MavCrawl: A Comprehensive Dataset of Maven Libraries with Metadata and Their Dependencies

Abhinav Jamwal
Indian Institute of Technology Roorkee
Uttarakhand, India
abhinav_j@cs.iitr.ac.in

Parvathi Nair
Indian Institute of Technology Roorkee
Roorkee, India
parvathi_n@cs.iitr.ac.in

Siya Arora
Indian Institute of Technology Roorkee
Roorkee, India
siya_a@cs.iitr.ac.in

Manish Agrawal
Indian Institute of Technology Roorkee
Roorkee, India
m_agrawal@cs.iitr.ac.in

Sandeep Kumar
Indian Institute of Technology Roorkee
Roorkee, India
sandeep.garg@cs.iitr.ac.in

Abstract

Modern software engineering depends on accurate dependency information for software maintenance, software quality assessment, and software supply chain auditing. In Java projects, dependencies are often not fully specified as direct entries in a single build file. They must be resolved through parent POM inheritance, placeholder substitution, and centralized configuration rules such as `dependencyManagement`. As a result, datasets that record only declared dependencies or omit provenance and module context do not support large-scale and reproducible analysis. We present *Maven Dependency Crawler (MavCrawl)*, a tool-and-dataset contribution that reconstructs build-relevant dependency relations by resolving Maven configuration across nested POM files and extracting artifact metadata (e.g., publication timestamps, JAR size, descriptions, SCM URLs, and parent-child hierarchies). The resulting dataset is released in a structured format with a clear schema and completeness statistics, which enables users to filter records based on metadata availability and supports replicable empirical studies. MavCrawl enables downstream evaluation and modeling tasks in software engineering, including learning-to-rank or classification settings for dependency recommendation and risk prediction, and supports fair comparisons by providing resolution-aware ground truth.
Dataset (Zenodo DOI): https://zenodo.org/records/19678811
Source code (GitHub): https://github.com/arorasiya008/Maven-Dependency-Crawler

CCS Concepts

• **Software and its engineering → Software libraries and repositories**; **Software maintenance tools**.

EASE '26, Glasgow, United Kingdom

ACM ISBN 979-8-4007-2348-3/26/06
https://doi.org/10.1145/3816483.3816599

Keywords

Maven, Dependency Management, Metadata Extraction, Transitive Dependencies, Software Reuse, POM Analysis, Software Supply Chain, Software Ecosystems

ACM Reference Format:
Abhinav Jamwal, Parvathi Nair, Siya Arora, Manish Agrawal, and Sandeep Kumar. 2026. MavCrawl: A Comprehensive Dataset of Maven Libraries with Metadata and Their Dependencies. In *International Conference on Evaluation and Assessment in Software Engineering (EASE '26), June 09–12, 2026, Glasgow, United Kingdom.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3816483.3816599

1 Introduction

Software reuse is a cornerstone of modern software engineering [12, 15]. In contemporary practice, reuse is realized largely through third-party libraries, which are integrated and updated through package managers and build systems. Within the Java ecosystem, *Maven*[1] has become the de facto standard for dependency management: developers declare required libraries in `pom.xml` files, and Maven resolves both direct and transitive dependencies to construct a buildable classpath. This workflow reduces the integration effort and enables rapid development, but it also makes projects deeply dependent on the structure and evolution of the surrounding dependency ecosystem.

This convenience comes with substantial analytical and operational challenges. Maven dependency graphs can be large and highly nested due to transitive resolution, multi-module hierarchies, and extensive reuse of parent POMs. In addition, POM files frequently contain property placeholders (e.g. `${project.version}$`) whose values are defined across parent–child chains and may require recursive substitution before dependencies and metadata can be interpreted correctly. These characteristics complicate fundamental tasks such as measuring dependency evolution, diagnosing version conflicts, and auditing software supply chains [1, 9, 13]. Although prior work has studied related problems, including dependency bloat [22], Android library detection [20], and research tools to query Maven artifacts (e.g. MARIN) [5], the community still lacks a curated, reusable, large-scale dataset that combines (i)

[1] https://maven.apache.org/

rich artifact metadata with (ii) dependency relationships extracted under realistic Maven resolution semantics. In particular, datasets built from raw POM parsing often miss inherited declarations or leave placeholders unresolved, whereas query-oriented systems typically do not provide a persistent corpus that can be independently analyzed, replicated, and extended.

To address this gap, we present *Maven Dependency Crawler (MavCrawl)*, an automated pipeline for constructing a high-fidelity Maven artifacts dataset from Maven Central[2] and three additional repositories: Google[3], Atlassian[4] and Cloudera[5]. MavCrawl extracts artifact metadata (e.g., update timestamps, JAR sizes, descriptions, SCM URLs, and module hierarchy) and resolves dependency information while systematically handling recursive property expansion across nested POM structures and incomplete metadata. Rather than treating dependency information as a byproduct of ad-hoc queries, we decouple dataset construction from downstream analyses and release both the crawler and a structured corpus intended to support empirical research, tool evaluation, and supply-chain investigations.

The resulting dataset supports a broad range of studies, including dependency evolution, conflict analysis, and security assessments along dependency chains, as well as downstream tasks such as automated library recommendation. This paper makes the following contributions:

- **Automated extraction and resolution pipeline:** We design a scalable, multi-repository crawling and parsing pipeline that robustly reconstructs artifact metadata and dependency links under Maven-specific challenges such as parent inheritance and recursive property substitution.
- **Curated open dataset of Maven artifacts:** We release a publicly available dataset of Java artifacts enriched with metadata (timestamps, sizes, SCM URLs, and module relations), direct dependency links, and repository provenance to enable reproducible ecosystem-level analyses.
- **Empirical utility and guidance:** We outline how the dataset can be used to support dependency analysis, software supply-chain security studies, and build automation research, and document practical limitations (e.g., incomplete metadata and resolution timeouts) that affect downstream validity.

2 Related Work

MavCrawl is positioned as a data set for software engineering research on dependency management, ecosystem analysis, and software supply chain studies in the Java and Maven setting. Previous work provides valuable building blocks but is fragmented across datasets that target specific analyses, graph snapshots that focus on a single repository, and query interfaces that do not yield a stable, reusable corpus. Table 1 summarizes how these works differ in scope and in the dependency semantics they support.

