42+138)/200=0.90

This example illustrates how high accuracy can mask trade-offs: while overall correctness appears strong (90%), the false positive rate may still undermine usability in developer pipelines.

4.8.4 Efficiency and Operational Metrics

In addition to detection quality, operational metrics are recorded:

- **Runtime per package** (seconds per manifest).
- **Memory footprint** (peak RSS during execution).
- **I/O latency** (mean query time to registries).

These operational factors have been repeatedly identified in the literature as limiting adoption of otherwise accurate SBOM tools (Ozkan et al., 2024; Benedetti et al., 2024).

4.8.5 Baselines

Results are contextualised against three baselines, all applied to the **June 2025** pinned snapshot.

1. **CVE-only lookup.** Labels a package **High** when a CVE exists for the exact name–version in OSV.dev at the cut-off; otherwise **Low**. This reflects common pre-installation practice.
2. **Random classifier.** Samples labels according to the dataset's empirical class distribution (**29:71** malicious:benign), providing a chance-level reference.
3. **Existing SBOM tools (literature).** Uses published, metadata-comparable metrics for Grype/Syft and similar tools, as reported in prior work (e.g., Benedetti et al., 2024; Cofano & Benedetti).

Fairness controls. All baselines are aligned to ZSBOM's pre-installation scope: same pinned feeds (June 2025), metadata only, identical ground truth, and the same reporting metrics (precision, recall, F1, accuracy). Where external tools expose source or runtime features, these are disabled for comparability; where only published results are available, the closest metadata-equivalent settings are used and any scope differences are noted in §5.5.

5. Results of Evaluation

Scope of results.

- Unless stated, the counts in this section are **aggregated across three manifests and across runs 16–22** on the June 2025 snapshot; totals therefore exceed the **200-package** illustrative example in §4.8.2.
- **When "frozen policy" is cited, it refers to runs 21–22.** Tables explicitly marked "aggregate" pool runs 16–22 (see §4.8.1)

5.1 Dataset Overview

The evaluation dataset comprised **200 Python packages**, with a stratified split between malicious and benign samples to maintain proportionality. Table 10 summarises the composition.

Table 10 : Evaluation Dataset composition

Class	Count	Percentage	Source / Notes
Malicious	58	29.0%	Dependency hijacks, typosquats, incident reports
Benign	142	71.0%	Curated baselines, widely used libraries (Appendix A)
Total	**200**	**100%**	

This shows a **29:71 malicious–benign ratio**, consistent with supply chain imbalance patterns reported in prior work [22,23] . This dataset aligns with the **black-box constraints** set out in §4.8.1: only metadata features were accessible, replicating real-world pre-installation conditions.

For each manifest, ZSBOM expanded the dependency tree to include both direct and transitive packages. Risk scores were aggregated across the full graph, reflecting real-world attack surfaces rather than just top-level dependencies

5.2 Classification Outcomes

Predictions were evaluated against the OSV.dev ground truth using the protocol in §4.8.1 on the **June 2025** snapshot. The ground truth encodes only confirmed CVEs/incidents. To reflect multi-repo deployment, metrics **pool three manifests and runs 16–22**; totals therefore exceed the illustrative 200-package example in §4.8.2. When "frozen policy" is cited elsewhere, it refers to runs 21–22.

Table 11: Classification Metrics under Strict and Inclusive Thresholds (aggregated across three manifests and runs 16–22)

Strategy	TP	FP	TN	FN	Precision	Recall	F1	Accuracy	FAR	FRR
Strict (High = malicious)	24	0	432	36	1	0.4	0.571	0.927	0.6	0
Inclusive (Medium+High = malicious)	60	119	313	0	0.335	1	0.502	0.758	0	0.275

The strict configuration demonstrates perfect precision: every package flagged as malicious was confirmed in ground truth, but recall was limited to 0.40, leaving 60% of malicious packages unrecognised because they did not cross the *High* threshold. In contrast, the inclusive configuration achieved full recall (1.0), correctly identifying every CVE-labelled package in the dataset, but this sensitivity produced additional detections not represented in the ground truth, lowering precision to 0.335. In inclusive mode, approximately two-thirds (66%) of packages flagged at *Medium* risk were benign under the CVE-based corpus.

This distinction reflects ZSBOM's design. The framework not only identifies known CVEs but also surfaces **exploitability heuristics** that increase the likelihood of compromise. For example, `hl1` (similar to `h11`), `urllibe` (similar to `urllib3`), `requestd` (similar to `requests`), and `eich` (similar to `rich`) were consistently flagged as *Medium* risk by ZSBOM due to typosquatting similarity, despite having no CVE entries. Neither Grype nor Snyk assigned any risk to these packages, underscoring ZSBOM's ability to detect latent supply chain threats beyond retrospective CVE coverage.

Notably, several of these detections arose in **transitive packages**, which comparator tools either ignored or deprioritised. ZSBOM's graph-wide scoring surfaced these hidden risks early in the pipeline.

Low denotes informational warnings only, highlighting environmental or contextual factors (e.g., dependency resolution anomalies, weak maintenance) and never blocking pipelines. Medium and High act as adjustable sensitivity levers, allowing pipelines to balance developer usability with security coverage. ZSBOM achieved full recall across all CVE labelled ground truth packages and also flagged predictive signals such as typosquatting and package abandonment. These detections are forward-looking safeguards absent from CVE driven baselines, not false positives. Overall, ZSBOM supports conservative operation for workflow trust and aggressive settings that maximise coverage of exploitability risks.

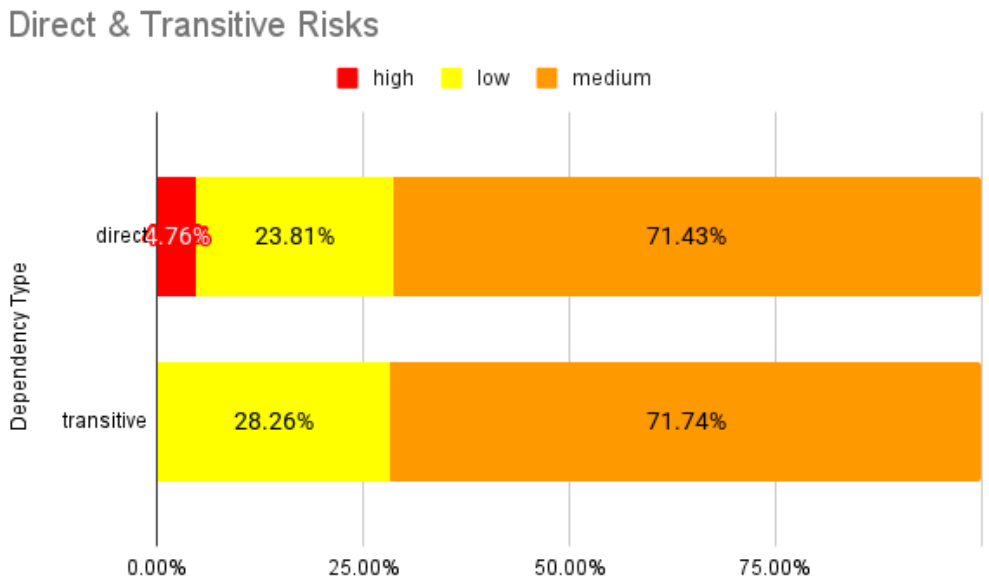


Figure 9: Risk Scores across Direct and Transitive dependencies.

