TU WIEN Informatics

A Systematic Approach to the Temporal Analysis of Dependency Updates in Software Projects

A Case Study on Selected Open Source Projects

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Informatics

by

Konstantin Unterweger
Registration Number 12222196

to the Faculty of Informatics

at the TU Wien

Advisor: Privatdoz. Dipl.-Ing. Mag.rer.soc.oec. Dr.techn. Edgar Weippl
Assistance: Univ.Lektor Dipl.-Ing. Dr.techn. Alexander Schatten
Philip König, BA BSc MSc

Vienna, June 18, 2025

Konstantin Unterweger	Edgar Weippl

Abstract

Effectively managing third-party dependencies is crucial for software maintenance, yet we know surprisingly little about what drives update practices and version synchronisation. This thesis investigates two key questions: First, how do traditional measures of **project size** (GitHub stars, contributor count, and commit volume) affect the **latency of dependency updates**? Second, how does the number of manifests compare to the used dependencies?

To answer these questions, I created a language-agnostic tool that reconstructs the complete history of dependency manifest changes (e.g., `pom.xml`, `package.json`, `requirements.txt`) across commits. I applied this tool to nine diverse open-source repositories written in Java, Python, and JavaScript. From these, I extracted five metrics: update frequency, mean time between updates, update latency, version lag, and average criticality score. I then analysed how these metrics relate to project size indicators and **structural complexity** (the number of manifest files).

My findings show that traditional project size metrics have only a weak correlation with update latency and version lag. However, **structural complexity**, specifically the number of distinct dependency manifest files, demonstrates a strong and statistically significant association. Projects with five or more manifest files experience an average update latency that is up to 250 days longer and lag several releases behind upstream. A case study of Keycloak highlights the effectiveness of an automated "dependency-sync" pipeline, which reduced its Mean time between updates from 30 weeks to less than 7 weeks.

These results emphasise the critical need to minimise manifest fragmentation and adopt automated synchronisation to enhance software maintainability and security. I conclude with practical guidelines for developers and suggest future research avenues, including import-level analysis, cross-ecosystem extensions, and the socio-technical aspects of dependency management.

Contents

CHAPTER 1

Motivation and Research Questions

1.1 Motivation

The management and evolution of software dependencies are critical components in maintaining software quality and ensuring long-term security. The evolution of software dependencies refers to the continuous changes and updates of external libraries and frameworks that a software project uses. As early as 1994, Parnas [Par94] introduced the concept of software aging, noting that, like any engineered artefact, software deteriorates over time due to the accumulation of patches, workarounds, and adaptations to changing environments. This notion applies not only to the source code itself but also to external libraries, which can become outdated, leading to compatibility issues and heightened security risks.

Empirical research underscores the widespread reliance on third-party libraries in modern software development. For instance, Kula et al.(2018)[KGO+18] found that 81.5% of a sample of 4,600 GitHub repositories declared at least one external dependency. This update inertia, Wang et al.(2020)[WCH+20] report that 54.9% of projects never update more than half of their declared dependencies, highlighting the ongoing maintenance burden and associated risks within the open-source ecosystem.

1.2 Research Questions

While the concept of software ageing is well established concerning source code, there remains limited understanding about how external dependencies evolve and how characteristics of software projects influence their maintenance behaviour. To address this gap, this thesis investigates the following research questions:

RQ1 - Relationship between Project Size and Update Latency To what extent is a project's size, measured by GitHub stars, the number of active contributors, and project age, associated with its *update latency*, defined as the time between the public release of a dependency version and its adoption in the project?

- **Project Size:** GitHub stars, Number of active contributors in the repository, Project age, measured from the timestamp of the first public commit
- **Update Latency:** The number of days between a dependency version's public release and the commit in which it is adopted.

RQ2 - Influence of Number of Manifest Files on Number of Dependencies How does the number of manifest files impact the number of unique dependencies used in a project?

- **Number of manifest files:** Total count of dependency manifest files (e.g., `pom.xml, build.gradle, package.json`) per project.
- **Number of unique dependencies:** Distinct third-party libraries declared across all manifest files of the project.

1.3 Objective

Building on the above motivation and research questions, the primary objective of this thesis is to design and evaluate an automated, language-agnostic tool that reconstructs the full history of dependency version changes across commits. This tool traverses every commit in selected Java, Python, and JavaScript repositories, extracting information from dependency configuration files (e.g., `pom.xml`, `package.json`, `requirements.txt`) and build scripts (e.g., `build.gradle`, `setup.py`). The resulting dataset enables a comprehensive analysis of how and when dependencies are updated in practice.

1.4 Structure of the Thesis

To address the research questions and achieve the stated objective, the thesis is structured as follows:

- **Chapter 2 – Background and Fundamentals:** It outlines how the focus of Open Source Software (OSS) projects shifts over time, introduces key dependency management tools (Maven, Gradle, npm, pip), and outlines the relevant configuration file formats.
- **Chapter 3 – Related Work:** Reviews prior research on software ageing, repository mining, dependency evolution, and existing tools for dependency analysis.
- **Chapter 4 – Methodology:** Details the project selection criteria, data collection approach using GitHub, identification of relevant commits, extraction and normalisation of version and date information, and computation of core metrics such

as update latency, version lag, and the Open Source Security Foundation (OSSF) score.

- **Chapter 5 – Results:** Presents the findings of the empirical analysis, including descriptive statistics (e.g., number of commits and repositories), temporal trends in dependency updates, language-specific comparisons, and correlations with project size indicators.
- **Chapter 6 – Discussion:** Interprets the results in light of existing literature, discusses the implications for maintainability and software security, and critically reflects on methodological limitations such as incomplete release data and potential selection biases.
- **Chapter 7 – Conclusion and Outlook:** Summarises the main findings, provides practical recommendations for dependency management, and outlines directions for future research in this area.

CHAPTER 2

Background and Fundamentals

This chapter lays the groundwork by outlining the typical lifecycle phases of open-source projects and motivating commit-log analysis as an evolution indicator, surveying common dependency-management tools and formats and their implications for build reproducibility and security, and listing the specific file types analysed in this work.

2.1 Typical lifecycle of Open-Source Projects

OSS projects typically evolve through several distinct phases: initialisation, growth, and maintenance. During the initialisation phase, a project is created, foundational functionality is implemented, and the groundwork for future development is laid. The growth phase is characterised by active development, the addition of new features, increasing contributor participation, and the expansion of the user base. Eventually, many projects enter a maintenance phase, in which development slows, and the focus shifts to bug fixing, security updates, and dependency management.

To track these phases transition, commit logs serve as a reliable indicator of a project's evolution, as they document both the frequency and type of changes made over time. Foundational studies, such as those by Mockus and Votta [MV00] and Hassan [Has09], have demonstrated that commit data can reveal valuable insights into software maintenance, development practices, and overall software quality. This thesis builds on these insights by analysing the temporal evolution of dependency versions and examining their implications for software maintenance and quality.

2.2 Dependency Management Tools

Modern projects rely heavily on dependency-management systems to declare, resolve and integrate external software:

Maven (`pom.xml`): a build automation and dependency management system for Java Projects. By utilising an XML-based Project Object Model (POM), Maven not only defines and documents the direct dependencies of a project in its `pom.xml`, but also automatically resolves and lists all transitive dependencies from its central repositories. With its extensive plugin ecosystem, it has become a big player in dependency management for Java [Fou25].

Gradle (`build.gradle, build.gradle.kts`): a dependency management and build automation tool designed for larger, more complex projects. It supports various languages such as Android, Java, and Kotlin. Unlike Maven's XML-based Project Object Model, Gradle offers a scripting Domain Specific Language (DSL) (Groovy/Kotlin) that provides high flexibility and customisation of build logic, making it well-suited for multi-module builds, custom task definitions, and intricate dependency resolution strategies. Gradle still resolves direct and transitive dependencies from repositories by default, but its configurable resolution rules, incremental compilation, and performance optimisations make it ideal for advanced, complex build pipelines [Fsc25a].

npm (`package.json`): a comprehensive open-source software registry and package manager for JavaScript and Node.js applications. Maintained by a global community of open-source developers, npm hosts millions of packages, enabling easy installation of libraries, frameworks, and Command Line Interface (CLI) tools [Fsc25b].

pip (`requirements.txt, pyproject.toml`): a library installation and environment reproducibility for Python projects. It installs packages from Python Package Index (PyPI) or custom indexes, pins exact versions in `requirements.txt`, and leverages PEP 517/518 build-system declarations in `pyproject.toml` [Fsc25c].