Maven centric datasets and artifact corpora. Early work demonstrated the value of curating Maven Central into research grade datasets. Raemaekers et al. [19] released the *Maven Repository Dataset of Metrics, Changes, and Dependencies (MDD)*. It consolidates Maven Central artifacts and provides the dataset through multiple storage backends, including relational and graph oriented exports. This enables studies on artifact evolution, metrics, and dependencies. Mitropoulos et al. [16] constructed the *Maven Ecosystem Bug Catalog* by statically analyzing a large portion of Maven Central with FindBugs. In addition to bug and metric output, their dataset stores per version metadata such as JAR size and POM derived dependencies. These datasets are influential, but they are not designed as a general purpose, resolution aware dependency substrate. In particular, they do not aim to expose Maven specific configuration mechanisms, such as recursive parent inheritance and property placeholder expansion, in a way that directly supports reproducible dependency tree reconstruction and cross study comparability.

Temporal and graph based representations of Maven dependencies. Benelallam et al. [2] introduced *The Maven Dependency Graph (MDG)*. It provides a temporal graph based representation of Maven Central, distributed as periodic snapshots, for longitudinal analysis of dependency networks. MDG is an important step toward ecosystem scale graph analytics. At the same time, the authors report representation gaps that can affect downstream uses, and the scope remains tied to artifacts present in Maven Central [2]. For dataset driven research, this leaves a practical gap. Many studies need both broader repository coverage and richer artifact provenance metadata so they can evaluate dependency phenomena outside a single repository boundary and do so with fewer reconstruction steps.

Querying infrastructures and Research-Centered Interfaces. MARIN [5] proposes a research centric interface to query software artifacts on Maven repositories. Its goal is to simplify access to package metadata, releases, and dependency related information through a unified API. This is useful for interactive querying and rapid prototyping. However, an interface does not substitute for a stable versioned dataset snapshot that can be archived, cited, and reused in replication packages. For dataset papers, that difference matters because repeatable evaluation often requires fixed inputs, explicit schemas, and reproducible extraction assumptions.

Empirical studies showing the need for precise dependency trees. Several empirical studies in the Maven ecosystem show that dependency semantics is not a secondary concern. Soto Valero et al. [22] study bloated dependencies and introduce DEPCLEAN. Their results rely on broad dependency relationships and show that transitive dependencies dominate the Maven dependency structure. This underscores that analyses can become misleading when dependency trees are approximated or extracted without handling inheritance and configuration effects. Beyond Maven, ecosystem security studies show the same methodological risk from a different angle. Liu et al. [13] demonstrate in the NPM ecosystem that reachability style reasoning which ignores official resolution rules can yield incorrect transitive dependencies. Maven and NPM use different resolution semantics, but the implication carries over. Dataset builders should make dependency extraction rules explicit, testable, and reproducible so that downstream results are interpretable and comparable.

Cross ecosystem dependency network analysis. A complementary line of work studies dependency networks across package ecosystems to understand structural properties and evolution. Kikas

[2]https://mvnrepository.com/repos/central
[3]https://maven.google.com/
[4]https://packages.atlassian.com/maven/public/
[5]https://repository.cloudera.com/artifactory/cloudera-repos/

Table 1: Mechanism oriented comparison of Maven related datasets and resources, and the gap addressed by MavCrawl.

Work	Primary artifact	Ecosystem scope	Dependency information	Emphasis / limitations (as reported)
MDD [19]	Dataset of metrics, changes, and dependencies with multiple database exports	Maven Central	Dependencies and evolution oriented metadata	Designed for broad ecosystem studies. It is not centered on fully resolved, rule explicit dependency trees.
Bug Catalog [16]	FindBugs based bug and metric dataset with per version metadata (e.g., size, dependencies)	Maven Central	Dependencies recorded as metadata alongside static analysis outputs	Targets bug and quality analysis. Dependency resolution is not the central contribution.
MDG [2]	Temporal dependency graph snapshots (e.g., Neo4j)	Maven Central	Graph representation of dependency relations over time	Reports representation gaps for some dependency aspects and practical scope constraints.
MARIN [5]	Research centric querying interface or API	Maven repositories (interface driven)	Dependency related retrieval and version or range utilities	Interface first. It does not provide a curated, downloadable dataset snapshot with standardized resolved metadata.
DEPCLEAN study [22]	Empirical dataset and tool labels for bloated dependencies	Maven Central (sampled artifacts)	Large set of dependency relationships including transitive effects	Problem driven debloating. Dataset scope is tied to the study sample and labels.
AndroLibZoo [20]	Android third party library whitelist derived via dependency mining	Maven and Google plus OSS Android projects	Uses dependency trees during extraction. Dataset is a library list	Not a general Maven artifact dataset. It targets Android library identification.
MavCrawl	**Curated multi repository dataset with enriched metadata and resolved dependencies**	**Maven Central plus additional repositories**	**Direct dependencies with explicit handling of inheritance and placeholders. Transitive dependencies are reconstructable**	**Dataset first design for replication, benchmarking, and downstream software engineering tasks**

et al. [10] analyze the structure and evolution of package dependency networks and motivate ecosystem level measurements as a lens on software reuse and risk. Decan et al. [4] compare the evolution of dependency networks across seven packaging ecosystems using libraries.io data, highlighting the growing emphasis on large scale, comparable ecosystem datasets. These studies also make an important point for Maven centric research. Cross ecosystem conclusions are only as reliable as the underlying dependency data quality. That strengthens the motivation for a Maven dataset that standardizes dependency semantics and provenance metadata in a form that is straightforward to reuse.

Dependency based datasets beyond Maven and library identification. Dependency mining is increasingly used to build datasets for adjacent software engineering tasks. Samhi et al. [20] introduce AndroLibZoo, a data set of third party libraries for Android analysis that is produced by dependency mining and designed to stay up to date. Although AndroLibZoo targets Android library whitelisting rather than Maven dependency graph reconstruction, it reinforces a key lesson from the dataset. Dependency based construction can improve precision, but scope, coverage boundaries, and extraction semantics must be documented so that others can reproduce and extend the dataset.