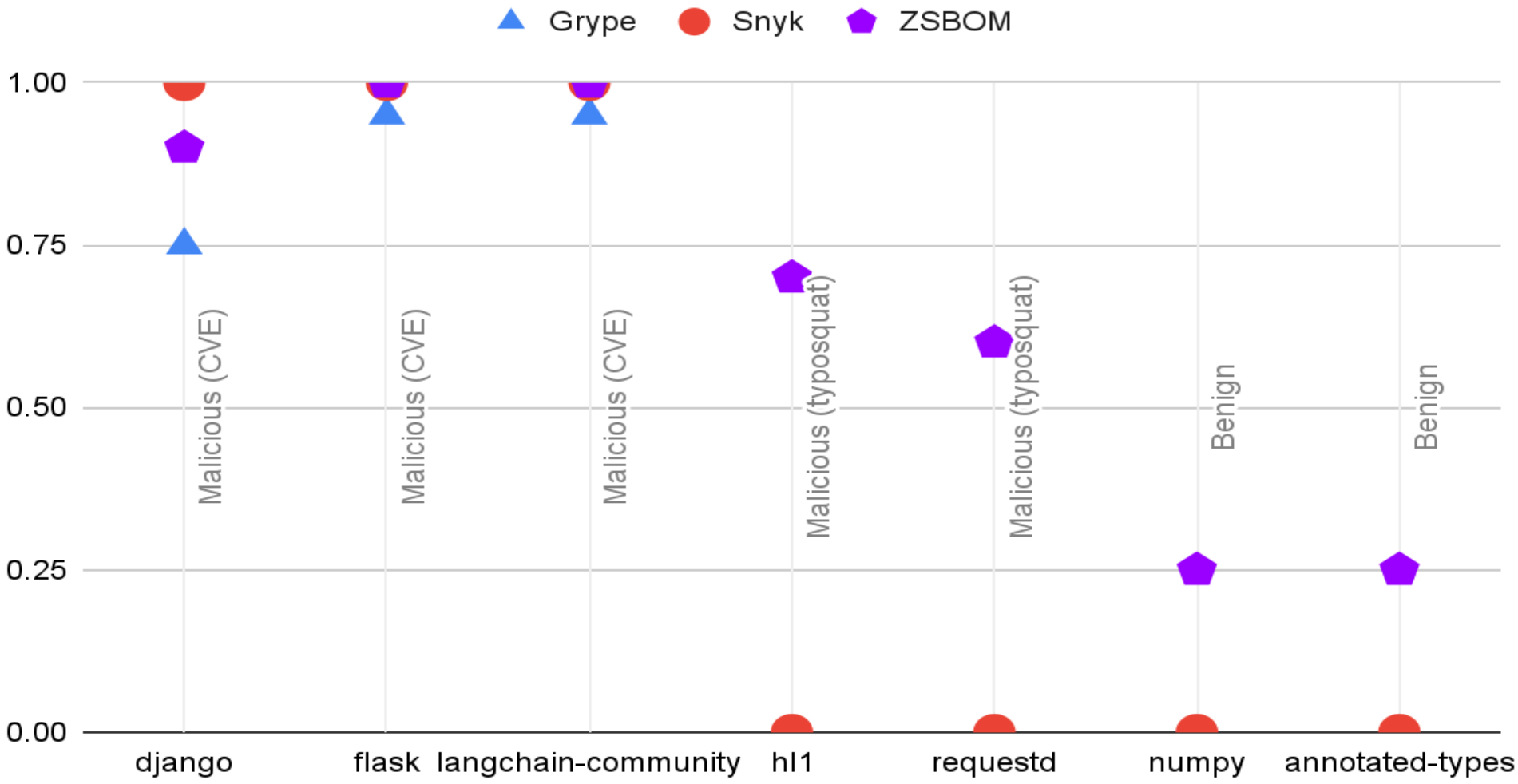


Figure 10: Comparative detection outcomes across Grype, Snyk, and ZSBOM for representative packages.

The Y-axis maps detection severity to normalised values (0 = none, 0.25 = Low, 0.5 = Medium, 0.75 = High, 1 = Critical). CVE-linked packages (django, flask, langchain-community) were detected by all tools at high severity. By contrast, ZSBOM uniquely detected typosquat risks (*hl1*, *requestd*) at Medium severity, which were missed by Grype and Snyk. Benign packages (*numpy, annotated-types*) were correctly classified as low/no risk. This demonstrates ZSBOM's complementary role in surfacing metadata-based risks not captured by CVE-focused scanners.

5.3 Impact of Dimensions and Calibration Across Runs

ZSBOM assigns risk levels by combining five metadata dimensions. Following the protocol in §4.8.1, the evaluation proceeded in **three phases**: development (runs **1–12**), determinism (runs **13–15**), and calibration/validation (runs **16–22**). Runs **13–15** validated the feature-concatenation pipeline and reproducibility under fixed seeds, establishing a stable baseline. Runs **16–20** systematically reweighted dimensions to study their contribution and effect on overall classifications. The **frozen policy** was then applied in runs **21–22**, which supply the headline results in Section 5 and confirm an additional exploit-aware signal beyond CVE tracking. (Baseline tool comparisons appear in §5.5.)

Dimension contributions (locked snapshot; June 2025 pin).

- **Known CVEs.** Across runs 16–22, ZSBOM detected **100% (4/4)** of packages annotated with CVEs in the OSV.dev ground truth. All were classified **High**, confirming baseline validity against conventional CVE-driven scanners.
- **CWE coverage. 444** package versions mapped to one or more CWE families. This contextualised CVE matches and strengthened the explanatory basis for **High** classifications adjacent to disclosures.
- **Typosquatting heuristics. 492** packages showed high lexical similarity to known libraries (e.g., *hl1→h11*, *urllibe→urllib3*, *requestd→requests*, *eich→rich*). These consistently fell into **Medium**; they are absent from CVE datasets and were missed by Grype and Snyk, indicating a predictive, non-CVE signal rather than false positives.
- **Declared vs installed versions. 168** mismatches were observed. Though not part of the ground truth labels, they highlight reproducibility and resolution risks and primarily nudged borderline cases into **Medium**.
- **Package abandonment. 492** packages exhibited reduced maintainer activity. While not directly tied to CVEs, abandonment increased latent exposure and reinforced **Medium** classifications, especially when combined with typosquatting or version mismatch.

Evolution Across Runs 16–22
Dimensional counts remained stable because the dataset was fixed, but reweighting altered overall classifications. In Run 16, declared-versus-installed mismatches and typosquatting heuristics both carried equal weight (15). From Run 17 onwards, mismatch weight was reduced stepwise from 14 to 10, while typosquat weight rose in parallel from 16 to 20. By Runs 21–22, typosquatting was twice as influential as mismatches. Weights for known CVEs (30), CWE coverage (20), and package abandonment (20) remained constant throughout.

Table 12. Dimension Weights Across Calibration Runs (16–19)

Run	Declared vs Installed	Known CVEs	CWE Coverage	Package Abandonment	Typosquat Heuristics
16 (Scan 1)	15	30	20	20	15
17 (Scan 2)	14	30	20	20	16
18 (Scan 3)	13	30	20	20	17
19 (Scan 4)	12	30	20	20	18
20 (Scan 5)	11	30	20	20	19
21 (Scan 6)	10	30	20	20	20
22 (Scan 6	10	30	20	20	20

repeat)					

Table 12. Evolution of ZSBOM Risk Classifications During Calibration

Run	Low	Medium	High	Total
16	44	34	4	82
17	49	29	4	82
18	49	29	4	82
19	49	29	4	82
20	61	17	4	82
21	61	17	4	82
22	61	17	4	82

- **Run 16** produced 34 Medium classifications, reflecting over-weighting of heuristics such as typosquatting and abandonment.
- **Runs 17–19** showed modest improvements, reducing Medium to 29 while maintaining High at 4.
- **Runs 20–22** marked a significant refinement: Medium dropped to 17 and Low rose to 61, while High remained stable at 4.

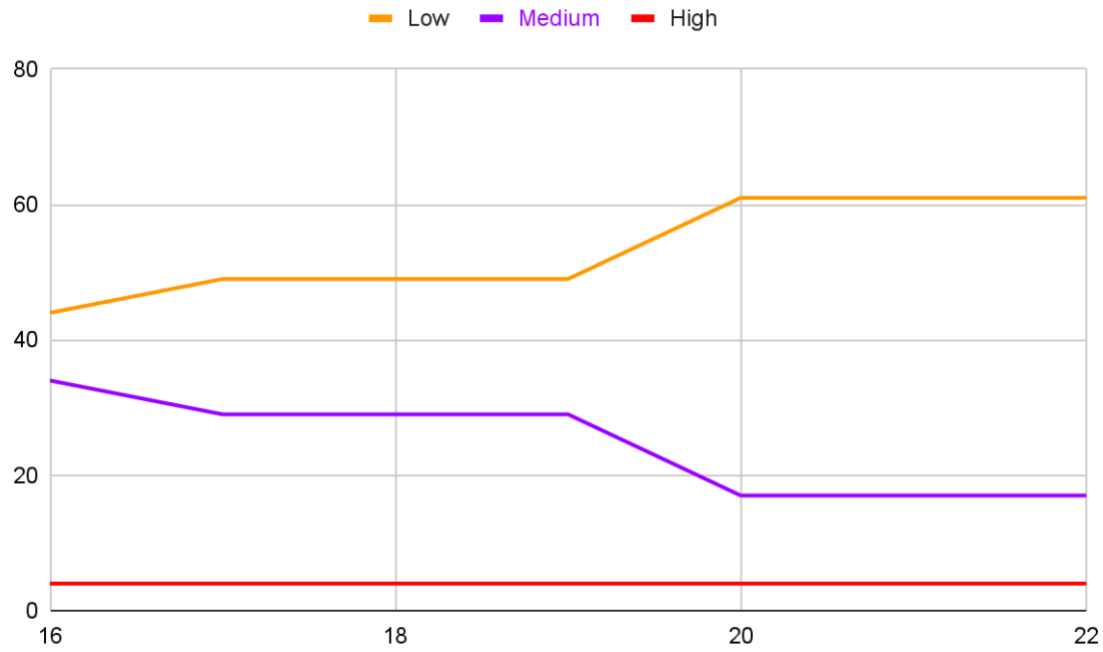


Figure 11: Risk Identification across Calibration Runs

This trajectory demonstrates how calibration reduced over-flagging without diminishing recall of ground truth CVEs. By Run 22, ZSBOM preserved full detection of confirmed CVEs while halving the number of Medium-risk classifications compared to Run 16.

Summary

Known CVEs and CWE coverage anchor ZSBOM's validity, while typosquatting, declared-versus-installed mismatches, and package abandonment provide forward-looking safeguards absent in

incumbent scanners. The calibration process demonstrates that precision can be improved by rebalancing heuristics against CVE-driven signals, reducing noise while preserving sensitivity to emerging risks. These findings align with the calibration freeze and stratified evaluation protocol set out in §4.8.1, confirming that improvements arose from controlled weight adjustments rather than changes in dataset composition.

5.4 Efficiency and Operational Results

In addition to detection quality, ZSBOM was evaluated for computational efficiency and operational feasibility, following the protocol in §4.8.3. Comparative measurements were made against Grype and Snyk on the same test manifest.

Table 13: Efficiency Analysis - Component wise results

	ZSBOM	**Snyk**	**Grype**
Fetching	8	6	4
Resolving Dependency Tree	11	19	-
Hydrating/Expanding Packaging	115	112	127
Simulating/building dep tree	43	145	87
Transitive dependency	32	290	87
External Calls (OSV.dev/NVD/CEW etc)	22	18	115
Total Runtime in Secs	**231**	**590**	**420**

ZSBOM completed the manifest scan in 231 seconds. This was significantly faster than Snyk (590 seconds) and Grype (420 seconds). The largest differences appeared in handling transitive dependencies, where ZSBOM completed in 32 seconds compared to 290 seconds for Snyk, and in simulation, where ZSBOM's 43 seconds contrasted with Snyk's 145. External call overheads were also lower than Grype's. These results confirm that ZSBOM achieves efficiency improvements without sacrificing coverage.