2.3 Analysed Files

This thesis will only look at the following files, each corresponding to one of the dependency-management systems introduced above:

- `pom.xml` (maven)
- `build.gradle` & `build.gradle.kts` (Gradle Groovy/Kotlin)
- `package.json` (npm)
- `requirements.txt` (pip requirements file)
- `setup.py` (supported by pip)
- `pyproject.toml` (PEP 517/518 Python build system declaration, pip)

These files were selected because they are the primary sources for dependency declarations in their respective ecosystems. Changes to these files reflect updates to direct or transitive dependencies. By tracking how and when these files are modified across a project's history, this thesis aims to extract patterns in dependency evolution and understand how open-source projects manage external code over time.

CHAPTER 3

Related Work

3.1 Analysis of Commit Histories and Code Changes

The systematic analysis of version control data has been widely explored in software engineering research. A Version Control System (VCS) is a system that tracks version changes. In Programming, Git is one of the biggest VCS and widely used. Mockus and Votta [MV00] were among the first to show that meaningful insights can be extracted from historical software change data. Hassan [Has08] extended this line of research by developing automated techniques for classifying commit messages, distinguishing between different types of changes such as bug fixes and feature additions.

These studies form the methodological foundation for many modern repository mining approaches and demonstrate the analytical potential of commit history in understanding software evolution.

3.2 Studies on Dependency Evolution and Update Behaviour

A key focus of this thesis is the investigation of library version updates and dependency evolution over time. While much prior work has concentrated on general code changes or defect prediction (e.g., [CKK+18], [MPS08]), the analysis of dependency management in OSS has recently gained increased attention. Researchers have begun to explore how frequently libraries are updated, under what conditions updates occur, and what impact these decisions have on maintainability, compatibility, and security. Combined with metrics such as the OSSF Score, these studies enable a more nuanced understanding of software quality in the context of dependency management.

3.3 Tools and Frameworks for Dependency Analysis

Various tools and platforms have been developed to support the analysis and management of software dependencies. For example, Degraph[1] provides visualisation and analysis capabilities for dependencies within and across different versions of a project. Similarly, VersionEye[2] was an online service designed to monitor outdated or vulnerable libraries used in software projects. Although VersionEye has since been discontinued and its source code archived, it represents an early effort to automate dependency monitoring.

Modern build tools, as discussed in Section 2.2, offer built-in mechanisms for dependency resolution and management. Their configuration files (e.g., pom.xml, build.gradle) serve as valuable sources for analysing version histories and tracking dependency changes over time.

A particularly prominent and widely adopted tool in this context is GitHub's Dependabot[3]. Dependabot automatically detects and parses manifest files within a repository to identify outdated dependencies and known security vulnerabilities. It regularly checks for new releases of dependencies and provides developers with update recommendations, including the urgency level based on security advisories. Furthermore, Dependabot can autonomously generate pull requests to update dependencies, streamlining the maintenance process and reducing the manual effort required for dependency management.

These tools, along with related research efforts, underscore the growing importance of systematically tracking and analysing dependency updates in OSS projects. This work contributes to that body of research by providing a structured, version-controlled investigation into the evolution of dependency configurations, offering insights into software maintenance practices and long-term project sustainability.

[1] `https://riy.github.io/degraph/` (last accessed 2 June 2025)
[2] `https://www.versioneye.com/` (last accessed 2 June 2025)
[3] `https://github.com/dependabot`(last accessed 2 June 2025)

CHAPTER 4

Methodology

Software repositories undergo continuous evolution, shaped by patterns of dependency updates, maintenance activities, and signals that may indicate technical debt or varying levels of software quality. To analyse these developments in a systematic and reproducible manner, a structured methodological approach is required, one that captures the historical progression of dependencies, ensures consistency through normalisation, and facilitates the extraction of relevant metrics from individual commits.

In this context, Section 4.1 introduces the set of selected projects and outlines the criteria used for their inclusion in the study. Sections 4.2 and 4.3 describe the data collection and preprocessing pipeline in detail, including repository access, the identification of relevant commits involving dependency changes, and the parsing and normalisation of extracted data into a uniform format. Furthermore, these sections explain how additional contextual information, such as release dates and the most recent available versions, is obtained to support subsequent metric computation. Section 4.4 then defines the set of metrics used in the analysis and outlines their method of calculation. Finally, Section 4.5 presents the structure of the resulting output file, which consolidates all computed values and serves as the basis for further analysis.

4.1 Project selection

A diverse set of open-source projects was selected, both large, well-established repositories and smaller, more recent ones to enable comparative analysis across different project maturities. The selection criteria included the number of GitHub stars, the total number of commits, the date of the initial commit, and the number of published releases. The goal was to achieve a balanced sample that reflects different stages of project evolution. For instance, *Keycloak* is a mature project, with 82 official releases, approximately 27,000 stars, and an initial commit dated July 2, 2013, which has accumulated over 28,000 commits. In contrast, *Caldera* is a more recent project, featuring 34 releases, around 6,200

Repo	Main Language	Commits	Contributors	Age (Years)	Stars
Keycloak	Java	28,000	1,300	12	27k
Apache Tomcat	Java	27,000	130	19	7.8k
ZAP	Java	10,000	218	15	13.6k
checkmk	Python	86,000	251	16	1.9k
cve-search	Python	2,000	62	12	2.5k
IntelOwl	Python/JavaScript	2,800	72	5	4.1k
YETI	Python	3,400	59	12	1.9k
SpiderFoot	Python	3,700	53	13	14.5k
Caldera	Python	4,400	100	8	6.2k

Table 4.1: List of all analysed repositories (URLs in Appendix: Repository URLs)

stars, and an initial commit on July 23, 2018. The table 4.1 contains all the repositories that are going to be analysed in this thesis.

4.2 Data mining

To effectively obtain all the necessary information from the selected OSS projects, the repositories were cloned locally from GitHub. This approach was chosen over exclusive reliance on the GitHub Application Programming Interfaces (API) due to its request rate limitation of 5,000 requests per hour, which is insufficient when analysing repositories with over 28,000 commits. Additionally, querying the API for each commit introduces significant latency, making the process inefficient. By cloning the repositories, it becomes possible to access the complete commit history efficiently and retrieve the contents of specific commits as needed for further analysis. For this analysis, we included only commits made on or before June 2, 2025.

The full source code, data-collection scripts, and analysis notebooks used in this study are publicly available (see Appendix 7.4).

After cloning the repository, the next step involves identifying all files relevant to the analysis, based on a predefined list of target file types. This is achieved by traversing the repository and comparing each file against the list of relevant filenames or extensions. Whenever a match is found, the corresponding file path is stored in a collection for further processing. Once the traversal is complete, we obtain a list of file paths pointing to the relevant files within the repository.

To retrieve the commit history for each relevant file, including all commits in which the file was modified, even across renames or file moves, the following Git command is executed for every identified path via a subprocess:

```
git -C <repo_path> log --follow --pretty=format:%H --name-only -- <
    rel_path>
```

Listing 4.1: Git command used to extract commit history for relevant files (uses `--follow` to trace renames, `--pretty=format:%H` to output commit SHAs, and `--name-only` to list file paths)

As shown in Listing 4.1, `<repo_path>` refers to the path of the repository currently being analysed, while `<rel_path>` denotes the relative path to a relevant file, as previously identified. This command returns a complete list of commit hashes in which the specified file was modified, thereby forming the foundation for subsequent content analysis.

By executing this command for every file in the list of relevant files and aggregating the resulting commit hashes into a unique set, a minimised list of commits in which at least one relevant file was changed can be obtained. This reduction limits the number of commits whose file contents must be loaded, thereby improving overall efficiency.