Taken together, existing work either focuses on Maven Central only, targets analysis specific outputs, or provides query time access rather than an archival dataset. MavCrawl addresses this gap by releasing a curated multi repository corpus that emphasizes rich artifact metadata and provenance, together with dependency information extracted under Maven specific configuration mechanisms, including parent inheritance and placeholder resolution. This design reduces the burden on researchers and tool builders who otherwise need to reimplement resolution logic and metadata aggregation before they can evaluate dependency evolution, ecosystem risks, or supply chain questions at scale.

3 Motivation

Third-party libraries are central to software reuse, but they also increase the number of external dependencies that a project must manage, which can affect maintenance effort, analysis reliability, and software supply chain security [1, 17]. Many software engineering tasks depend on accurate dependency information and artifact provenance, including identifying risky imports, studying dependency evolution, diagnosing conflicts, and evaluating ecosystem level quality trends. These tasks become difficult when dependency relations cannot be reconstructed consistently or when basic artifact metadata such as release time, repository origin, or source code location are missing. This challenge is especially visible in cross ecosystem studies. Decan et al. report that Maven dependency information is difficult to obtain reliably from common aggregation sources, which can prevent Maven from being included in comparative dependency analyses [4]. At the same time, empirical studies show that transitive and inherited dependencies have a dominant effect on what projects actually ship, which means that the construction of the data sets must go beyond the declared dependencies at the surface level [22].

Existing resources only partially address these needs. AndroLibZoo [20] is a well engineered dataset for Android library identification, where the main output is a whitelist of library package names intended to separate third party code from app code. That design is appropriate for program analysis, but it limits reuse in studies that require artifact level metadata and dependency semantics. For example, it cannot support analyses that require release timestamps, provenance signals, or repository-sequenced evolution because these attributes are outside its goal and schema. In parallel, MARIN [5] provides a research centric interface for querying Maven artifacts. It improves access to artifact and dependency related information, but it does not release a stable, analysis ready dataset snapshot. For replication and benchmarking, relying on query time retrieval

can introduce hidden variability through repository changes, configuration choices, and query specific extraction logic. In addition, Maven Central alone does not represent the full diversity of the JVM ecosystem. Large repositories maintained by major vendors and platforms also host widely used artifacts, and excluding them can bias ecosystem analyses toward a single distribution channel [10].

These limitations motivate the need for a curated dataset that is both reusable and faithful to Maven specific dependency semantics. MavCrawl addresses this need by constructing a unified corpus across Maven Central[2], Google[3], Atlassian[4], and Cloudera[5]. It enriches artifacts with metadata needed for provenance and maintenance analyses, including update timestamps, JAR sizes, parent child hierarchy relations, and source repository URLs. It also extracts dependency information using explicit resolution steps so that inherited configuration and placeholder based declarations are handled consistently. By releasing a downloadable dataset snapshot rather than only an interface or a task specific whitelist, MavCrawl lowers the barrier for reproducible studies on dependency evolution, bloat, and vulnerability exposure, and it enables fairer benchmarking of tools that operate on Maven dependency ecosystems.

4 Application

Modern software development relies on large dependency ecosystems such as Maven. These ecosystems make integration convenient, but they also increase maintenance effort and widen the software supply chain attack surface [1]. As a result, research and tooling increasingly need rich and structured dependency metadata to support tasks such as static analysis, vulnerability assessment, and ecosystem level measurement. MavCrawl is designed to provide this substrate by coupling dependency links with artifact level provenance and maintenance signals in a reusable dataset.

The following outlines the key metadata attributes captured in this dataset and why they are useful.

- **JAR size:** JAR size is a practical signal for build and distribution analysis. Large artifacts can increase build time and deployment cost. They can also indicate unnecessary packaging, dependency bloat, or unexpected payload.
- **Last modified timestamp:** Update time provides a lightweight proxy for maintenance recency. It can support studies of update lag and library freshness and can help flag dependencies that remain outdated for long periods.
- **Description:** Descriptions provide a short functional summary that supports search, classification, and manual inspection. They are useful when developers or researchers need to understand the role of an unfamiliar dependency.
- **Source code URL:** Links to source repositories support provenance tracking and enable repository level mining, such as tracing changes, auditing history, and connecting artifacts to maintainers and development activity.
- **Direct dependencies:** Direct dependencies expose the declared functional requirements of an artifact. They enable compatibility checks, dependency context modeling, and analysis of dependency usage patterns. They also provide the basis for reconstructing transitive dependency chains through graph traversal.
- **Parent module:** Parent and module relations capture shared configuration and inheritance structure. This information supports studies of Maven project organization and enables analyses that account for inherited dependency declarations and shared build settings.

These metadata attributes support a variety of practical applications.

- **Software Supply Chain Security:** Combine provenance signals with dependency chains to study update lag, identify potentially abandoned artifacts, and support vulnerability assessments that require understanding which components are reachable through dependencies.
- **Library Recommendation and Dependency Prediction:** Use dependency co-use patterns and artifact attributes to recommend libraries that match a project context. Metadata such as recency and provenance can also be used to recommend actively maintained alternatives.
- **Empirical Studies on Software Ecosystem Evolution:** Track adoption and decline over time using timestamps and repository provenance. Analyze how the depth of dependency, churn, and the concentration of the ecosystem change and compare the patterns between repositories that represent different segments of the JVM ecosystem.
- **Automated Build Configuration and Optimization:** Study configuration reuse through parent module structure and dependency declarations. Support tools that diagnose configuration anomalies, detect overly complex dependency structures, and suggest cleaner dependency sets.
- **Software Quality Assessment:** Relate the dependency structure and maintenance signals, such as recency, dependency depth, and module structure, to quality and maintainability outcomes. This supports empirical models and assessments of software health.

5 Dataset Construction and Description

We construct the MavCrawl dataset using *MavCrawl* and collect artifacts from four complementary Maven repositories. Maven Central provides broad coverage of open source Java libraries, while Atlassian, Cloudera, and Google contribute vendor maintained and domain specific artifacts. This combination supports analyses that need both ecosystem breadth and repository specific context. Figure **??** provides a visual overview of the workflow and Algorithm 1 summarizes the pipeline and the semantics used to extract `direct_dependencies`.

5.1 Artifact Discovery

For Maven Central, Atlassian, and Cloudera, we discover artifacts by traversing repository directory structures and parsing HTML listings. For each discovered artifact, we enumerate all available versions, using the latest version as the initial seed for this dataset release. During dependency extraction, when we observe referenced dependency coordinates (including non-latest versions), we enqueue and process them to materialize dependency edges. This scope choice reduces redundant processing across versions and focuses the dataset on the dependency landscape that developers are most likely to use.