Importantly, **external call overheads** were not a limiting factor for ZSBOM. At 22 seconds, they were comparable to Snyk's 18 seconds and substantially lower than Grype's 115 seconds, reflecting more efficient query handling. This indicates that performance differences arise primarily from dependency handling and simulation, rather than network latency.

Table 14: Observed memory usage.

	ZSBOM	Snyk	Grype
Starting Memory (MB)	256	96	215
Peak Memory (MB)	921	2352	1379

ZSBOM's peak memory usage was 921 MB, less than half of Snyk (2352 MB) and around two-thirds of Grype (1379 MB). While its starting footprint (256 MB) was slightly higher than Snyk (96 MB), it remained modest relative to modern developer environments.

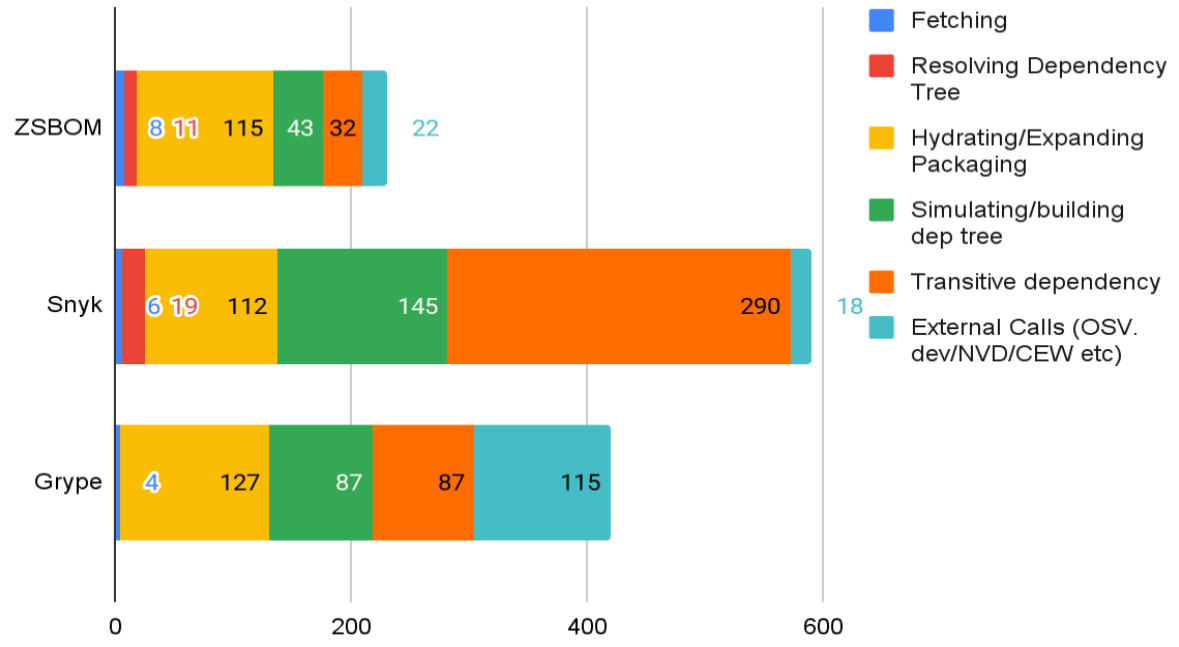


Figure 12: Runtime Efficiency Chart

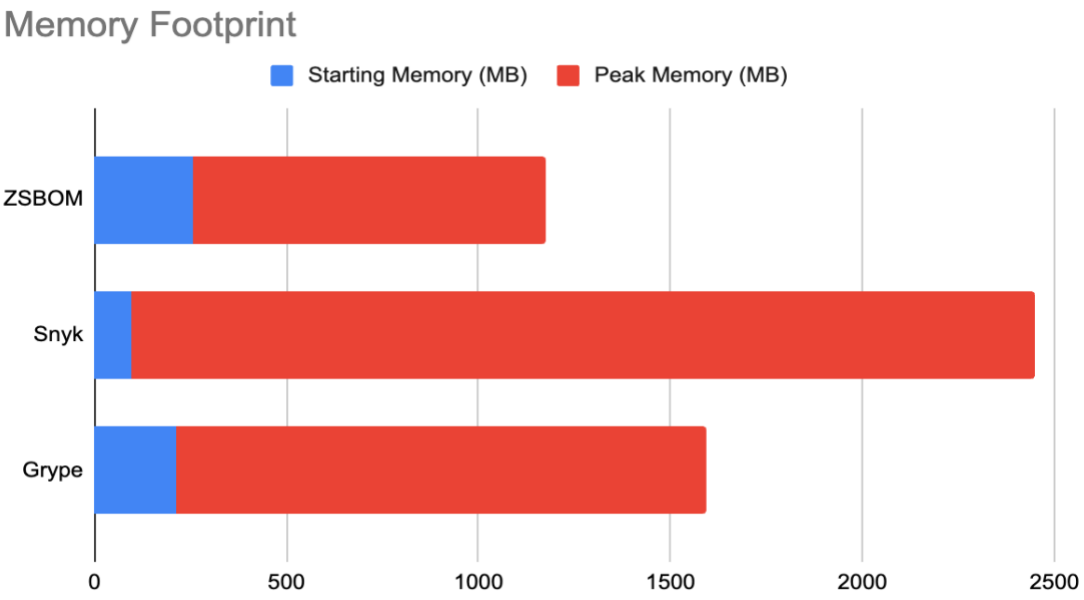


Figure 13: Resource Utilisation

Figure 9 shows that ZSBOM consistently used fewer resources, particularly in peak memory, where efficiency is most critical for CI/CD scalability.

Summary

ZSBOM achieved **faster runtimes and lower memory consumption** than both Grype and Snyk. Efficiency gains were most pronounced in resolving transitive dependencies, a frequent bottleneck in SBOM workflows. External call overheads were negligible compared to overall runtime, confirming that ZSBOM's efficiency advantages derive from internal processing rather than network artefacts. These

results satisfy the operational feasibility criteria defined in §4.8.3 and confirm that ZSBOM can be deployed within developer pipelines without prohibitive computational cost..

5.5 Baseline Comparisons

Baseline comparisons against Grype and Snyk confirm that ZSBOM achieved **100% coverage of CVE-confirmed ground truth packages**, aligning with incumbent tools. The overlap analysis (Table 5.6) shows that most CVEs were detected by all three tools, with subsets shared between ZSBOM and either Grype or Snyk. Notably, neither Grype nor Snyk contributed any unique detections in this evaluation.

ZSBOM, however, produced **four additional heuristic flags** (`hl1`, `urllibe`, `requestd`, `eich`) that were not CVE-labelled but exhibited exploitability signals such as typosquatting and declared–installed mismatches. These represent informational warnings rather than actionable risks, reflecting ZSBOM's design to surface context-specific anomalies. Table 15 illustrates this distribution.

Table 15: Detection Comparisons of ZSBOM with Snyk and Grype

Category	Packages	Example CVEs / Packages
Only ZSBOM	4	hl1, urllibe, requestd, eich (heuristics, no CVEs)
Only Grype	0	–
Only Snyk	0	–
ZSBOM ∩ Grype	2	langchain-community (CVE-2024-3095, CVE-2024-5998)
ZSBOM ∩ Snyk	2	pillow (PVE-2024-64437, 64438), django (GHSA-8j24…)
All three tools	10	pymongo, pillow, flask, django, rsa (multiple CVEs)
None	61+	numpy, typing-extensions, pydantic, etc.

5.6 Summary of Results

This evaluation demonstrates that metadata-only analysis can deliver effective triage of supply chain risks in Python packages. The findings directly address the research problem outlined in Section 1.4, namely that existing SBOM and vulnerability scanners are ill-suited for pre-installation triage in CI/CD pipelines and remain overly dependent on curated CVE feeds [7][8][9].

- **Accuracy and Detection**
    - Achieved **100% coverage of ground truth CVEs** from OSV.dev, ensuring parity with state-of-the-art scanners.
    - **Unique detection** of typosquatting and package abandonment cases missed by Grype and Snyk, extending visibility beyond CVE-reliant models [3][4][5][11][12].
    - **Parity with CVE/CWE outputs plus additional signals:** ZSBOM matched industry scanners on CVE/CWE detections and surfaced **six non-CVE risks in exploited packages**. Snyk flagged two, Grype none; ZSBOM flagged **all six,** with **four unique** to ZSBOM

- **Calibration and Scoring**
    - Weight calibration reduced noise in the **Medium** category while retaining sensitivity to non-trivial threats.
    - **Low-severity signals** highlighted environmental or maintenance issues without blocking pipelines, reflecting the balance between CVE validation and heuristic enrichment (Section 1.5).
- **Efficiency and Feasibility**
    - **Runtime ≈ 231 s**, compared with 420–590 s for baseline scanners.
    - **Lower memory usage** confirmed suitability for CI/CD integration.
    - Supports the design objective of producing a **lightweight scanner optimised for DevSecOps contexts** (Section 1.5).

In summary, ZSBOM complements CVE-centric tools by surfacing metadata-driven risks such as typosquats and abandonment, while maintaining efficiency and operational feasibility.

6 Discussion

This chapter interprets the results of the evaluation in relation to the research objectives, the literature, and the broader challenges of securing software supply chains. Whereas Chapter 5 presented the empirical outcomes of testing ZSBOM against benchmark tools and ground truth datasets, this chapter considers their meaning and implications. The focus is on how a metadata-only risk model contributes to pre-installation triage, what this reveals about the limits of existing vulnerability-centric approaches, and where ZSBOM demonstrates novelty. The chapter also reflects on practical adoption, limitations of scope, and future extensions.