4.3 Parsing and Normalisation

With the list of relevant commits identified, the next step involves extracting the required dependency information from the corresponding files. This process varies significantly depending on the structure and format of each file type.

For instance, `requirements.txt` files are relatively straightforward to parse, as they typically consist of a plain list of dependencies, each optionally accompanied by a version specifier. In such cases, the parsing process involves iterating through the commit history of each file, extracting all listed dependencies along with their specified versions. This allows us to determine exactly which version of each dependency was used at any given point in time.

A similar approach applies to `package.json` files, which define dependencies in a structured JavaScript Object Notation (JSON) format. These files also explicitly declare dependencies and their versions, making it possible to extract the required information reliably by parsing their content at each relevant commit.

In contrast, parsing `pom.xml` files from Maven projects presents additional complexity due to their hierarchical and declarative structure. In `pom.xml`, it is common practice to define variables (typically using the <properties> tag), which are then referenced throughout the project or inherited by child POMs. This indirection can make it difficult to resolve the actual version of a dependency from a single file alone, as the value of a referenced variable may be declared in a parent POM located elsewhere in the project.

To address this, the approach involves first identifying the path to the parent POM, which is declared in the current POM via the <parent> tag. The parent POM is then loaded and checked for the definition of the required variable. If the variable is not found, the process is repeated recursively up the hierarchy until the root POM is reached. If the variable is not declared at any level, the POM structure is considered invalid for dependency resolution.

To perform this process efficiently, the commit history is traversed in chronological order. For each commit, all relevant `pom.xml` files are loaded and their latest versions are cached in memory. This caching mechanism minimises the number of Git subprocess calls required, thus improving performance during the resolution of inherited variables and dependencies.

To consolidate the extracted data, it was serialised into a single, uniform JSON document keyed by each commit's hash and timestamp. Within each entry, a subsection for every relevant file was included, listing each dependency alongside its resolved version. An additional field, `ecosystem`, identifies the source registry (e.g., PyPI, npm), which later guides the retrieval of supplementary metadata. Based on the ecosystem field, the public `deps.dev`[1] API is queried for each dependency's full release history. From that response, the latest version as of the commit date is recorded. If `deps.dev` does not list the package or does not provide a release date, a secondary lookup against `Maven Central` [2] is performed. Finally, the associated OSSF Score from deps.dev is fetched, thereby enriching the dataset with security and maintenance metrics needed for downstream analyses.

4.4 Metrics

To quantify dependency-related activity at the granularity of individual commits and files, a set of complementary metrics was defined. These metrics capture both the temporal dynamics of updates and the qualitative characteristics associated with each change. Five principal measures were adopted: the frequency of project-wide dependency updates, the mean time between updates (MTBU) for each dependency, the latency with which new versions are adopted, the version lag relative to the latest available release, and the criticality of dependency changes as reported by the OSSF Score. In addition, indicators were introduced to detect occurrences of pre-release versions (i.e., alpha, beta, or release candidate) within each commit. The following paragraphs detail how each metric is computed and integrated into the final, aggregated dataset.

First, to assess the breadth of dependency activity over time, the entire commit history is partitioned into fixed-length intervals, each spanning thirteen weeks. An interval's start point is determined by the timestamp of the earliest relevant commit in the repository under analysis, and successive intervals follow in thirteen-week increments. Each commit that modifies at least one dependency-related file is assigned to the interval corresponding to its date. Within each interval, the number of unique dependencies that appear in added or updated files is counted, producing the project update frequency. This frequency reflects the intensity of dependency changes: intervals with higher counts indicate periods when the project as a whole experienced more diverse dependency modifications.

Second, the MTBU for a given dependency is calculated by examining the sequence of commits in which that dependency was altered. After sorting commits chronologically by

[1] `https://docs.deps.dev/api/v3/`

[2] `https://central.sonatype.org/search/rest-api-guide/`

their timestamps, the difference in days between consecutive updates for each dependency is computed. These inter-update intervals are then averaged and converted to weeks by dividing the mean value by seven. In practice, this results in a single MTBU value (in weeks) for each dependency, characterising how regularly that dependency receives updates. A smaller MTBU suggests a rapid update cycle, whereas a larger MTBU indicates that updates occur more sporadically.

Third, update latency measures how promptly a newly released version of a dependency is incorporated into the project. For each commit, the date on which the updated version became available upstream is retrieved primarily via the public `deps.dev` API. In cases where release-date information or the latest version cannot be found on `deps.dev`, a fallback lookup is performed against Maven Central (for Java dependencies) to ensure that no release metadata is omitted. The difference, in days, between the release date and the commit date is recorded as the latency. When multiple dependencies are updated in the same commit, the latency values are averaged to produce a single update latency per commit and file. This average latency reflects the team's responsiveness to upstream changes: low latency implies that new releases are adopted quickly, whereas high latency suggests a delay between release publication and project integration.

Fourth, version lag quantifies how far behind the project's adopted version is relative to the latest available release at that point in time. For each dependency, the version specified in the commit is compared against the most recent version obtained—preferentially from `deps.dev` and, when necessary, from Maven Central—at the commit timestamp. The numerical difference, expressed according to semantic versioning rules, yields the version lag. Specifically, if the commit's version is 1.2.3 and the latest upstream version is 1.4.0, the lag in major/minor/patch is computed as the number of intervening patch or minor releases. In many cases, one can calculate it as the difference in the patch component when the major/minor match; otherwise, it is a sum of releases. When a commit updates multiple dependencies, individual version lags are averaged to produce a single value for that commit and file. A version lag of zero indicates that the project was using the latest available version at the time of the commit, while a positive lag reveals that newer releases were already available but not yet integrated.

Fifth, to capture the security and maintenance relevance of dependency changes, the OSSF Score is retrieved for each dependency version via `deps.dev`. When a dependency's score is not present in `deps.dev`—for example, if the package is new or lacks a registry entry—a secondary lookup is performed on Maven Central to obtain any available metadata; if neither source provides a score, the dependency's criticality remains undefined. For each commit, the criticality scores of all updated dependencies are aggregated in two ways. The average criticality score (denoted as `proj_criticality_scoreAVG`) is computed by taking the mean of all valid OSSF Scores associated with updated dependencies in that commit. Simultaneously, the sum of criticality scores (denoted as `proj_criticality_scoreSUM`) provides an aggregate measure of the overall urgency of changes within that commit. If no valid numeric scores are available for a given commit, owing to missing or non-existent entries in both `deps.dev` and Maven Central,

commit_hash	commit_date	file_path	proj_update_frequency	proj_mtbu	proj_update_latency	proj_version_lag	proj_criticality_scoreAVG	proj_criticality_scoreSUM	proj_alpha_best	proj_alpha_used
81726bfbdcf08fdf2fde2517545bfbd2ee36095e	2018-12-20 15:41:44	package.json	N/A	40.52	84.5	0.0	5.46	43.7	0	0.0
990bd85041ce07c7aa4f6dcede434c2ce0ea1d80	2018-12-20 17:02:54	package.json	N/A	46.28	128.36	0.0	5.29	58.2	1	1.0
af2297ea7460cb8e7524db1568920e5a7078d671	2019-02-11 15:43:39	package.json	9.0	56.32	178.78	0.0	3.36	30.2	0	0.0
90704ae43d88910336c0e6e7e28cc2193f0c205e	2019-04-05 15:03:37	package.json	4.0	16.72	268.5	0.0	3.05	6.1	0	0.0
980368866ce18f56cac24b6bd0debb8226ea1d36d	2019-04-12 08:47:30	package.json	4.0	26.96	1.0	0.0	3.1	3.1	0	0.0
1c67595631f14ebd0f488aaf8a437d8768f208a6	2019-05-13 07:35:53	package.json	4.0	25.61	16.0	0.0	3.8	3.8	0	0.0
cf01ef03d72a2e345f19ade4335b286a1ed253c1	2019-05-13 08:30:42	package.json	4.0	17.0	13.5	0.0	5.35	10.7	0	0.0
e279d5960e21773b5f2d8aeb7f5f633bd843baf8	2019-06-26 20:59:30	package.json	19.0	26.49	461.0	17.0	4.1	4.1	0	0.0
c378f89ad664a5713b5a0b3b1fb4a0ecfc00036d	2019-08-15 14:27:44	package.json	19.0	81.61	497.0	2.0	1.5	1.5	0	0.0

Figure 4.1: Example output of metrics

both average and summed criticality fields are marked as 'N/A' to distinguish them from valid zero-valued scores.