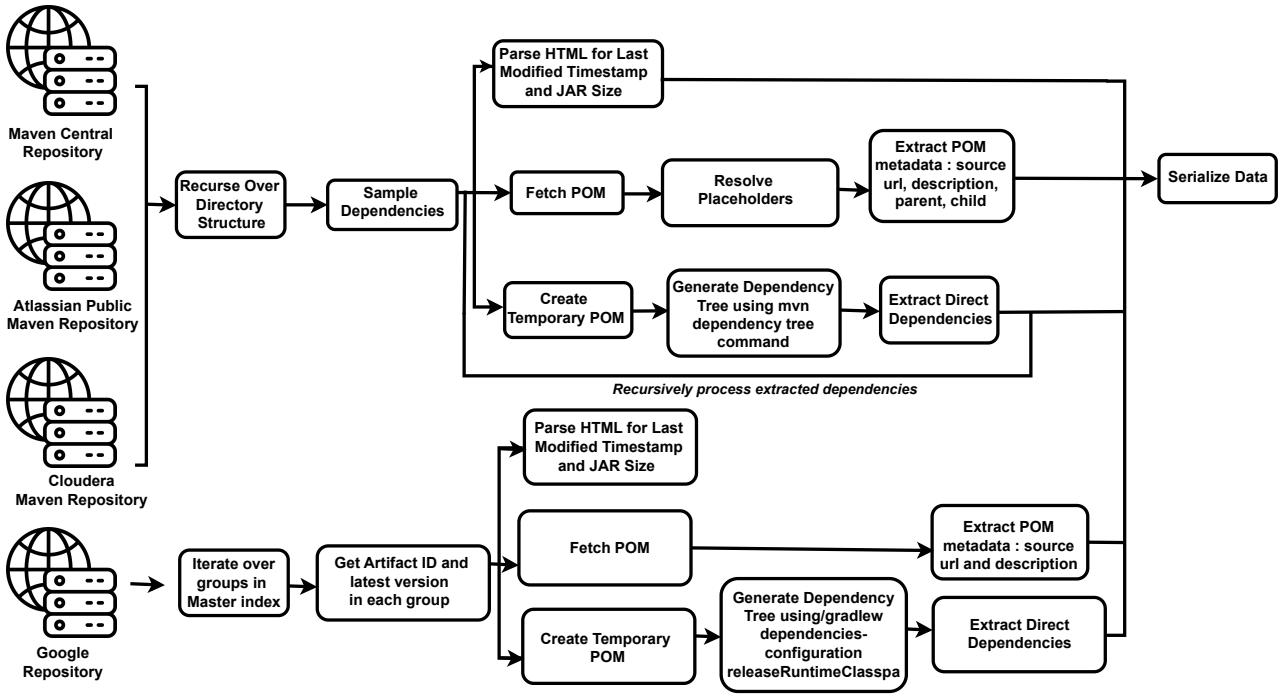

Figure 1: Dataset Construction Methodology.

To manage repository scale while preserving diversity, we apply random sampling within large `groupId` namespaces for Maven Central and Cloudera. The goal is to avoid overrepresenting a small set of popular namespaces and to include artifacts from a wider range of domains.

Google Maven supports index driven discovery and does not require HTML parsing. We iterate `master-index.xml` to enumerate `groupId` values, then read each `group-index.xml` to obtain `artifactId` values and versions. As in other repositories, we select the latest version for processing.

5.2 Metadata Parsing and Property Resolution

For each selected artifact, we construct the corresponding directory URL and retrieve the repository listing to collect basic file metadata, including update timestamps and binary size when available. We then download and parse the artifact POM to extract the fields required for provenance and hierarchy analysis, including description, SCM URL, and parent module.

Maven POM files commonly use placeholders such as `${project.version}` and custom properties. We resolve placeholders recursively using the artifact POM and its parent POM chain. When a property is defined at multiple levels, the child definition takes precedence. To avoid non-termination and to make failures explicit, we cap recursive resolution to a fixed depth and store an explicit unresolved marker when expansion is not possible. This step is applied before persisting the record, so that the stored metadata reflects expanded values whenever possible.

5.3 Direct Dependency Resolution

To extract direct dependencies for artifacts from Maven Central, Atlassian, and Cloudera, we rely on resolver output so that inheritance and `dependencyManagement` effects are reflected in the extracted edges. We use an isolated resolution procedure based on Maven tooling.

- For each target artifact, we generate a temporary `pom.xml` that declares the target artifact as a dependency.
- We execute `mvn dependency:tree -Ddepth=2` on the temporary project to obtain a resolver-produced dependency tree under a default scope configuration, capturing dependencies as resolved in the standard build context, primarily including compile and runtime scopes.
- We parse the output and store as `direct_dependencies` the immediate children of the target artifact node in the printed tree (i.e., depth-1 edges below the target), rather than the temporary project's declared dependency list.

Resolution semantics (applies to all extracted edges unless stated otherwise): (i) We perform dependency extraction using the resolver's default scope configuration, which includes dependencies from all standard scopes; (ii) exclusions are honored as reported by the resolver; (iii) optional and profile-activated dependencies are not forced on and are included only if they appear under the chosen resolution configuration.

During data set construction, newly observed dependency coordinates are added to the processing queue so that the data set becomes progressively more complete with respect to reachable artifacts within the snapshot of the repository. We store only direct