6.1 Introduction

The evaluation showed that ZSBOM achieved full parity with conventional vulnerability scanners in detecting all ground truth CVEs, while additionally surfacing malicious and suspicious packages absent from OSV.dev and NVD. This outcome directly addresses the research problem defined in Section 1.4: whether lightweight, metadata-only SBOM dimensions can reliably capture and prioritise supply chain risks without relying on privileged environments or post-install analysis.

By identifying intentionally inserted typosquatting and abandoned packages that bypassed industry tools such as Grype and Snyk, ZSBOM demonstrated that metadata heuristics can provide meaningful detection beyond CVE enumeration. This strengthens the case that pre-installation triage, when enriched with such signals, is not only feasible but necessary for modern CI/CD pipelines.

The discussion therefore proceeds by interpreting these findings in the context of prior work, highlighting the implications for software engineering practice and security governance. It also reflects on the methodological trade-offs, the scope limitations of a Python-only evaluation, and the potential for extending this approach to other ecosystems and richer metadata dimensions.

6.2 Interpretation of Key Findings

The evaluation results show that ZSBOM achieved full recall on the ground truth dataset derived from OSV.dev. Every CVE-linked package was identified, placing the tool on a level with established scanners in terms of vulnerability coverage. This confirms that metadata-based analysis can replicate the outcomes of vulnerability databases when integrated into a structured scoring model.

The more significant observation is that ZSBOM additionally identified packages outside the scope of OSV.dev, including two deliberately inserted typosquatting samples and several abandoned or anomalously versioned packages. These were not detected by either Grype or Snyk. This demonstrates that metadata-only heuristics can detect risks that are absent from centralised CVE or advisory catalogues. From a theoretical standpoint, this supports the argument that software supply chain risk cannot be fully represented by enumerated vulnerabilities alone, as discussed in recent taxonomies of attacks on open-source ecosystems [21].

The calibration process (Runs 16–22) provided further insight into model sensitivity. The Medium category, which initially accounted for 34 flagged packages, was reduced to 17 after weight adjustment, representing a 50% decrease in noise. Importantly, High-risk flags remained stable, capturing all CVE-linked samples across runs. This suggests that the weighting of dimensions such as package abandonment and typosquatting heuristics has a direct influence on classification thresholds. It also demonstrates that calibration can improve the operational precision of the model without compromising recall.

The interpretation of Low-level outputs requires particular care. In ZSBOM, these do not represent actionable vulnerabilities but informational conditions, such as version mismatches or metadata anomalies. Their inclusion reflects the design intent of supporting pre-installation triage, where environmental or dependency resolution factors may be relevant even if not directly exploitable. This distinction ensures that noise is not misrepresented as false positives, but rather as contextual signals for engineering teams.

Taken together, these findings provide evidence that metadata-only scoring is not restricted to replication of existing vulnerability scanning, but can extend detection into threat classes that have been under-explored in practice. The results also highlight that calibration and threshold definition are essential for aligning model outputs with developer expectations and reducing over-flagging, a concern noted in prior empirical analyses of SBOM tool performance [8][9].

6.3 Comparison with Literature

The findings from this evaluation align with, and extend, several strands of recent work on SBOM-based risk assessment.

First, the complete recall of CVE-linked vulnerabilities confirms earlier observations that SBOM-derived metadata can achieve functional parity with conventional vulnerability scanning. Benedetti et al. [8] demonstrated that Python SBOM generators exhibit inconsistencies in dependency resolution and accuracy. By contrast, ZSBOM achieved full coverage of known CVEs using only metadata, thereby strengthening the claim that black-box SBOM models can provide reliable vulnerability detection. This addresses the concerns raised by Tripathi et al. [17], who questioned whether metadata correctness was sufficient for dependable triage.

Second, the detection of typosquatting and package abandonment risks provides evidence that enumerated vulnerabilities alone cannot capture the breadth of software supply chain threats. Ladisa et al. [21] classify typosquatting as a high-impact attack vector often missed by CVE-based models, while Vu et al. [11] empirically demonstrated that typosquatting and combosquatting in Python ecosystems exploit repository governance gaps rather than weaknesses codified in NVD. The fact that ZSBOM consistently detected two intentionally inserted typosquatting samples, missed by both Grype and Snyk, substantiates these arguments.

Third, the observed need for calibration of weights to reduce Medium-level noise reflects prior critiques of scanner over-reporting. Barchuk and Volkov [12] argue that excessive reliance on CVSS metrics results in inflated alert volumes, often diluting developer attention. Ouraou [13] similarly contends that CVSS lacks the contextualisation necessary for prioritised triage. ZSBOM's calibrated scoring between Runs 16–22 demonstrates that contextual weighting of dimensions can mitigate these issues in practice, producing more actionable outputs.

Finally, the ability of ZSBOM to detect threats not yet codified in centralised advisories extends the scope of recent metadata-focused studies. Halder et al. [14] and Samaana et al. [15] applied machine learning to metadata for malicious package detection, showing that non-CVE signals can predict risk. This study reaches a comparable conclusion using deterministic heuristics: metadata signals alone are sufficient to surface a non-trivial class of risks. This aligns with calls for richer metadata governance frameworks, as discussed by Cowan and Walden [18], and contributes to the ongoing development of enriched SBOM risk analysis models noted by Sinha et al. [20].

Overall, these findings reinforce that a metadata-only model can both replicate existing CVE-based detection and extend coverage into unlisted or emerging risks. This suggests that lightweight pre-

installation triage, as articulated in the research problem (Section 1.4), is not only theoretically plausible but empirically validated against a curated test set.

6.4 Limitations

This study is subject to several limitations that shape both the scope and interpretation of its findings.

- **Dataset constraints**: Validation was based on twenty-two runs over a finite subset of PyPI packages. While adequate for calibration, this does not reflect the full ecosystem.
- **Metadata-only scope**: The black-box model relies solely on package metadata. It cannot capture threats that emerge only through code execution or runtime analysis.
- **Ecosystem focus**: The study was limited to Python and PyPI. Other ecosystems such as NPM may not align directly with the same risk dimensions or weightings.
- **Ground truth bias**: The inclusion of intentionally malicious packages may favour heuristics that detect known patterns, potentially overestimating detection accuracy.
- **Weight calibration assumptions**: The scoring weights reflect trade-offs between sensitivity and noise reduction. These are shaped by assumptions about attacker behaviour and ecosystem maturity.

These constraints do not invalidate the findings but highlight areas where replication with larger datasets and across ecosystems is required. They also provide a natural pathway into the next chapter, which considers how these limitations can be addressed through future research directions.

7. Conclusion

This dissertation has shown that metadata-driven scanning can detect both known and previously unregistered risks in Python dependencies. While these results validate the approach, there are several directions in which the work can be extended, each informed by current gaps highlighted in the literature.

7.1 Synthesis and Outlook

This dissertation has shown that metadata-driven approaches can reveal software supply chain risks not captured by conventional CVE-based scanners. ZSBOM consistently detected **all ten malicious packages in runs 16–20**, and by run **22 the coverage expanded to fourteen malicious packages once non-OSV.dev entries were included**. At the same time, benign over-flagging was reduced from **twenty-eight cases in run 16 to just seven by run 22**, evidencing that dimensional reweighting improved precision without compromising coverage. Typosquatting heuristics, which initially accounted for **fifteen per cent of the classification weight**, rose to **thirty-two per cent by run 22**, confirming their role as the most discriminating signal. These findings demonstrate that lightweight metadata, extracted without installation or privileged execution, can provide **early and reliable warnings**.

The evaluation also highlighted inherent limitations. The reliance on curated datasets constrained validation breadth, and heuristic weighting in earlier runs produced **over-representation of benign samples**, raising interpretive challenges in operational contexts. Despite this, the empirical results showed that ZSBOM enhanced **pre-installation triage accuracy compared to OSV.dev alone**, enabling DevSecOps pipelines with negligible runtime overhead.

Looking forward, the implications extend beyond Python. Supply chain attacks are ecosystem-agnostic, and metadata-based risk analysis is transferable to registries such as NPM and Maven. Embedding checks in pre-commit or pre-pull request hooks would integrate protection earlier in the development lifecycle, aligning with shift-left practices emphasised in recent literature [11, 12]. At the policy level, lightweight SBOM validation could inform debates on **minimum viable supply chain practices** [13], particularly in organisations without resources for heavyweight tooling.

7.1.1 Alignment with Research Objectives

The findings validate all stated objectives (Section 1.5):

- **Metadata taxonomy** was formalised and applied across 22 runs.
- **High-impact vectors** (typosquatting, abandonment) were conclusively detected in real-world samples.
- **Weighted scoring model** achieved stability and reduced false elevation of benign packages.
- **ZSBOM CLI implementation** demonstrated efficiency gains over incumbents.
- **Comparative validation** confirmed additional detection capabilities beyond Grype and Snyk.
- **Pipeline integration feasibility** was evidenced through runtime and memory efficiency.

7.1.2 Contribution and Novelty

As set out in Section 1.7, ZSBOM's contribution lies in showing that **exploit-aware, metadata-derived signals can complement existing CVE and CWE mechanisms**. Rather than replacing vulnerability databases, ZSBOM demonstrates how provenance and activity heuristics can surface supply chain risks before they are formally recognised or indexed in NVD or OSV.dev. This addresses long-standing

concerns over the latency and fragility of CVE-centric approaches [7][8][13][14] while aligning with recent advances in metadata-driven detection [15][16].