Finally, to track the adoption of pre-release dependency versions, two binary indicators are recorded for each dependency update: one corresponding to labels containing "alpha", "beta", or "rc", and another flagging the introduction of a new pre-release version. During parsing, each dependency's version string is examined for substrings that match common pre-release identifiers. If present, the corresponding indicator is set to one. When multiple dependencies within the same commit match these criteria, the flags are summed to reflect the total number of pre-release adoption events. These counts `proj_alpha_best` for any occurrence of alpha, beta, or release candidate tags, and `proj_alpha_used` for newly introduced pre-release versions allow for analysis of how frequently the project adopts non-stable releases.

4.5 Data output

Once all component metrics have been calculated, they are merged at the granularity of each combination of commit hash, commit date, and file path. This fusion produces a single, comprehensive record per file change, encompassing project update frequency (based on the interval assignment), MTBU for the specific dependency modified, average update latency, average version lag, average and summed OSSF Scores, and counts of pre-release indicators. All numerical metrics are rounded to two decimal places. If a metric is undefined for a given commit-file (e.g., no OSSF Score exists), the field is set to 'N/A'. The resulting dataset is sorted in ascending order by commit date, yielding a time-series view of dependency updates at the file level. This consolidated output file (as exemplified in Figure 4.1) serves as the fundamental basis for downstream analyses, enabling in-depth investigations into temporal patterns, correlations between update practices and criticality, and potential indicators of technical debt.

CHAPTER 5

Results

In this chapter, the collected data are presented in the same order as the metrics defined in Chapter 4. The raw data are shown primarily as tables and figures without extensive interpretation. A summary of key findings follows in section 6.3.

5.1 Descriptive statistics

Following the systematic analysis of nine open-source repositories — Keycloak, Apache Tomcat, The ZAP, checkmk, cve-search, IntelOwl, YETI SpiderFoot, and Caldera — descriptive statistics were extracted to provide an overview of the underlying dataset. Table 5.1 reports, for each project, the total number of commits that changed at least one dependency file, identified over the study period, as well as the total number of dependency files detected within the respective codebases. These figures serve as foundational metrics for interpreting later results.

Project	Commits	Dependency Files
Keycloak	4,869	161
Apache Tomcat	91	4
ZAP	277	16
checkmk	630	23
cve-search	153	1
IntelOwl	135	8
YETI	55	5
SpiderFoot	67	4
Caldera	87	2

Table 5.1: Total dependency-related commits and number of manifest files per project.

Table 5.1 shows that Keycloak has the highest number of dependency-update commits (4,869) and the largest set of dependency files (161), whereas cve-search has only 153 commits and a single dependency file. This disparity reflects differences in project ecosystems: for example, Keycloak's Java-based modules rely on numerous `pom.xml` files, while cve-search maintains a single Python `requirements.txt` file.

Project	Upd. Freq. (updates/quarter)			Upd. Latency (days)			Ver. Lag (releases behind)			Crit. Score AVG		
	Mean	Median	Std. Dev.	Mean	Median	Std. Dev.	Mean	Median	Std. Dev.	Mean	Median	Std. Dev.
Keycloak	124.06	111.00	48.78	118.35	5.00	327.36	4.19	0.00	18.51	5.76	5.65	1.54
Apache Tomcat	10.39	10.00	4.89	62.65	11.50	146.83	0.62	0.00	1.17	4.82	5.13	1.39
ZAP	17.62	18.00	10.03	203.79	17.75	532.29	0.56	0.00	1.39	—	—	—
checkmk	77.39	34.00	103.35	178.56	44.79	350.37	1.66	0.00	4.88	5.16	5.30	1.36
cve-search	13.45	11.00	13.43	119.83	2.00	317.40	0.26	0.00	0.70	4.58	4.40	0.95
IntelOwl	15.58	13.50	8.53	72.36	21.00	125.26	0.57	0.00	1.41	4.96	4.90	1.21
Yeti Platform	8.29	6.50	8.59	152.21	20.84	253.21	0.00	0.00	0.01	5.29	5.40	1.27
SpiderFoot	20.43	6.00	21.36	290.46	110.19	359.59	6.36	0.00	23.26	4.07	3.83	1.32
MITRE Caldera	8.37	6.00	6.35	183.38	87.25	267.56	1.68	0.00	3.50	5.11	5.20	1.22

Table 5.2: Summary of Key Update Metrics for each Project
Note: '—' indicates no valid data for that metric.

The Table 5.2 presents, for each of the nine selected projects, three summary statistics (mean, median, standard deviation) across four core metrics: update frequency (measured in updates per quarter), update latency (days between an upstream release and its incorporation), version lag (number of released versions behind at each commit), and average criticality score (a numeric indicator of dependency risk). Each row corresponds to one project, and each group of three columns under a given metric shows its central tendency and variability. The entries denoted by '—' under the Criticality Score column indicate that the source dataset does not include any OSSF score information for the analysed dependencies, and thus, criticality values could not be computed.

Update Frequency quantifies the number of dependency-update commits performed per quarter: higher values indicate more frequent integration of upstream changes, whereas lower values suggest a more deliberate and less volatile update cadence. Update Latency denotes the interval, in days, between a dependency's upstream release and its incorporation into the project; shorter latency is desirable because it reflects more rapid adoption of security patches and feature updates. Version Lag measures how many released versions the project trails behind the latest release at any given commit; a smaller lag signifies that the codebase remains closely synchronised with upstream developments. The Average Criticality Score ranges from 0 to 10, with a score of 10 representing minimal dependency risk and thus denoting the most favourable outcome.

5.2 Understanding Average Latency

Figure 5.1 shows the average update latency (in days) for each project, calculated on a per-year basis. Each row corresponds to one of the nine Projects (Keycloak, Apache Tomcat, ZAP, checkmk, cve-search, IntelOwl, Yeti, SpiderFoot, and Caldera), and each column represents a calendar year from 2011 through 2025. The colour of each cell

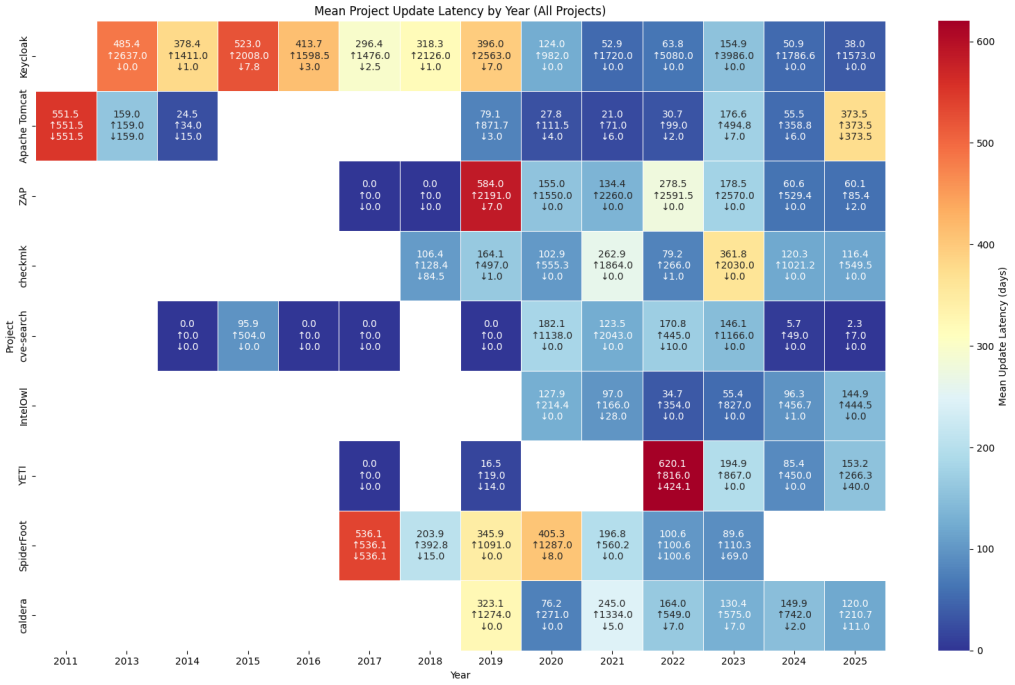
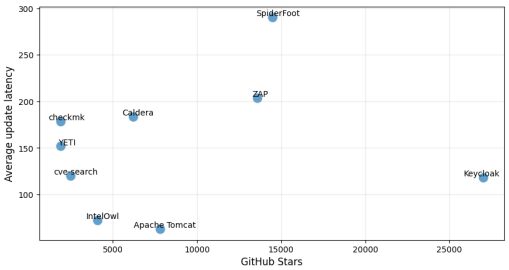