Algorithm 1 MavCrawl pipeline with resolution-aware direct dependencies

Require: Repositories $\mathcal{R}$, seed artifacts $\mathcal{S}$ (latest versions), timeout T
Ensure: Dataset $\mathcal{D}$ with metadata, hierarchy, and `direct_dependencies`
$Q \leftarrow \mathcal{S}$; $Seen \leftarrow \emptyset$; $\mathcal{D} \leftarrow \emptyset$
while $Q \neq \emptyset$ **do**
  $a \leftarrow$ pop(Q)
  if $a \in Seen$ **then continue**
  end if
  $Seen \leftarrow Seen \cup \{a\}$
  $(pom, meta_{repo}) \leftarrow$ fetch POM and repository metadata within T
  if POM not found **then**
    continue
  end if
  $props \leftarrow$ recursively collect and resolve properties from POM and parent chain
  $deps \leftarrow$ resolve direct dependencies via build tool configured for a and extract depth-1 dependencies from resulting tree
  if dependency resolution fails **then**
    continue
  end if
  $(desc, scm, parent, children) \leftarrow$ parse POM.
  Property Resolution:
  Resolve placeholders in $(desc, scm, parent, children, deps)$ using $props$
  if any unresolved placeholder **then**
    $extraction\_status \leftarrow Unresolved\ Properties$
  else
    $extraction\_status \leftarrow Success$
  end if
  Hierarchy update:
  if $parent \neq Unknown$ **then**
    if $parent \in \mathcal{D}$ **then**
      $parent.child\_modules \leftarrow parent.child\_modules \cup \{a\}$
    else
      store placeholder record($parent$, $null$, $null$, $null$, $null$, $\{a\}$, [], $Unprocessed\ Parent$) in $\mathcal{D}$
    end if
  end if
  store record(a, $meta_{repo}$, $desc$, scm, $parent$, $children$, $deps$, $extraction\_status$) in $\mathcal{D}$
  for all $d \in deps$ **do**
    if $d \notin D$ **then**
      $Q \leftarrow$ enqueue(Q, d)
    end if
  end for
end while
return $\mathcal{D}$

dependencies. Transitive dependencies can be reconstructed by recursively traversing direct dependency links in the data set.

For Google Maven, artifacts often include Android libraries packaged as AARs and standard JARs. For these cases, we use Gradle based resolution in a temporary project and extract first level dependencies in a similar manner. Concretely, we run `./gradlew dependencies` across multiple configurations, including:

- `releaseCompileClasspath`
- `releaseRuntimeClasspath`
- `debugCompileClasspath`
- `debugRuntimeClasspath`
- `testCompileClasspath`
- `testRuntimeClasspath`

using a temporary `build.gradle` file that includes the target artifact.

5.4 Module Processing

We infer multi module structure incrementally using the `parent` and `modules` fields in the POM. When a child module is processed before its parent, we create a placeholder entry for the parent coordinate and update the parent's `child_modules` list to include the child. When the parent is later processed, we populate the placeholder with resolved metadata. This strategy supports reconstruction of project hierarchies without requiring a strict processing order.

5.5 Deduplication and Scalability

Each artifact record is keyed by `groupId:artifactId:version`, which serves as the primary key within each repository-specific MongoDB collection and prevents duplicate entries during extraction. Placeholder records created during module processing share the same key and are merged when the corresponding artifact is fully processed.

When a property cannot be resolved during metadata extraction, unresolved placeholders are retained in the stored values, allowing downstream analyses to distinguish incomplete resolution from missing metadata.

To improve throughput, the pipeline supports parallel processing with per-artifact execution limits. Each record includes an extraction_status field (e.g., success, unresolved properties, unprocessed parent) to capture the outcome of extraction. Dependency resolution is performed using repository-appropriate tooling, namely Maven for Maven Central, Atlassian, and Cloudera, and Gradle for Google Maven.

Repository-specific discovery logic is modularized, with HTML traversal for Maven Central, Atlassian, and Cloudera, and index-based discovery for Google Maven. Artifacts originating from different repositories are preserved separately, with the combination of `groupId:artifactId:version` and `origin_repository` effectively acting as a composite identifier in the merged dataset.

These design choices allow the pipeline to scale to large repositories and make it straightforward to extend MavCrawl to additional Maven-compatible repositories in future releases.

5.6 Repositories

The data set spans the following sources.

- **Maven Central** [2]**:** The largest public JVM repository, hosting approximately 19.3 million artifacts as of February 2026.
- **Google Maven Repository** [3]**:** A primary source for Android and Google maintained libraries.
- **Atlassian Public Repository** [4]**:** Artifacts used by Atlassian products, plugins, and platform components.
- **Cloudera Repository** [5]**:** Dependencies related to Cloudera Runtime and associated distributions.

5.7 Schema and recorded attributes

Each data set record corresponds to one artifact version, identified by the Maven coordinate `groupId:artifactId:version`. We store this coordinate as `_id`. The remaining fields capture artifact metadata, dependency information, and module structure to support filtering, aggregation, and cross repository analyses. Table 2 summarizes the MongoDB schema used to store the data set, and Table 3 shows representative entries, including their `origin_repository`.

For each artifact, we record the following attributes.

- **Identifiers:** `groupId`, `artifactId`, and `version` uniquely identify each entry within a repository. The concatenated form `groupId:artifactId:version` serves as the artifact identifier. As the same artifact may be resolved from multiple

Table 2: MongoDB Dataset Schema for Maven Dependencies.

Field	Description	Data Type
`_id`	Unique identifier in the format `groupId:artifactId:version`	String
`last_modified`	Last modified timestamp of the artifact in the Maven repository	String
`jar_size`	Size of the JAR or AAR file in bytes (as string)	String
`description`	Project description extracted from the POM file	String
`direct_dependencies`	List of direct dependencies of the artifact	Array of Strings
`source_code_url`	URL of the source code repository	String
`parent_module`	Parent POM module if it exists	String
`child_modules`	List of child modules derived from this artifact	Array of Strings
`origin_repository`	Source repositories from which the artifact was resolved (e.g., Maven Central, Atlassian, Cloudera, Google)	String
`extraction_status`	Indicates the outcome of the metadata and dependency extraction process (e.g., success, unresolved properties, unprocessed parent).	String

repositories, records are uniquely identified by the combination of _id and origin_repository, forming a composite key.

- **JAR size:** Stores the size of the published artifact in bytes when available. This field supports build and distribution analyses and helps identify unusually large artifacts.
- **Last modified timestamp:** Records the most recent update time reported by the hosting repository. This signal supports studies of update lag and maintenance recency.
- **Description:** Captures the artifact description from the POM when present. This supports search and lightweight functional categorization.
- **Source code URL:** Records the URL of the version controlled source repository when it can be extracted from SCM-related POM fields. This enables provenance tracking and repository level mining.
- **Parent and children modules:** Encodes multimodule structure through `parent_module` and `child_modules`. This supports analyses that account for the inherited configuration and modular project organization.
- **Direct dependencies:** Stores direct dependencies as a list, where each dependency is represented in the form `groupId:artifa` The list includes dependencies inherited from the parent when present. Transitive dependencies are not stored explicitly, but can be derived by recursively traversing direct dependency links.
- **Origin repository:** Records the repository from which the artifact was resolved (e.g., Maven Central, Atlassian, Cloudera, Google). This attribute enables studies of ecosystem fragmentation, repository usage patterns, and artifact provenance across different distribution channels.
- **Extraction status:** Indicates the outcome of the metadata and dependency extraction process (e.g., success, unresolved properties, unprocessed parent). This field supports filtering and quality assessment of the extracted dataset.