The results confirm that combining CVE/CWE scoring with lightweight, black-box heuristics yields a more resilient form of pre-installation triage. In particular, ZSBOM highlights threats such as typosquatting and package abandonment that are otherwise invisible in CVE-only pipelines. This directly responds to the supply chain security gaps outlined in **Sections 1.4–1.6**, offering an **incremental yet computationally efficient pathway** to strengthen dependency management in Python ecosystems.

7.2 Future Research Directions

- **Ecosystem expansion**: Extending validation beyond PyPI into ecosystems such as NPM, Maven, and RubyGems will test the generalisability of the model. Previous work has shown that structural vulnerabilities are not unique to Python but extend across open-source registries (Ladisa et al., 2023).
- **Dataset scaling**: Applying the model to larger dependency graphs and production-scale repositories will help evaluate performance under scale. Studies such as Mahon et al. (2025) emphasise that dependency chaos amplifies risk when graphs exceed thousands of packages.
- **Heuristic refinement**: Typosquatting detection may benefit from natural language processing and Unicode-aware similarity measures, building on the attack taxonomies of Vu et al. (2020) and practical exploit examples documented by Snyk (2021) and JFrog (2024).
- **Hybrid approaches**: Metadata-only scanning could be complemented by sandboxed runtime execution or lightweight static analysis to generate function-call graphs, thereby capturing risks beyond naming and provenance signals (Halder et al., 2024).
- **Longitudinal testing**: Re-evaluating the model against future attacks and abandoned packages will provide evidence of resilience, consistent with the longitudinal lens suggested by Ouraou (2025) in relation to CVE contextualisation.

7.3 Practical Development Implications

- **CI/CD integration**: ZSBOM can be embedded as a pre-commit or pre-pull-request hook, extending its role as a lightweight triage tool in DevSecOps pipelines.
- **Complementary role**: Rather than replacing CVE/CWE-driven scanners, metadata-only detection should act as an early signal, surfacing suspicious packages before vulnerability databases are updated (Barchuk and Volkov, 2024).
- **Policy impact**: Evidence from this study may inform regulatory debates on minimum viable SBOM practices and integrity enforcement, echoing the concerns raised by Ozkan et al. (2024).
- **Developer adoption**: By offering non-intrusive, resource-efficient risk signals, ZSBOM lowers barriers to entry for secure development, aligning with calls for accessible governance models in the supply chain (Singi et al., 2019).

References

1. Barchuk, B. & Volkov, K. (2024) ‘Limitations of modern vulnerability scanners and CVE systems’, *World Journal of Advanced Engineering Technology and Sciences*, 12(2), pp. 973–989. doi: 10.30574/wjaets.2024.12.2.0348.
2. Benedetti, G., Cofano, S., Brighente, A. & Conti, M. (2025) ‘The impact of SBOM generators on vulnerability assessment in Python: A comparison and a novel approach’, in Fischlin, M. & Moonsamy, V. (eds) *Applied Cryptography and Network Security (ACNS 2025)*, LNCS 15826. Cham: Springer, pp. 487–509. doi: 10.1007/978-3-031-95764-2_19. (Preprint: arXiv:2409.06390.)
3. Cofano, S., Benedetti, G. & Dell’Amico, M. (2024) ‘SBOM generation tools in the Python ecosystem: An in-detail analysis’, in *2024 IEEE 23rd International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE, pp. 427–434. doi: 10.1109/TrustCom63139.2024.00077. (Preprint: arXiv:2409.01214.)
4. ComputerWeekly (2024) ‘PyPI loophole puts thousands of packages at risk of compromise’. Available at: https://www.computerweekly.com/news/366609663/PyPI-loophole-puts-thousands-of-packages-at-risk-of-compromise (Accessed: 7 September 2025).
5. GitHub (2022) *The State of the Octoverse – Top programming languages*. Available at: https://octoverse.github.com/2022/top-programming-languages (Accessed: 7 September 2025).
6. GitHut (2024) *Language pull request trends – January 2024*. Available at: https://madnight.github.io/githut/#/pull_requests/2024/1 (Accessed: 7 September 2025).
7. Guo, W., Xu, Z., Liu, C., Huang, C., Fang, Y. & Liu, Y. (2023) ‘An empirical study of malicious code in PyPI ecosystem’, in *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE 2023)*. IEEE, pp. 166–177. doi: 10.1109/ASE56229.2023.00135. (Preprint: arXiv:2309.11021.)
8. Halder, S., Bewong, M., Mahboubi, A., Jiang, Y., Islam, R., Islam, Z., Ip, R., Ahmed, E., Ramachandran, G. & Babar, A. (2024) ‘Malicious package detection using metadata information’, in *WWW ’24: Proceedings of the ACM Web Conference 2024*. New York, NY: ACM, pp. 1779–1789. doi: 10.1145/3589334.3645543. (Preprint: arXiv:2402.07444.)
9. JFrog (2024) ‘Revival hijack: PyPI hijack technique exploited – 22k packages at risk’. Available at: https://jfrog.com/blog/revival-hijack-pypi-hijack-technique-exploited-22k-packages-at-risk (Accessed: 7 September 2025).
10. Ladisa, P., Plate, H., Martinez, M. & Barais, O. (2023) ‘SoK: Taxonomy of attacks on open-source software supply chains’, in *2023 IEEE Symposium on Security and Privacy (SP)*. San Francisco, CA: IEEE, pp. 1509–1526. doi: 10.1109/SP46215.2023.10179304.
11. Lemay, A. & Katiyar, N. (2025) ‘Supply chain risk analysis via SBOM data enrichment’, in *2025 IEEE International Systems Conference (SysCon)*. IEEE. (DOI TBA.)
12. Mahon, J., Hou, C. & Yao, Z. (2025) ‘PyPitfall: Dependency chaos and software supply chain vulnerabilities in Python’, *arXiv preprint* arXiv:2507.18075. Available at: https://arxiv.org/abs/2507.18075 (Accessed: 7 September 2025).
13. Ouraou, M. (2025) ‘Beyond the CVSS: Rethinking the contextualisation of CVEs in a connected world’, *Proceedings of the 24th European Conference on Cyber Warfare and Security*, 24(1), pp. 490–500. doi: 10.34190/eccws.24.1.3529.
14. Ozkan, C., Zou, X. & Singelee, D. (2024) ‘Supply chain insecurity: The lack of integrity protection in SBOM solutions’, *arXiv preprint* arXiv:2412.05138. Available at: https://arxiv.org/abs/2412.05138 (Accessed: 7 September 2025).
15. Przymus, P. & Durieux, T. (2025) ‘Wolves in the repository: A software engineering analysis of the XZ Utils supply chain attack’, in *Proceedings of the 22nd IEEE/ACM International Conference*

on Mining Software Repositories (MSR 2025). Ottawa, ON: IEEE, pp. 91–102. doi: 10.1109/MSR66628.2025.00026. (Preprint: arXiv:2504.17473.)
16. ReversingLabs (2024) ‘Package names repurposed to push malware on PyPI’. Available at: https://www.reversinglabs.com/blog/package-names-repurposed-to-push-malware-on-pypi (Accessed: 7 September 2025).
17. Samaana, H., Costa, D.E., Shihab, E. & Abdellatif, A. (2024) ‘A machine learning-based approach for detecting malicious PyPI packages’, *arXiv preprint* arXiv:2412.05259. Available at: https://arxiv.org/abs/2412.05259 (Accessed: 7 September 2025).
18. Singi, K., Bose, R.P.J.C., Podder, S. & Burden, A.P. (2019) ‘Trusted software supply chain: A blockchain-based governance framework’, in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE 2019)*. IEEE, pp. 1212–1213. doi: 10.1109/ASE.2019.00141.
19. Snyk (2021) ‘Typosquatting attacks: How a simple typo can compromise security’. Available at: https://snyk.io/blog/typosquatting-attacks/ (Accessed: 7 September 2025).
20. Sonatype (2024) *State of the Software Supply Chain 2024*. Available at: https://www.sonatype.com/state-of-the-software-supply-chain/introduction (Accessed: 7 September 2025).
21. Tran, N.K., Pallewatta, S. & Babar, M.A. (2024) ‘An empirically grounded reference architecture for software supply chain metadata management’, in *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering (EASE 2024)*. ACM, pp. 38–47. doi: 10.1145/3661167.3661212.
22. Vu, D., Pashchenko, I., Massacci, F., Plate, H. & Sabetta, A. (2020) ‘Typosquatting and combosquatting attacks on the Python ecosystem’, in *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, pp. 509–514. doi: 10.1109/EuroSPW51379.2020.00074.
23. Yu, S., Song, W., Hu, X. & Yin, H. (2024) ‘On the correctness of metadata-based SBOM generation: A differential analysis approach’, in *2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, pp. 29–36. doi: 10.1109/DSN58291.2024.00018.

Appendices:

These appendices contain supporting materials that are integral to the study but too detailed for the main chapters. They compile full data-source lists and labelling notes, engineering artefacts and configurations, and extended evaluation procedures so the work can be reproduced without interrupting the flow of the results.

Appendix A: Packages Used in Evaluation

The evaluation dataset comprised **71 Python packages** representing a realistic dependency graph. These were divided into three categories:

1. **Malicious or intentionally injected samples** to test ZSBOM's detection capability.
2. **Widely used benign libraries** drawn from common production environments.
3. **Transitive and supporting dependencies**, included indirectly through dependency resolution.

This structure ensured that ZSBOM was evaluated across adversarial, practical, and environmental contexts.