Figure 5.1: Mean update latency for each project
(*Note:* ↑ is the max value, ↓ is the min value in the interval)

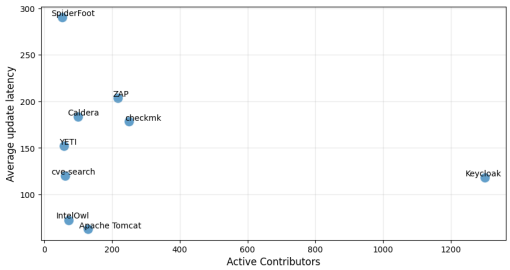
encodes the numeric value of the mean update latency in that year (with darker reds indicating higher latency and dark blues indicating very low or zero latency), as noted in the adjacent colour bar. Cells left blank (white) denote years for which no valid latency data were available, which could indicate that the project has not yet started or has paused. In each non-blank cell, the exact average latency (rounded to one decimal place) is overlaid in white font.

In Figure 5.1, Apache Tomcat exhibits a four-year hiatus during which no dependencies were updated; after this gap, updates resume with relatively low latency. This pattern may suggest that the project paused until certain upstream dependencies reached a stable state before proceeding. Additionally, many projects display higher latencies during their initial phases, which is to be expected as maintainers often delay first-time dependency upgrades until the codebase reaches sufficient maturity.

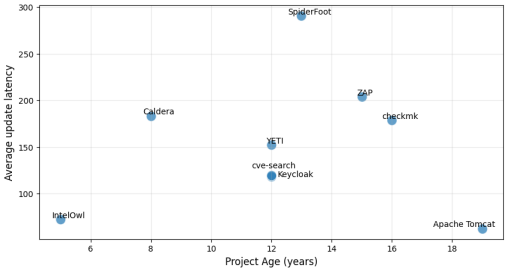
The association between mean update latency and project-size proxy, namely: GitHub stars, contributor count, and project age, is depicted in Figure 5.2, as described in Section 4.1.



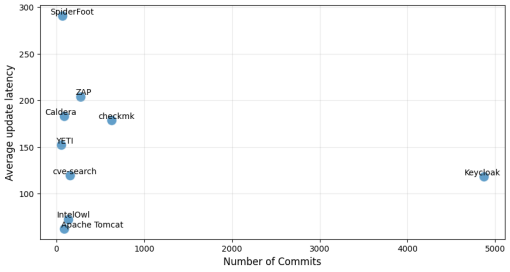
(a) GitHub Stars vs. Mean Update Latency



(b) Contributors vs. Mean Update Latency



(c) Project Age (years) vs. Mean Update Latency



(d) Number of Commits vs. Mean Update Latency

Figure 5.2: Relationship between project-size metrics and mean update latency.

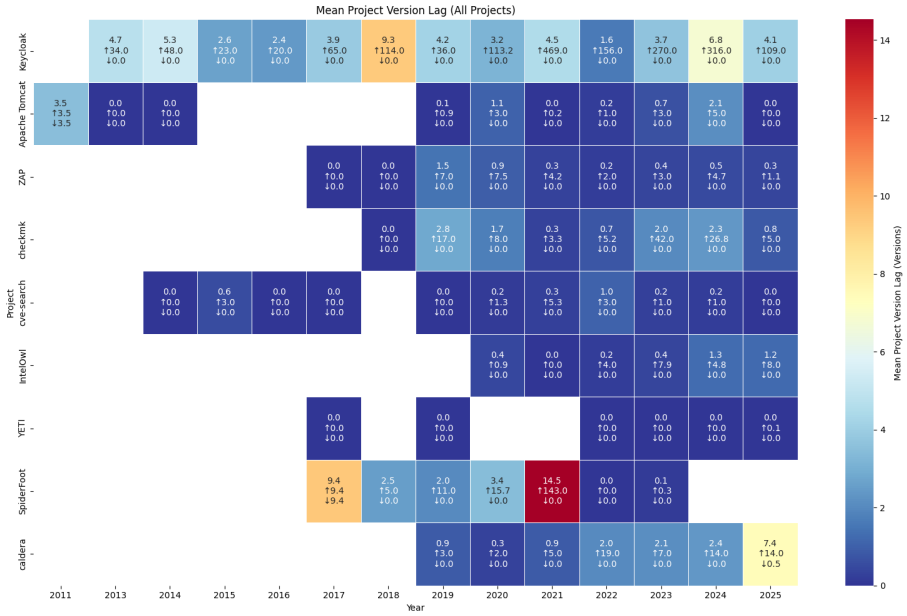
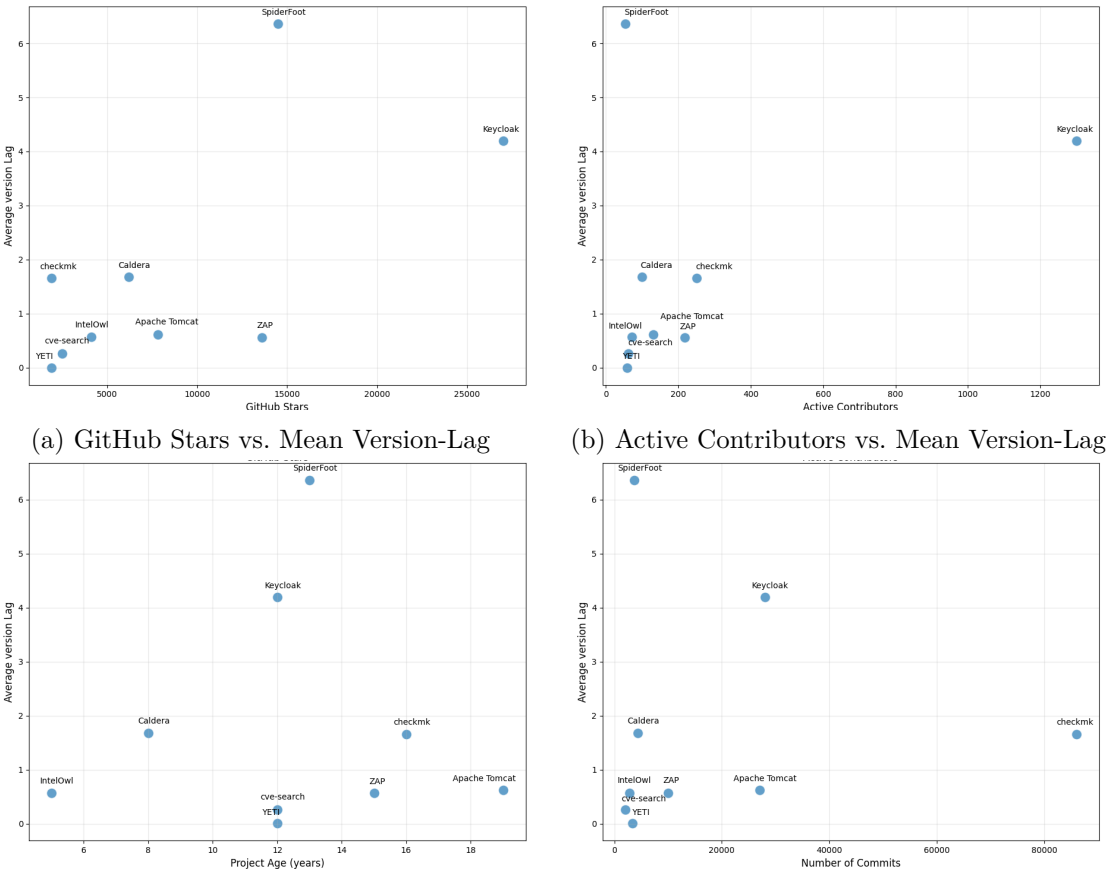


Figure 5.3: Mean Version Lag per Year
(*Note:* ↑ is the max value, ↓ is the min value in the interval)



(a) GitHub Stars vs. Mean Version-Lag

(b) Active Contributors vs. Mean Version-Lag

(c) Project Age (years) vs. Mean Version-Lag

(d) Number of Commits vs. Mean Version-Lag

Figure 5.4: Influence of project size on the average version lag.