5.8 Statistical overview

The dataset contains 68,013 unique artifact coordinates (`_id` =groupId :artifactId:version) collected from four repositories after cross-repository deduplication. Combining community-driven and vendor-maintained sources supports analyses of dependency usage, metadata availability, and repository-specific software supply-chain practices. Unless stated otherwise, dataset-wide statistics are computed over unique `_id` records; repository-level statistics are computed by filtering on `origin_repository`.

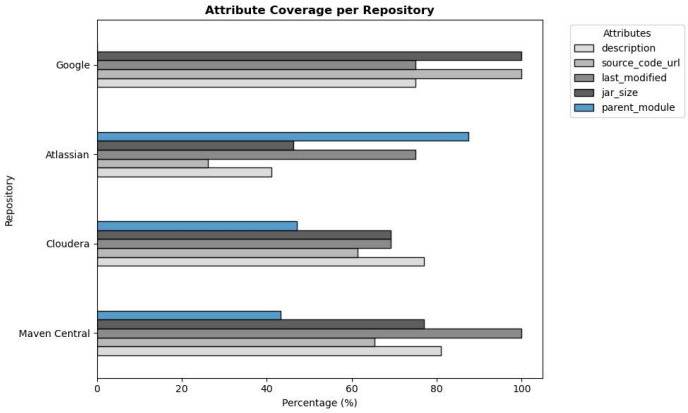

Figure 2: Overview of the attribute coverage of the *MavCrawl* dataset.

Figure 2 reports attribute coverage by repository as the proportion of artifacts with non-missing values for each attribute (computed within each `origin_repository` subset). Figure 3 shows the distribution of the number of direct dependencies per artifact using a box-and-whisker plot, including the median, interquartile range, and outliers. Table 4 lists the ten most frequent `groupId` values appearing as direct dependencies, highlighting widely used libraries and prominent maintainers in this snapshot.

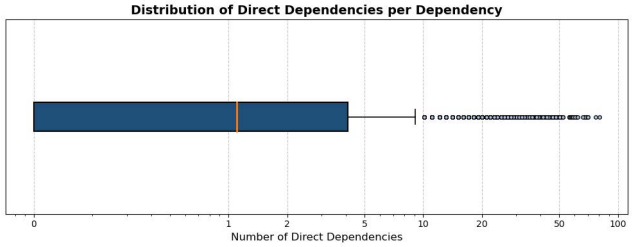

Figure 3: Distribution of direct dependencies per artifact, showing median, interquartile range, and outliers.

6 Limitations

Although the data set construction pipeline was designed to balance diversity, completeness, and scalability, several limitations remain that can affect how the data set should be used and interpreted.

- **Incomplete or inconsistent metadata.** Some artifacts expose partial metadata in their POM files. Fields such as description and SCM information may be missing, and some

Table 3: Sample Entries from Maven Dependencies Dataset.

_id	Last_modified	jar_size	Description	direct_dependencies	source_code_url	parent_module	child_modules
com.fasterxml.jackson.module:jackson-modules-java8:2.19.1	2025-06-14 01:34	Unknown	Parent pom for Jackson modules needed to support Java 8 features and types	[com.fasterxml.jackson.core:jackson-core:2.19.1:compile, com.fasterxml.jackson.core:jackson-databind:2.19.1:compile]	http://github.com/FasterXML/jackson-modules-java8	com.fasterxml.jackson:jackson-base:2.19.1	[com.fasterxml.jackson.datatype:jackson-datatype-jsr310:2.19.1, com.fasterxml.jackson.datatype:jackson-datatype-jdk8:2.19.1, com.fasterxml.jackson.module:datatypes:2.19.1]
com.azure:azure-ai-openai-assistants:1.0.0-beta.5	2025-02-21	401437	Microsoft Azure OpenAI Assistants client library.	[azure-json, azure-core, azure-core-http-netty]	github.com/Azure/	com.azure:azure-client-sdk-parent:1.7.0	[]
org.springframework.data:spring-data-jpa-parent:3.5.0	2025-05-16 11:00	Unknown	Parent module for Spring Data JPA repositories.	[org.slf4j:slf4j-api:2.0.2:compile]	https://github.com/spring-projects/spring-data-jpa	org.springframework.data.build:spring-data-parent:3.5.0	[org.springframework.data:spring-data-jpa:3.5.0, org.springframework.data:spring-data-envers:3.5.0, org.springframework.data:spring-data-jpa-distribution:3.5.0]
org.jboss.logmanager:jboss-logmanager:3.1.2.Final	2025-02-17 22:06	808980	An implementation of java.util.logging.LogManager	[org.jboss.logging:jboss-logging:3.5.3.Final:compile, io.smallrye.common:smallrye-common-constraint:2.2.0:compile, jakarta.json:jakarta.json api:2.1.3:compile,org.eclipse.parsson:parsson:1.1.6 :compile]	https://github.com/jboss-logging/jboss-logmanager/tree/main/	org.jboss.logging:logging-parent:1.0.3.Final	[]
net.anotheria.portalkit:portalkit:4.2.2	2026-01-02 20:33	Unknown	PortalKIT	[net.anotheria:configureme:4.0.0 :compile, net.anotheria:distributeme-support:4.0.2:compile,org.apache.httpcomponents.core5:httpcore5:5.2.3: compile]	https://github.com/anotheria/portalkit.git	net.anotheria:parent:4.3	[net.anotheria.portalkit:pk-engines:4.2.2,net.anotheria.portalkit:apis:4.2.2]

Table 4: Top 10 `groupId` values occurring as direct dependencies in the dataset.

groupId	Artifact count
org.jetbrains.kotlin	14,727
org.jetbrains.kotlinx	10,068
org.springframework	9,152
org.slf4j	7,178
io.netty	7,061
org.scala-lang	6,361
com.fasterxml.jackson.core	4,656
io.ktor	4,547
org.jetbrains.compose.ui	3,809
org.springframework.boot	3,765

records contain nonstandard or inconsistent values across repositories. When parent POMs or referenced properties are unavailable, a subset of placeholders remains unresolved. We retain such artifacts to preserve coverage and mark missing or unresolved fields so that users can filter them when needed.