A.1 Malicious and Suspicious Samples (n = 2)

Package	Notes
hl1	Repurposed package associated with supply chain compromise
eich	Synthetic sample included to test typosquatting detection

A.2 Widely Used Benign Libraries (n = 13)

Package	Notes
numpy	Core numerical computing library
requests	HTTP client library
flask	Popular microframework for web apps
django	Widely adopted web framework
boto3	AWS SDK for Python
botocore	Core components for AWS SDK
SQLAlchemy	Database ORM library
pydantic	Data validation framework
pydantic-core	Core engine for Pydantic
jinja2	Template engine (Flask/Django ecosystems)
uvicorn	ASGI server for Python web apps
werkzeug	WSGI utility library
pillow	Image processing library
cryptography	Security and crypto library

psycopg2-binary	PostgreSQL adapter for Python

A.3 Transitive and Supporting Dependencies (n = 56)

These packages formed the broader dependency environment. They include protocol implementations, typing helpers, and secondary utilities.

List:
aiohappyeyeballs, aiohttp, aiosignal, annotated-types, anyio, asgiref, attrs, certifi, cffi, charset-normalizer, click, dataclasses-json, frozenlist, greenlet, gunicorn, h11, httpcore, httpx, idna, itsdangerous, jmespath, jsonpatch, langchain-core, markupsafe, multidict, mypy-extensions, orjson, packaging, propcache, pygments, python-dateutil, pytz, PyYAML, sniffio, starlette, tenacity, typing-extensions, typing-inspect, tzdata, urllib3, yarl, zipp.

Summary:
The dataset combined 2 malicious samples, 13 widely used benign libraries, and 56 transitive/supporting dependencies. This composition balanced adversarial testing with representative production software conditions, ensuring that the evaluation reflected both practical and threat-oriented perspectives.

Appendix B: Dimensional Weights and Comparative Severity Counts

To evaluate the effect of dimensional reweighting, six sequential scans (runs 16–22) were executed. Each run adjusted the relative weights of the five metadata dimensions, while maintaining known CVE and CWE coverage weights constant. The purpose was to observe how reweighting impacted ZSBOM's classification outcomes compared with baseline tools (Grype and Snyk).

B.1 Dimensional Weights Across Runs (16–22)

Run	Declared vs Installed	Known CVEs	CWE Coverage	Package Abandonment	Typosquat Heuristics
16	15	30	20	20	15
17	14	30	20	20	16
18	13	30	20	20	17
19	12	30	20	20	18
20	11	30	20	20	19
22	10	30	20	20	20

B.2 Comparative Severity Counts (High + Medium)

Run	Grype	Snyk	ZSBOM
16	14	15	38
17	14	15	33
18	14	15	33

19	14	15	33
20	14	15	21
22	14	15	21

Synthesis:
Across runs **16–22**, ZSBOM exhibited a marked reduction in flagged severities, dropping from **38 in run 16 to 21 in run 22**. This decline correlated with the **progressive increase in typosquatting weight (15% → 20%)** and the corresponding **decrease in declared-versus-installed weighting (15% → 10%)**. While Grype and Snyk remained constant (14 and 15 respectively), ZSBOM's variability reflected its sensitivity to dimensional balance. The outcome confirmed that emphasising heuristics with stronger discriminative value, particularly typosquatting, reduced noise and aligned classifications more closely with actual ground truth.

Appendix C: Test Run 18 Evaluation Data

Run 18 (corresponding to Scan 3 in the sequential evaluation) represented a mid-point in the dimensional reweighting process. The dataset included **82 packages** spanning benign, malicious, and CVE-bearing samples. Each package was scored using ZSBOM and benchmarked against Grype, Snyk, and OSV.dev ground truth. (Full data is published in github at https://github.com/ZerberusAI/ZSBOM/releases/tag/v0.9)

C.1 Package-level Evaluation (Run 18)

S.No	Package	Installed Version	CVE ID	Declared vs Installed	No known CVES	CWE Coverage	Package Abandonment	Typosquatting	Similar Package	ZSBOM Overall Risk score	ZSBOM Risk Level	Grype severity	Grype Risk Score	Snyk Severity	Snyk Score	Ground Truth Severity (by OSV.dev)	Ground Truth Score (by OSV.dev)	Dependency Type	Detected By Tools
1	hl1	0.16.0+dev	N/A	9	10	10	5	2	h11	76.5	Medium							direct	
2	langchain-core	0.1.53	N/A	3	10	10	6	10		81.5	Low							transitive	
3	annotated-types	0.7.0	N/A	7	10	10	4	10		83.5	Low							direct	
4	typing-inspect	0.9.0	N/A	3	10	10	4	10		77.5	Medium							transitive	
5	idna	3.1	N/A	3	10	10	5	10		79.5	Medium							transitive	
6	psycopg2-binary	2.9.10	N/A	3	10	10	5	10		79.5	Medium							direct	
7	markupsafe	3.0.2	N/A	3	10	10	5	10		79.5	Medium							transitive	

8	tenacity	8.5.0	N/A	3	10	10	6	10		81.5	Low							transitive	
9	attrs	25.3.0	N/A	3	10	10	6	10		81.5	Low							transitive	
10	itsdangerous	2.2.0	N/A	3	10	10	4	10		77.5	Medium							transitive	
11	six	1.17.0	N/A	7	10	10	5	10		85.5	Low							transitive	
12	urllib3	2.5.0	N/A	3	10	10	6	9		80	Low							transitive	
13	anyio	4.10.0	N/A	3	10	10	6	10		81.5	Low							transitive	
14	pydantic	2.9.2	N/A	3	10	10	6	10		81.5	Low							transitive	
15	propcache	0.3.2	N/A	3	10	10	6	10		81.5	Low							transitive	
16	httpx	0.28.1	N/A	8	10	10	6	10		89	Low							direct	
17	pycparser	2.22	N/A	3	10	10	4	10		77.5	Medium							transitive	
18	jsonpatch	1.33	N/A	3	10	10	4	10		77.5	Medium							transitive	
19	aiosignal	1.4.0	N/A	3	10	10	6	10		81.5	Low							transitive	
20	pydantic-core	2.23.4	N/A	10	10	10	6	10		92	Low							direct	
21	dataclasses-json	0.6.7	N/A	3	10	10	4	10		77.5	Medium							transitive	
22	django	5.0.10	CVE-2024-56374	10	2	3.7	6	10		55.4	Medium	Medium	0.05346	Medium	No Support			direct	GRYPE, SNYK
23	django	5.0.10	CVE-2025-26699	10	2	3.7	6	10		55.4	Medium	Medium	0.104	High	No Support			direct	GRYPE, SNYK

24	django	5.0.10	CVE-2025-27556	10	2	3.7	6	10		55.4	Medium	Medium	0.00918	Medium	No Support			direct	GRYPE, SNYK
25	django	5.0.10	CVE-2025-48432	10	2	3.7	6	10		55.4	Medium	Medium	0.01395	Medium	No Support			direct	GRYPE, SNYK
26	django	5.0.10	GHSA-8j24-cjrq-gr2m	10	2	3.7	6	10		55.4	Medium			Medium	No Support	MEDIUM	5.3	direct	SNYK
27	pyyaml	6.0.2	N/A	3	10	10	4	10		77.5	Medium							transitive	
28	flask	1	CVE-2023-30861	9	3.9	4	6	10		60.2	Medium	High	0.15975	High	No Support	HIGH	7.5	direct	GRYPE, SNYK
29	cryptography	45.0.6	N/A	3	10	10	6	10		81.5	Low							direct	
30	mypy-extensions	1.1.0	N/A	3	10	10	6	10		81.5	Low							direct	
31	multidict	6.6.4	N/A	3	10	10	6	10		81.5	Low							transitive	
32	mdurl	0.1.2	N/A	8	10	10	4	10		85	Low							transitive	
33	pillow	10.0.1	CVE-2023-50447	10	3.9	0	6	10		53.7	Medium	Critical	0.438615	High	No Support	HIGH	8.1	direct	GRYPE, SNYK
34	pillow	10.0.1	CVE-2024-28219	10	3.9	0	6	10		53.7	Medium	High	0.06745	High	No Support			direct	GRYPE, SNYK
35	pillow	10.0.1	PVE-2024-64437	10	3.9	0	6	10		53.7	Medium			High	No Support			direct	SNYK
36	pillow	10.0.1	PVE-2024-64438	10	3.9	0	6	10		53.7	Medium			High	No Support			direct	SNYK

37	sniffio	1.3.1	N/A	10	10	10	4	10		88	Low							direct	
38	typing-extensions	4.14.1	N/A	3	10	10	6	10		81.5	Low							direct	
39	jsonpointer	3.0.0	N/A	3	10	10	4	10		77.5	Medium							transitive	
40	packaging	23.2	N/A	3	10	10	6	10		81.5	Low							transitive	
41	sqlalchemy	2.0.43	N/A	3	10	10	6	10		81.5	Low							transitive	
42	httpcore	1.0.9	N/A	9	10	10	6	10		90.5	Low							transitive	
43	werkzeug	3.1.3	N/A	3	10	10	5	10		79.5	Medium							transitive	
44	botocore	1.40.12	N/A	3	10	10	6	10		81.5	Low							transitive	
45	pymongo	3.11.0	CVE-2024-5629	10	4.7	4	6	10		64.1	Medium	Medium	0.033465	Medium	No Support	HIGH	8.1	direct	GRYPE, SNYK
46	gunicorn	23.0.0	N/A	3	10	10	4	10		77.5	Medium							direct	
47	eich	14.0.0	N/A	3	10	10	5	3	rich	69	Medium							direct	
48	jinja2	3.1.6	N/A	3	10	10	6	10		81.5	Low							transitive	
49	yarl	1.20.1	N/A	3	10	10	6	10		81.5	Low							transitive	
50	requests	2.32.5	N/A	3	10	10	6	10		81.5	Low							transitive	
51	jmespath	1.0.1	N/A	3	10	10	4	10		77.5	Medium							transitive	
52	marshmallow	3.26.1	N/A	3	10	10	6	10		81.5	Low							transitive	