5.3 Insights into Version Lag

To investigate the association between average version lag and project-size metrics, Figure 5.4 presents four scatterplots: version lag versus GitHub stars (Fig.5.4a), active contributors (Fig.5.4b), project age (Fig.5.4c), and total dependency-update commits (Fig.5.4d). A deeper insight into version Lag brings us the Figure 5.3, where we have the version lag for each project and the corresponding value of version lag.

Another very valuable insight into the updated cycle of OSS projects is the MTBU. To examine that, a heat-map was created 5.5.

Figure 5.5 illustrates the annual MTBU for each project, thereby revealing distinct temporal trajectories. Lower MTBU values denote more frequent update cycles, whereas higher values indicate that dependency updates occur less often. In the case of Keycloak, the MTBU undergoes a marked reduction over the project's duration: intervals that initially spanned approximately 20 weeks have converged to an approximate 6-week

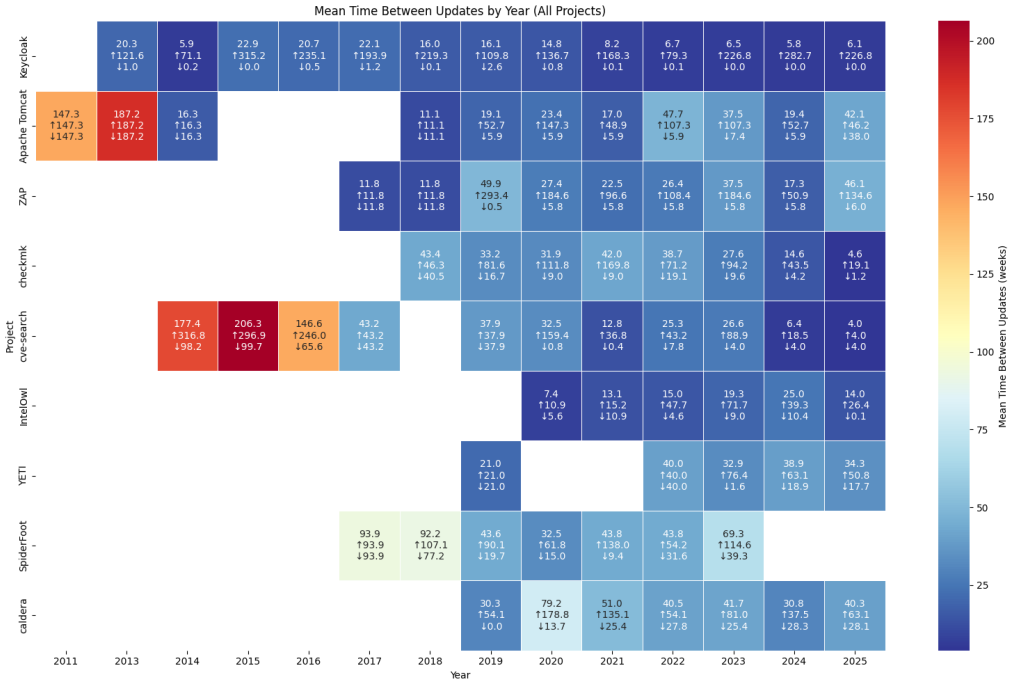


Figure 5.5: Mean Time between updates per project per year
(*Note:* ↑ is the max value, ↓ is the min value in the interval)

cadence in the most recent years.

5.4 Temporal trends mean number of dependencies

Figure 5.6 illustrates the temporal evolution of the mean number of unique dependencies across all selected OSS projects. The analysis is divided into three subfigures to provide both an overall and a more granular perspective on dependency trends.

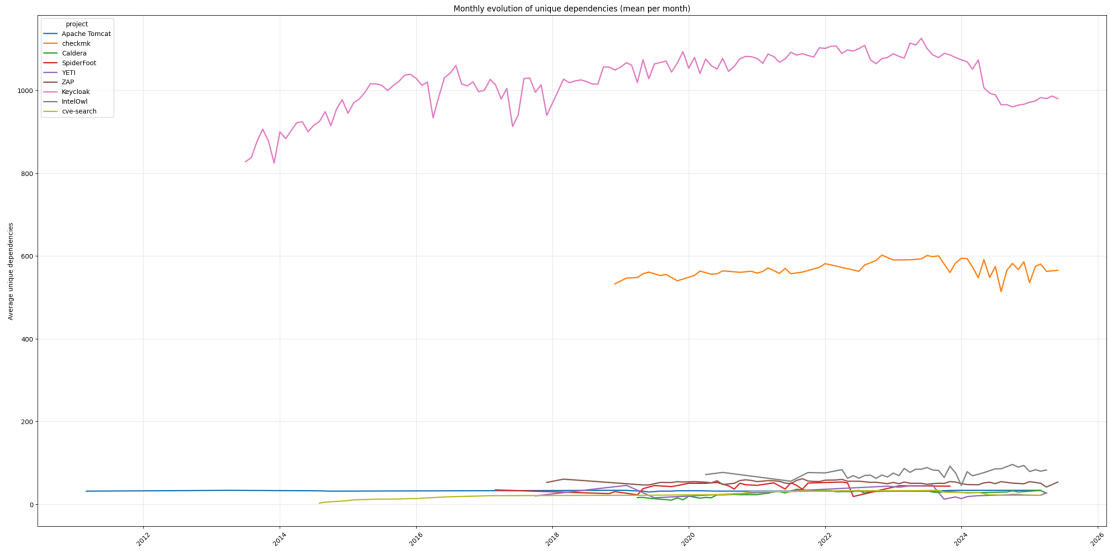
Subfigure 5.6a displays the monthly mean number of unique dependencies across all projects. The results reveal a highly skewed distribution, with projects such as Keycloak and checkmk exhibiting a significantly larger number of dependencies compared to other projects.

To mitigate the effect of such outliers and to better understand the trends in projects with smaller dependency footprints, subfigure 5.6b presents the monthly evolution for a filtered subset of projects. The figure reveals more fine-grained patterns: while some projects, such as Caldera and ZAP, show a steady increase in their dependency counts over time, others, including IntelOwl and YETI have a more varying number of dependencies.