- **Dependency resolution timeouts.** The temporary POM-based resolution is performed using the `mvn dependency:tree` command for artifacts from Maven Central, Atlassian, and Cloudera repositories, while Gradle's dependency resolution mechanism is used for artifacts from Google repositories. To ensure scalability across a large number of artifacts, the resolution runs under fixed timeouts. Projects with complex dependency graphs or slow resolution, for example, due to downloading POM files and metadata from remote repositories, may exceed this time limit. In such cases, the process is terminated early, which can result in incomplete dependency information for those artifacts.
- **Repository coverage and representativeness.** The data set covers Maven Central, Google Maven, Atlassian, and Cloudera. This provides broad coverage of public and vendor maintained artifacts, but it does not include private repositories or organization specific artifact hosts. As a result, conclusions drawn from MavCrawl reflect the only public and widely distributed portion of the Maven ecosystem.
- **Dependency scope and build context.** We record direct dependencies under standard build configurations, including common Maven scopes such as `compile` and `runtime`. Dependencies introduced through runtime mechanisms, such as reflection based loading, external configuration, shading,

or custom build plugins, are not captured. Similarly, build profiles and environment specific dependency activation can lead to differences between the resolved dependencies in our pipeline and those in a specific project setup.

- **Property and placeholder resolution limits.** Our recursive placeholder resolution covers common Maven patterns and resolves most properties through the parent chain. However, unresolved placeholders can remain when parent artifacts cannot be retrieved, when metadata is inconsistent across repositories, or when properties rely on build time interpolation. These cases most often affect version fields and descriptive metadata, and they can also influence dependency coordinates in rare cases.
- **Module hierarchy inference.** Multi-module structure is inferred from `<parent>` and `<modules>` declarations in POM files. When these declarations are missing, inconsistent, or distributed across repositories, reconstructed hierarchies may be incomplete. Placeholder entries used during incremental reconstruction can also remain partially populated if the corresponding parent artifact is not present in the processed snapshot.

We mitigate these limitations by explicitly marking missing and unresolved fields, and documenting the extraction and resolution procedures. Users should consider these caveats when designing downstream analyses and when interpreting results that depend on metadata completeness or dependency resolution fidelity.

7 Data Quality Assessment

A data set for dependency and software supply chain studies must be reliable, not just large. We therefore report missingness, normalization decisions, and operational limitations so that users can filter records and interpret results correctly. Because dependency edges are the primary signal for downstream analyses and learning-based models, we also make dependency extraction auditable via per-artifact status and configuration fields.

7.1 Completeness

We quantify completeness as the missingness at the field-level for attributes commonly used in empirical analyses and tooling. Our release contains 52,016 artifacts from Maven Central, 1,817 from Google Maven, 8,820 from Atlassian, and 16,667 from Cloudera. These repository counts are source-observed counts from each repository snapshot prior to cross-repository merging. The released dataset is deduplicated by the Maven coordinate groupId:artifactId:version (stored as `_id`), when the same coordinate is observed in multiple repositories, a separate record is maintained for each repository it is observed in and its provenance is recorded in `origin_repository`. Figure 2 summarizes attribute coverage, and Table 5 reports missingness for `source_code_url`, `jar_size`, `description`, and `last_modified`. The normalized JSON release represents missing values as `Unknown`.

Differences across repositories reflect metadata practices: `last_modified` is near-complete in Maven Central listings, while `source_code_url` and `description` depend on maintainer-provided POM metadata and are therefore uneven across repositories.

Table 5: Missingness of key fields by repository. Values report the percentage of artifacts with missing values.

Repository	source_code_url	jar_size	description	last_modified
Maven Central	34.69	22.97	19.0	0.05
Google Maven	0.00	0.00	25.00	25.00
Atlassian	73.9	53.66	58.82	25.05
Cloudera	38.66	30.83	22.96	30.83

7.2 Consistency and normalization

Artifact key; Each artifact is identified by `groupId:artifactId:version` and stored as `_id`; we enforce exactly three segments after whitespace trimming.

Dependency edges. `direct_dependencies` denote the depth-1 children under the target artifact node in the resolver-produced tree (Section 5). Dependencies are stored as coordinate strings; Each direct dependency is stored in the form `groupId:artifactId:version:scope`.

Multiple repository sources. Since the same artifact may be resolved from multiple repositories, a record is maintained corresponding to each occurence of that artifact. Records are uniquely identified by the combination of `_id` and `origin_repository`, forming a composite key.

7.3 Accuracy and extraction reliability

SCM URL validity. On a checked subset of $n = 106$ artifacts with non-empty `source_code_url`, we observed 0 broken links (95% Wilson CI: [0.0, 3.5]). This check verifies that the URL is syntactically valid and reachable at the time of crawling, supporting reliable linking for downstream use.

Plausibility tags. We tag suspect values (e.g., unresolved `${...}` placeholders) to support the filtering of records with incomplete metadata.

Operational reliability of dependency extraction. Dependency extraction relies on external build tools and remote repositories. We use `mvn dependency:tree` for artifacts from Maven Central, Atlassian, and Cloudera, and Gradle dependency resolution for artifacts from Google Maven, with a 30-second per-artifact timeout. To keep the pipeline scalable, resolution is terminated when the timeout is exceeded. Artifacts with complex dependency graphs or slow metadata downloads may, therefore, yield incomplete dependency information. Users should interpret dependency lists with this limitation in mind, especially when comparing dependency-related statistics across repositories.

8 Availability and Artifact

We release both the MavCrawl dataset and the crawler implementation to support replication, reuse, and follow-up studies. The data set is archived on Zenodo as a versioned snapshot in structured JSON format. This makes it straightforward to load into databases or analysis pipelines and to reference the exact version of the data set used in an experiment. The Zenodo record also provides a persistent DOI for citation.