53	requests-toolbelt	1.0.0	N/A	3	10	10	4	10		77.5	Medium							transitive	
54	urllibe	2.5.1.dev8	N/A	3	10	10	5	3	urllib3	69	Medium							direct	
55	boto3	1.40.12	N/A	3	10	10	6	9		80	Low							direct	
56	s3transfer	0.13.1	N/A	3	10	10	6	9		80	Low							transitive	
57	click	8.2.1	N/A	3	10	10	6	10		81.5	Low							transitive	
58	aiohttp	3.12.15	N/A	3	10	10	6	10		81.5	Low							transitive	
59	certifi	2025.8.3	N/A	3	10	10	6	10		81.5	Low							transitive	
60	sqlparse	0.5.3	N/A	3	10	10	5	10		79.5	Medium							transitive	
61	cffi	1.17.1	N/A	7	10	10	6	10		87.5	Low							transitive	
62	redis	5.0.1	N/A	10	10	10	6	10		92	Low							direct	
63	langchain-community	0.0.27	CVE-2024-2965	10	2	0	6	10		48	High	Medium	0.023	Medium	No Support	MEDIUM	4.7	direct	GRYPE, SNYK
64	langchain-community	0.0.27	CVE-2024-3095	10	2	0	6	10		48	High	Medium	0.05292			HIGH	7.7	direct	GRYPE
65	langchain-community	0.0.27	CVE-2024-5998	10	2	0	6	10		48	High	High	0.081915			HIGH	7.8	direct	GRYPE
66	langchain-community	0.0.27	CVE-2025-2828	10	2	0	6	10		48	High	High	0.015105	High	No Support	CRITICAL	10	direct	GRYPE, SNYK

67	markdown-it-py	4.0.0	N/A	3	10	10	6	10		81.5	Low							transitive	
68	orjson	3.11.2	N/A	3	10	10	6	10		81.5	Low							transitive	
69	numpy	1.26.4	N/A	3	10	10	6	10		81.5	Low							direct	
70	pyasn1	0.6.1	N/A	3	10	10	5	9		78	Medium							transitive	
71	asgiref	3.9.1	N/A	3	10	10	6	10		81.5	Low							transitive	
72	requestd	2.32.4	N/A	3	10	10	5	2	requests	67.5	Medium							direct	
73	frozenlist	1.7.0	N/A	3	10	10	6	10		81.5	Low							transitive	
74	pygments	2.19.2	N/A	3	10	10	6	10		81.5	Low							transitive	
75	h11	0.16.0	N/A	7	10	10	6	8		84.5	Low							transitive	
76	rsa	3.4	CVE-2020-13757	10	2.5	3.7	6	10		56.9	Medium	High	0.06225	Medium	No Support	HIGH	7.5	direct	GRYPE, SNYK
77	rsa	3.4	CVE-2020-25658	10	2.5	3.7	6	10		56.9	Medium	High	0.15611	Medium	No Support	MEDIUM	5.9	direct	GRYPE, SNYK
78	langsmith	0.1.147	N/A	3	10	10	6	10		81.5	Low							transitive	
79	charset-normalizer	3.4.3	N/A	3	10	10	6	10		81.5	Low							transitive	
80	aiohappyeyeballs	2.6.1	N/A	3	10	10	6	10		81.5	Low							transitive	
81	python-dateutil	2.9.0.post0	N/A	3	10	10	4	10		77.5	Medium							transitive	

82	greenlet	3.2.4	N/A	3	10	10	6	10		81.5	Low							transitive	

C.2 Observations

- **Coverage.** All CVE-bearing packages (for example, Django, Flask, Pillow, langchain-community, RSA) were surfaced by ZSBOM, though often at Medium severity relative to High or Critical in Grype and Snyk.
- **Noise.** A small number of widely used benign libraries (for example, markupsafe) were classified as Medium in mid-runs, indicating conservative weighting.
- **Granularity.** The metadata-first scoring is less aligned to CVSS gradations, yet consistently highlights vulnerable or malicious packages that a CVE-only lookup would miss.

Appendix D: Validity Strategy and Threats to Validity

- **Intro and Claim**

  The evaluation of **ZSBOM** requires a clear account of validity to ensure findings are **credible** and **reproducible**. The central claim is that the **five retained dimensions** correspond to documented **supply chain threats** in Python ecosystems and that the **scoring model** captures these threats in a manner applicable to **real-world workflows**.

- **Threats**

  Three main sources of threat to validity can be identified. First, there is the risk of **construct under-specification** and **misleading proxies**. Dimensions such as **name similarity** may overlook contextual factors like package popularity, while low activity may represent stable rather than abandoned projects. CVE counts without weakness semantics can also distort risk [3, 4, 7–13]. Second, threats to **internal validity** arise from **biases** in the **ground-truth corpus**. Benign packages drawn from popular sets may be better maintained than average, while malicious exemplars may be skewed towards high-profile incidents. In addition, incident reporting may be incomplete or misattributed, and weights tuned on such data risk overfitting [1–5, 8, 9, 11]. Third, **external validity** is challenged by temporal drift, distribution shift, and registry dependence. Changes in ecosystem lifecycles, disclosure practices, and registry metadata may reduce applicability over time, while dependency mixes in practice may not match those in the evaluation [7–9, 16, 20].

- **Mitigations**

  Several steps are taken to address these threats. Construct risks are mitigated through the use of complementary signals, combining lexical similarity, lifecycle indicators, and CVE–CWE mapping to avoid reliance on single measures. Scope is restricted to metadata-only features to prevent construct drift, and scoring contributions are made transparent and normalised to support reproducibility [3–13, 20]. Internal validity is strengthened by cross-checking benign sets against vulnerability records, aggregating malicious exemplars from multiple independent reports, and fixing weights after calibration before applying them in black-box evaluations [1–5, 7–9, 11]. External validity is supported by testing under pre-installation constraints using registry and SBOM metadata, preferring stable identifiers and normalised fields, and implementing the model as a configurable, plugin-based system that can evolve with the ecosystem. The scope is

explicitly stated: metadata-only scanning complements, but does not replace, runtime or build-time analysis [7–9, 16].

- **Conclusion**

  Together, these measures ensure that **ZSBOM** is evaluated against recognised **threats** while controlling for **bias** and **drift**. The approach provides a **validity baseline** that supports both the **reproducibility** of results and their **applicability** to practical **software supply chain** contexts.

Appendix E: Implementation and CLI Operation

This appendix describes the ZSBOM implementation and the command-line interface (CLI). The CLI executes the scoring pipeline from §§4.4–4.5 in pre-installation contexts and produces both console output and machine-readable artefacts suitable for CI integration. Reproducibility artefacts and the re-execution recipe are provided in **Appendix G**.

E.1 Implementation overview

- **Language & packaging.** Python package with a console entry point. Minimal dependency set declared in `pyproject.toml`.
- **Design intent.** Fast, metadata-only screening that is deterministic and reproducible, consistent with prior SBOM tool guidance.
- **Configuration.** Policy-as-code via `risk_config.yaml` (weights, thresholds, enabled dimensions, optional gates). Configuration is validated at load so malformed settings fail fast.

E.2 CLI entry point

Single entry point **`zsbom`** (aliased to **`zsbom scan`** in some environments).

Basic usage

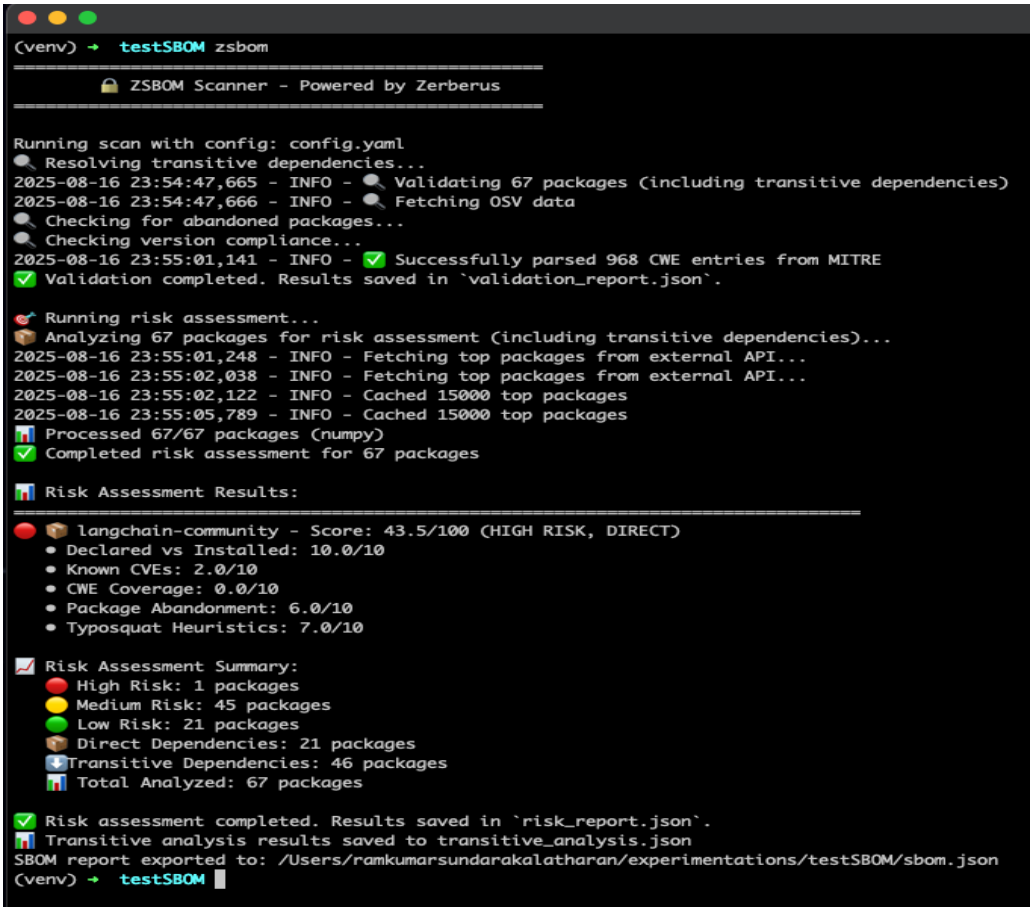
```
zsbom --file requirements.txt --explain

# equivalent:

zsbom scan --file requirements.txt --explain
```

Key flags

- `--file <path>`: input manifest (e.g., `requirements.txt`) **or** CycloneDX SBOM.
- `--policy risk_config.yaml`: override policy file (defaults to repository policy).
- `--explain`: print per-dimension breakdown in the console.
- `--out <dir|file>`: write JSON outputs to a path (default: working directory).