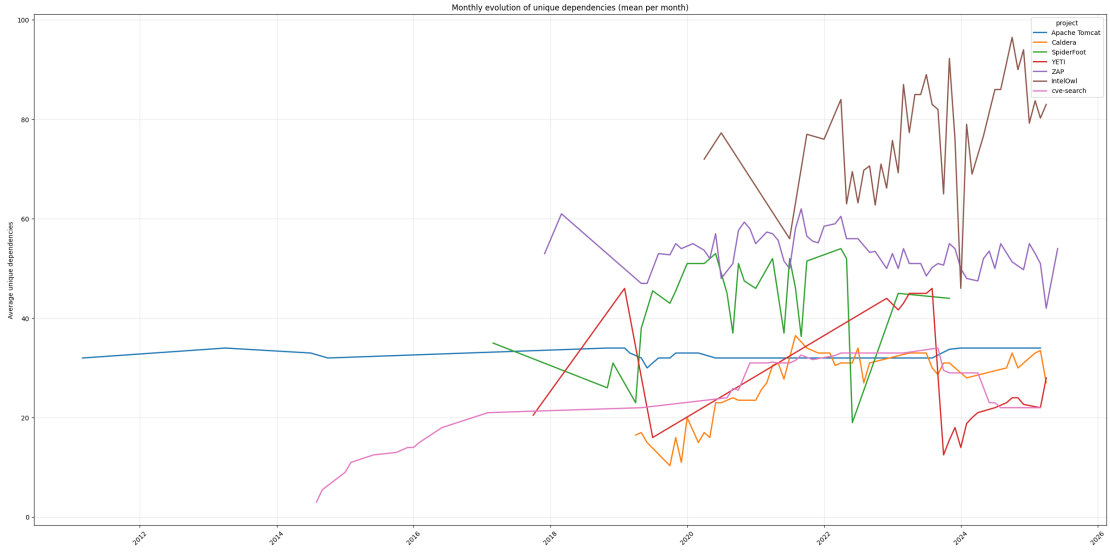
Subfigure 5.6c aggregates the same subset of projects at a yearly resolution, providing a broader temporal context. The yearly averages further reinforce the observed trends:

most projects display either a gradual increase or a consistent level of unique dependencies over time.

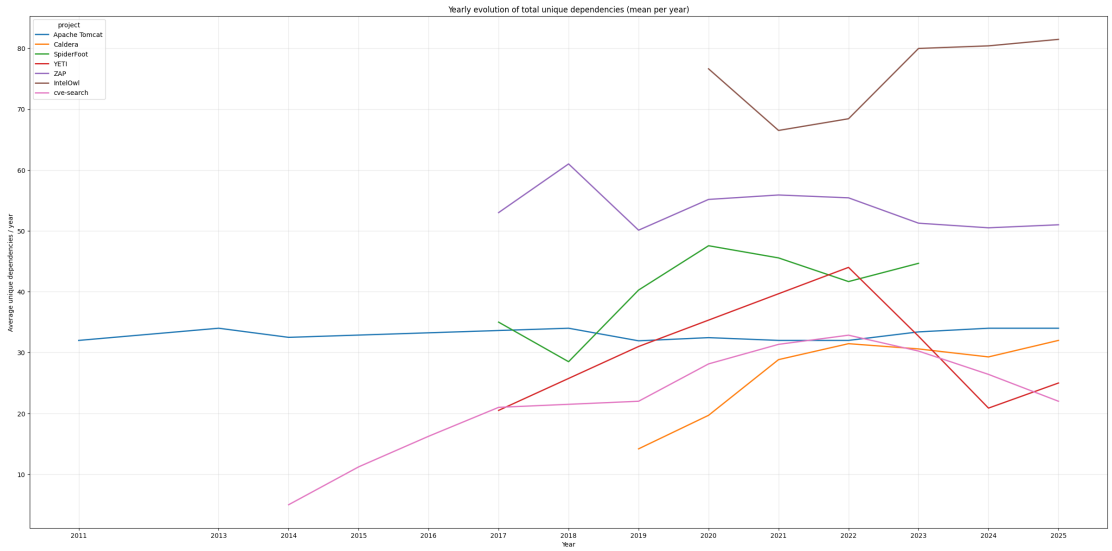
In conclusion, the descriptive analyses presented above provide a solid foundation for the discussion that follows in Chapter 6.



(a) Monthly evolution of the mean number of unique dependencies across all projects.



(b) Monthly evolution focusing on projects with fewer dependencies.



(c) Yearly evolution for the same subset of projects as in 5.6b.

Figure 5.6: Temporal evolution of the mean number of unique dependencies.

CHAPTER 6

Discussion

This chapter explores how project size influences dependency update practices and examines the impact of the number of manifest files on the diversity of dependencies, addressing RQ1 and RQ2. To facilitate comparison, projects are classified as *large* or *small* based on commit count and GitHub stars, as detailed in Table 4.1. Under this rule:

- **Keycloak** has approximately 28,000 commits and 27,000 stars, qualifying as a large project.
- **Apache Tomcat** has approximately 27,000 commits and 7,800 stars, qualifying as a large project based on commits.
- **ZAP** has approximately 10,000 commits and 13,600 stars, qualifying it as a large project based on the number of stars.
- **checkmk** has approximately 86,000 commits and 1,900 stars, qualifying as a large project based on the number of commits.
- **SpiderFoot** has approximately 3,700 commits and 14,500 stars, which qualifies as large by stars, despite the commit count.

Conversely, projects with fewer than 20,000 commits *and* fewer than 10,000 stars are classified as *small*. Under this criterion:

- **cve-search** has approximately 2,000 commits and 2,500 stars.
- **IntelOwl** has approximately 2,800 commits and 4,100 stars.
- **YETI** has approximately 3,400 commits and 1,900 stars.

- **Caldera** has approximately 4,400 commits and 6,200 stars.

By contrasting the update latencies observed in large projects (Keycloak, Apache Tomcat, ZAP, checkmk, and SpiderFoot) with those in small projects (cve-search, IntelOwl, YETI, Caldera), the aim is to determine whether project size, measured by commit count or star count, correlates with the timeliness of adopting new dependency releases—specifically, the mean update latency for each group as reported in Table 5.2.

First, Table 5.1 reveals that the projects classified as large differ substantially in both the number of commits affecting dependency files and the total number of dependency files. In particular, Keycloak represents an extreme case with 4,869 commits and 161 dependency files, whereas Apache Tomcat exhibits only 91 commits and 4 dependency files. This disparity suggests that Keycloak's dependency-management practices may diverge significantly from those of other large projects. Consequently, Keycloak will be analysed separately to determine the extent to which the sheer quantity of dependency files influences update latency and overall dependency-management behaviour.

6.1 RQ1

First, after observation of the Table 5.1, some crucial information can be gathered, which tells us that the projects classified as large vary extremely in the number of commits, where dependencies were changed and also in the number of files, where dependencies were managed. Keycloak, being a big outlier in number of commits and also number of files with 4,89 and 161 respectively, could have a vastly different outcome from Apache Tomcat, with a pretty opposite approach, with only 91 commits, and only 4 dependency files. Therefore, the analysis of what role the number of dependency files plays in the dependency management of projects is crucial. Moreover, Keycloak will be looked at separately, because even if it is arguably the biggest project, in contrast to other big projects, they have taken a completely different approach.

Taking a closer look at update latency, Figure 5.1 shows that most large projects experience a pronounced spike during the first year of their lifecycle. In contrast, Keycloak's average update latency remains relatively high from 2013 through 2020. This sustained latency is likely due to the substantial number of dependency files (161), which increases the complexity of coordinating updates and may necessitate the development of structured update mechanisms, such as the adoption of automated bots that handle dependency updates regularly. Notably, beginning in 2020, Keycloak demonstrates a clear downward trend in latency, reaching its lowest levels by 2025, suggesting that the project has optimised its dependency-management processes over time. This is also reflected in Figure 5.5, which clearly shows a shift toward a more frequent update cycle, a reduced MTBU, signalling a more frequent update cycle. With the values being very similar, this strongly suggests the use of an automated approach.

In contrast, cve-search manages only a single dependency file, which appears to facilitate rapid updates. As illustrated in heatmap 5.1, predominantly low latency values (depicted

in blue) indicate frequent and timely updates, with numerous intervals averaging a latency of 0.0 days. This suggests that, on average, cve-search adopts new dependency versions on the very day they are released, spanning a four-year time frame. However, from 2019 to 2023, a noticeable increase in update latency occurred, reflecting a reduced frequency of updates during this period. Correspondingly, the mean time between updates (MTBU) is also relatively high from 2019 to 2023, compared to the subsequent period from 2023 to 2025. Additionally, a significantly elevated MTBU observed from 2013 to 2016 may indicate that dependency updates were infrequent or less available during these years.

While analysing the projects, the question arose whether project size influences dependency version lag. This section aims to explore and answer that question. Version lag is defined as the number of releases by which a dependency is behind its latest available version at the moment of any commit that modifies at least one dependency file. Figure 5.4 illustrates the relationship between project age, project size (as previously defined), and version lag. This can be very clearly observed from Figure 5.3, where Keycloak leads in almost every year with the highest version lag.