To support reproducibility, the artifact package includes the dataset, the crawler source code, and documentation that describes the directory structure, schema, and the steps required to reproduce

the crawling and dependency resolution workflow. The repository also includes scripts for importing the JSON files into MongoDB and for reproducing the descriptive statistics and figures reported in this paper.
Dataset (Zenodo DOI): https://zenodo.org/records/19678811
Source code (GitHub): https://github.com/arorasiya008/Maven-Dependency-Crawler

We plan to publish future releases as new Zenodo versions so that users can access both the most recent snapshot and prior snapshots for longitudinal analyses. Each release will remain available as a citable, immutable record.

9 Conclusion and Future Work

This paper introduces *MavCrawl*, a dataset construction pipeline and curated dataset for studying Maven dependency ecosystems. MavCrawl collects artifacts from Maven Central, Google Maven, Atlassian, and Cloudera, and it records artifact metadata together with direct dependencies and module relationships. The dataset supports reproducible research on dependency usage, maintenance signals, ecosystem structure, and software supply chain questions, without requiring researchers to reimplement Maven specific metadata parsing and dependency resolution logic.

Future work will extend repository coverage and improve metadata completeness, especially for missing provenance fields. We also plan to expand the release process with additional validation checks and richer summary statistics to support benchmarking and longitudinal studies across dataset versions.

References

[1] Mahmoud Alfadel, Diego Elias Costa, and Emad Shihab. 2023. Empirical analysis of security vulnerabilities in python packages. *Empirical Software Engineering* 28, 3 (2023), 59.

[2] Amine Benelallam, Nicolas Harrand, César Soto-Valero, Benoit Baudry, and Olivier Barais. 2019. The maven dependency graph: a temporal graph-based representation of maven central. In *2019 IEEE/ACM 16th international conference on mining software repositories*. IEEE, 344–348.

[3] Baltasar Berretta, Augustus Thomas, and Heather Guarnera. 2025. Dependency update adoption patterns in the Maven software ecosystem. In *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories*. IEEE, 295–299.

[4] Alexandre Decan, Tom Mens, and Philippe Grosjean. 2019. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering* 24, 1 (2019), 381–416.

[5] Johannes Düsing, Jared Chiaramonte, and Ben Hermann. 2025. MARIN: A Research-Centric Interface for Querying Software Artifacts on Maven Repositories. In *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories*. IEEE, 591–595.

[6] Hongbo Fang, James Herbsleb, and Bogdan Vasilescu. 2024. Novelty begets popularity, but curbs participation-a macroscopic view of the python open-source ecosystem. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–11.

[7] Keheliya Gallaba, Christian Macho, Martin Pinzger, and Shane McIntosh. 2018. Noise and heterogeneity in historical build data: an empirical study of travis ci. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 87–97.

[8] Yogya Gamage, Nadia Gonzalez Fernandez, Martin Monperrus, and Benoit Baudry. 2025. Software bills of materials in Maven Central. In *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories*. IEEE, 339–343.

[9] Joseph Hejderup, Arie Van Deursen, and Georgios Gousios. 2018. Software ecosystem call graph for dependency management. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*. 101–104.

[10] Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. 2017. Structure and evolution of package dependency networks. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories*. IEEE, 102–112.

[11] Rio Kishimoto, Tetsuya Kanda, Yuki Manabe, Katsuro Inoue, Shi Qiu, and Yoshiki Higo. 2025. A dataset of software bill of materials for evaluating SBOM consumption tools. In *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories*. IEEE, 576–580.

[12] Charles W Krueger. 1992. Software reuse. *Comput. Surveys* 24, 2 (1992), 131–183.

[13] Chengwei Liu, Sen Chen, Lingling Fan, Bihuan Chen, Yang Liu, and Xin Peng. 2022. Demystifying the vulnerability propagation and its evolution via dependency trees in the npm ecosystem. In *Proceedings of the 44th International Conference on Software Engineering*. 672–684.

[14] Raphina Liu, Sofia Bobadilla, Benoit Baudry, and Martin Monperrus. 2025. Dirty-Waters: Detecting Software Supply Chain Smells. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*. 1045–1049.

[15] MD McIlroy. 1968. Software Engineering: Report on a conference sponsored by the NATO Science Committee. In *NATO Software Engineering Conference, NATO Scientific Affairs Division*. 138–155.

[16] Dimitris Mitropoulos, Vassilios Karakoidas, Panos Louridas, Georgios Gousios, and Diomidis Spinellis. 2014. The bug catalog of the maven ecosystem. In *Proceedings of the 11th Working Conference on Mining Software Repositories*. 372–375.

[17] Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. 2020. Detection, assessment and mitigation of vulnerabilities in open source dependencies. *Empirical Software Engineering* 25, 5 (2020), 3175–3215.

[18] Piotr Przymus, Mikołaj Fejzer, Jakub Narębski, Krzysztof Rykaczewski, and Krzysztof Stencel. 2025. Out of sight, still at risk: The lifecycle of transitive vulnerabilities in maven. In *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories*. IEEE, 329–333.

[19] Steven Raemaekers, Arie Van Deursen, and Joost Visser. 2013. The maven repository dataset of metrics, changes, and dependencies. In *2013 10th Working Conference on Mining Software Repositories*. IEEE, 221–224.

[20] Jordan Samhi, Tegawendé F Bissyandé, and Jacques Klein. 2024. Androlibzoo: A reliable dataset of libraries based on software dependency analysis. In *Proceedings of the 21st International Conference on Mining Software Repositories*. 32–36.

[21] Luís Soeiro, Thomas Robert, and Stefano Zacchiroli. 2025. Wild sboms: a large-scale dataset of software bills of materials from public code. In *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories*. IEEE, 164–168.

[22] César Soto-Valero, Nicolas Harrand, Martin Monperrus, and Benoit Baudry. 2021. A comprehensive study of bloated dependencies in the maven ecosystem. *Empirical Software Engineering* 26, 3 (2021), 45.

[23] Aman Swaraj and Sandeep Kumar. 2025. Bridging AI and Human Knowledge: Towards a Deeper Understanding of Stack Overflow and ChatGPT. In *Proceedings of the 29th International Conference on Evaluation and Assessment in Software Engineering*. 976–985.

[24] Anastasiia Tkalich, Eriks Klotins, and Nils Moe. 2025. Identifying critical dependencies in large-scale continuous software engineering. In *Proceedings of the 29th International Conference on Evaluation and Assessment in Software Engineering*. 690–695.