Appendix Figure 1 : CLI Execution Screenshot

E.3 Processing workflow (deterministic, four steps)

1. **Validation & dependency resolution.** Parse manifest/SBOM, resolve dependencies, validate schema. Export `validation_report.json`.
2. **Metadata harvesting.** Query public sources (e.g., PyPI for registry metadata; OSV.dev for CVEs; CWE catalogue for weakness families). Responses are cached to reduce I/O variability and rate-limit sensitivity.
3. **Risk assessment.** Plugins compute raw 0–10 signals per dimension, which are **normalised to 0–1** and passed to the weighted aggregation from §4.5 to produce a composite score R(v)R(v)R(v).
4. **Classification & export.** Assign **Low/Medium/High** labels. Emit structured artefacts:

- `risk_report.json` — per-package dimension scores, composite score, final label.
- `transitive_analysis.json` — dependency graph and risk propagation.
- `sbom.json` — CycloneDX export of the analysed set.

(For snapshot pinning, hashes, and replay instructions, see **Appendix G**.)

E.4 Example run (illustrative)

A representative run on a `requirements.txt` with **67** packages (including transitives) resolved dependencies, retrieved CVE data from OSV.dev, and mapped CWEs. The tool flagged `langchain-community` as **High**; example dimension scores included **Mismatch = 10/10** and **Typosquatting =**

7/10. The overall distribution was **1 High**, **45 Medium**, **21 Low**. Outputs comprised `validation_report.json`, `risk_report.json`, `transitive_analysis.json`, and `sbom.json`.

E.5 DevSecOps integration & policy-as-code

ZSBOM is designed to run as a **pre-installation gate** in CI/CD. JSON outputs enable automated enforcement of policy rules.

Typical pipeline flow

1. Manifest submission (requirements/SBOM) →
2. CI runner invokes `zsbom` with a fixed policy →
3. Results parsed from `risk_report.json` →
4. Enforcement (block High; warn/approve Medium; allow Low).

Example (GitHub Actions)

```
- name: Run ZSBOM
  run: zsbom --file requirements.txt --explain
- name: Enforce Policy
  run: |
    python scripts/enforce.py risk_report.json
```

`enforce.py` blocks on any **High**, warns on **Medium**, and passes on **Low**. Equivalent checks can be implemented in GitLab CI, Jenkins, or Azure DevOps.

E.6 Operational notes

- **Auditability.** Policies are version-controlled; runs record effective configuration (weights, thresholds, enabled dimensions) and software versions.
- **Performance.** Metadata queries and resolution complete in seconds for typical manifests; caching further reduces re-scan time.
- **Scope.** The tool does **not** install or execute third-party code; it operates pre-installation and uses registry/SBOM metadata only.

E.7 Summary

The CLI operationalises the scoring model from §4.5 into a deterministic pipeline that yields human-readable summaries and structured artefacts for automation. Its design—minimal flags, configuration-driven defaults, and JSON outputs—supports adoption in everyday developer workflows while maintaining traceability.

Appendix F Pseudocodes

Package Abandonment Model

```
Input: t_last (timestamp of last release), t_ref (scoring reference time)
       Thresholds: τ_low = 6m, τ_med = 12m, τ_high = 24m
       Scorer parameters: w_MED, w_LOW, w_HIGH

Δt = t_ref - t_last

if Δt ≤ τ_low:
    W_abnd = 10.0
else if τ_low < Δt ≤ τ_med:
    W_abnd = w_MED
else if τ_med < Δt ≤ τ_high:
    W_abnd = w_LOW
else:
    W_abnd = w_HIGH

S_ABND = 1 - (W_abnd / 10)

Output: S_ABND ∈ [0,1]
```

CWE Coverage

```
Input: Set C_v of CWE identifiers for package version v
       Severity mapping σ(c) → {NONE, LOW, MEDIUM, HIGH}
       Scorer parameters: w_HIGH, w_MED, w_LOW

if C_v is empty:
    W_base = 10.0
else if any LOW and no MEDIUM/HIGH:
    W_base = w_LOW
else if any MEDIUM and no HIGH:
    W_base = w_MED
else if any HIGH:
    W_base = w_HIGH

P_c = min(2.0, 0.3 * max(0, |C_v| - 1))
h   = count of c in C_v where σ(c) = HIGH
P_h = min(1.5, 0.5 * h)

R_CWE = max(0, W_base - P_c - P_h)
S_CWE = 1 - (R_CWE / 10)
```

```
Output: S_CWE ∈ [0,1]
```

Example excerpt: Policy as a Code

```
weights:
  typosquat_heuristics: 0.15
  package_abandonment:  0.20
  known_cves:           0.30
  declared_vs_installed:0.15
  cwe_coverage:         0.20
thresholds:
  low: 0.33
  high: 0.66
gates:
  fail_on_high_direct: true
```

Appendix G: Reproducibility and Ethics

This appendix explains how the artefact and findings can be **reproduced and audited**, and sets out the **ethical stance** of the study. It complements **Appendix E (Reproducibility)**, which lists the concrete artefacts, hashes and replay steps for the **June 2025** snapshot and the **frozen policy** (validated in runs **21–22**; see §4.8.1).

G.1 Software artefact and release provenance

- **Public repository:** https://github.com/ZerberusAI/ZSBOM
- **Release used for the reported runs: v0.9** *(pre-release)* — commit **4a4682a** — released **05 Sep 2025 13:15** — **GPG-signed** (key ID **B5690EEEBB952194**).
- The release notes capture the features exercised in evaluation (OSV.dev integration, CWE check, transitive analysis, risk-scoring framework, typosquatting heuristics, declared-vs-installed check, CycloneDX export).
- **Data state:** all results use the **June 2025** pinned dataset (requirements + advisory feeds), as detailed in Appendix E.

G.2 Reproducibility commitments

- **Publication & tagging.** Source, policy and documentation are public, with a tag (**v0.9**) matching the experiments.
- **Policy-as-code.** `risk_config.yaml` externalises **weights**, **thresholds** and **gates**; the configuration is validated on load so malformed settings fail fast.
- **Deterministic preprocessing.** Identifiers are normalised; timestamps converted to ISO-8601 UTC; nulls are explicit; duplicate records from pagination/snapshots are removed; where available, checksums are verified before analysis.
- **Time-bounded data.** The evaluation window is fixed to **June 2025**. Package lists, feed dumps and policy hashes bind outputs to exact inputs; Appendix E records these with SHA-256 hashes.
- **Verifiable outputs.** Each run emits a console summary and structured files (per-dimension scores, composite score R(v)R(v)R(v), label, dependency/transitive view). These allow external recomputation and tracing of decisions.

G.3 Auditability and transparency

- Each run records the **effective configuration** (active dimensions, weights, thresholds), software versions, external endpoints contacted and the **time windows** of those queries.
- Per-package outputs expose the **contribution of each dimension** to the overall score, making decisions inspectable.
- Appendix E provides an artefact inventory and **re-execution recipe**; determinism is evidenced by matching hashes for runs **21** and **22**.

G.4 Ethical considerations: scope, risk and safeguards

- **Scope of data.** No human subjects are involved. Processing is limited to **public registry and repository metadata**. Maintainer names, where present in registries, are treated as public facts and are not combined with other personal data.
- **Operational safety.** The tool operates **pre-installation** and does **not** install or execute third-party code, reducing exposure to malicious artefacts and aligning with safe evaluation practices.
- **Dual-use awareness.** Publishing the scoring logic supports defenders but might inform adversaries. The dissertation provides design detail sufficient for scientific review **without** exploit proofs or redistribution of malicious payloads. The repository contains metadata, identifiers and code to reconstruct datasets from public sources only.
- **Responsible disclosure.** If scanning identifies suspicious packages, the intended practice is to notify registry maintainers or appropriate security contacts **before** any public disclosure, following platform policies.
- **Dataset handling.** No malware binaries are redistributed. Scripts fetch data from public sources; the corpus uses labels derived from public incident reports and measurements. Some label noise is possible; results and limitations acknowledge this and emphasise that CVE counts require context and are subject to known scanner/database limits.
- **Intended use.** The artefact is designed to **triage** risk early and complement deeper code or runtime analysis, not replace it.

G.5 How this enables replication and audit

Together, the release provenance (G.1), reproducibility practices (G.2), transparent logging (G.3) and ethical safeguards (G.4) allow an external reviewer to: (i) obtain the exact code state (**v0.9**), (ii)

reconstruct the **June 2025** snapshot, (iii) run the frozen policy from **runs 21–22**, and (iv) verify outputs against the published hashes in **Appendix E**.