From Figure 5.4, it is apparent that smaller projects tend to maintain their dependencies more closely to the latest available versions than larger projects. For example, IntelOwl exhibits consistently low version lag despite its relatively small team, likely because its recent inception reduces compatibility constraints with older dependency releases. In contrast, older projects that originated around 2011 (such as Keycloak) often need to support legacy code paths and may therefore delay upgrades to avoid breaking existing functionality. These larger, more mature codebases accumulate more significant version lag due to the need for extensive testing and integration effort before adopting new dependency versions. However, Apache Tomcat could manage a very low average version lag, which could be partly due to the small number of dependency declaration files it manages.

Notably, Figure 5.3 shows a major spike in version lag for the project SpiderFoot, which could be due to a potential version upgrade followed by a downgrade due to version incompatibility.

Furthermore, the median version-lag across all projects is 0.0 (Table 5.1), which implies that at least half of all dependency updates are merged without any delay. In other words, despite occasional outliers, the typical project in our sample keeps its manifest fully up-to-date.

6.2 RQ2

An important structural aspect of software projects that may influence dependency-management practices is the number of manifest files used to declare dependencies. The analysis reveals a strong observable correlation between the number of dependency-related files and the number of unique dependencies used within a project.

As shown in Figure 5.6a, Keycloak exhibits by far the highest number of unique dependencies every month, consistently exceeding 1,000 over several years. This project also maintains the highest number of manifest files (161, as shown in Table 5.1), suggesting that the scale and modularity of the codebase contribute directly to its extensive dependency graph. The presence of such a large number of declaration files likely reflects a highly modular architecture, potentially involving multiple submodules or services, each with its specific dependencies.

A similar pattern can be observed in checkmk, which has the second-highest number of dependency files (23) and also ranks second in terms of average unique dependencies per month. This alignment supports the hypothesis that projects with a greater number of manifest files tend to utilise a broader range of third-party libraries, possibly due to the distributed nature of dependency responsibility across components.

Interestingly, ZAP further reinforces this pattern. While its absolute number of dependencies is lower than Keycloak or checkmk, it still ranks among the top projects in terms of unique dependencies and holds the third-highest number of manifest files (16). This intermediate case highlights that even among projects with fewer total dependencies, the number of manifest files may still serve as a structural indicator of dependency diversity.

In contrast, projects such as Caldera and cve-search maintain way fewer than 10 dependency files each, and consistently show significantly lower average counts of unique dependencies over time. Figure 5.6b, which focuses on projects with smaller dependency sets, visualises this inverse relationship more clearly. The reduced number of files likely facilitates tighter control over dependency scope and may reflect a more monolithic or centralised architecture.

Overall, the evidence points to a meaningful relationship between the number of manifest files and the scale of a project's dependency landscape. While causality cannot be definitively established from correlation alone, it is plausible that more manifest files contribute to greater complexity in dependency management, both by increasing the total number of declared libraries and by decentralising the update process across components. This complexity may, in turn, influence other aspects of dependency management, such as version lag and update frequency, as discussed in earlier sections.

6.3 Conclusion of discussion

In summary, the comparison between large and small projects suggests that it is not necessarily the overall project size that determines update latency, but rather the number of dependency files that must be managed. As previously discussed, Keycloak's 161 dependency-related files pose significant coordination challenges, making manual updates impractical and underscoring the importance of automated solutions. In contrast, projects maintaining fewer than ten dependency files, regardless of their overall size, exhibited consistently low update latencies and minimal issues after their initial development phase. Furthermore, these projects also declare significantly fewer dependencies. This highlights

the central role of structural complexity, particularly in terms of manifest file distribution, in shaping dependency-update practices and maintaining timely adoption of new releases.

CHAPTER 7

Conclusion and Outlook

7.1 Summary of Main Findings

This thesis investigated how characteristics of open-source projects affect the way dependencies are updated over time. Specifically, two research questions were examined: (1) the relationship between project size and update latency, and (2) the influence of the number of manifest files in comparison to the number of declared dependencies. The findings show that update latency is not primarily determined by traditional size indicators such as commit count or GitHub stars. Instead, the number of dependency-related files plays a more significant role. Projects with a large number of dependency files, such as Keycloak, exhibited longer update latencies, especially in their early stages, due to the coordination overhead involved. In contrast, smaller projects with fewer dependency files managed to maintain short latencies and adopt new versions rapidly after their initial setup phase.

Regarding the relationship between the number of manifest files and the number of declared dependencies, a clear correlation was observed. Projects with a higher number of dependency files consistently declared significantly more unique dependencies. For example, Keycloak and checkmk, which had the most manifest files in the sample, also exhibited the highest number of unique dependencies. This trend suggests that structural complexity—reflected in the distribution of dependency declarations across multiple files—correlates with a broader and more diverse dependency landscape. Conversely, projects with fewer manifest files, such as Caldera and cve-search, tended to declare fewer dependencies overall. These findings reinforce the notion that architectural complexity, rather than project size alone, influences both the scale and maintainability of dependency usage in OSS projects.

7.2 Practical Implications

The results highlight the importance of reducing complexity in dependency management. Projects with a large number of dependency files should consider implementing structured update mechanisms, such as automated bots, to reduce update latency and prevent version drift. Tools that automate routine dependency updates on a weekly or even daily basis can significantly improve maintainability and reduce security risks. For new projects, it is advisable to keep the number of separate dependency files low unless modularisation is necessary.

7.3 Limitations

Despite the insights gained, this thesis has several important limitations that should inform the interpretation of its results:

1. **Sample size and representativeness.** We analysed nine open-source GitHub repositories spanning three ecosystems (Java, Python, JavaScript). While selected to cover a range of sizes and maturities, this sample cannot capture the full diversity of OSS projects (e.g. niche languages, enterprise-only repositories). As a result, these findings may not generalise to projects with different governance models or domain constraints.

2. **Proxy measures for project activity.** We used GitHub stars, contributor count, and commit volume as proxies for community engagement and project size. Each carries biases: stars reflect popularity rather than active usage; contributor counts can include bot accounts; and commit volume does not distinguish trivial edits from substantive work. Future studies might incorporate issue tracker activity or download statistics to obtain a more nuanced view.

3. **Reliance on manifest file changes.** The analysis assumes that every dependency update is recorded via edits to manifest files (pom.xml, package.json, requirements.txt, etc.). In practice, some teams update dependencies by vendor-checking or manual library imports without modifying these files, leading to undercounts.

4. **Incomplete metadata and scoring.** Criticality scores (OSSF) were unavailable for some projects (e.g. ZAP), and vulnerability data from public databases may lag behind actual disclosures. Consequently, our "average criticality" metric represents only a partial view of security risk and may understate or overstate true dependency exposure.

5. **Temporal snapshot.** This Thesis limited its commit history analysis to data available as of June 2, 2025. Projects under active development may exhibit different update patterns after that date. Longitudinal studies extending beyond our cutoff will be necessary to validate the persistence of observed trends.

7.4 Future Work

While this thesis focused on high-level dependency declarations in selected file types, future research could explore dependency evolution at a more granular level, for example by tracking actual import statements in the source code. This would allow for a more precise understanding of when and how dependencies are actively used and updated. Furthermore, the methodology presented here could be extended to additional programming ecosystems, such as Rust (`Cargo.toml`), .NET (`csproj`), or Ruby (`Gemfile`). While conceptually applicable, such an extension would require adapting or developing new parsers and normalisation routines tailored to the specific characteristics of those ecosystems.

Overall, the results of this thesis provide actionable insights for open-source maintainers and suggest directions for further automated tooling in the context of sustainable software maintenance.

List of Figures

List of Tables

Glossary

Git A version control system that helps keeping track of changes code and helps when collaborate with others. 9, 12, 14

PEP 517/518 A build-system independent format for source trees. 6

software aging The progressive degradation of software performance, reliability, or maintainability over time, typically caused by factors such as accumulated technical debt, outdated dependencies, architectural erosion, or changes in the operating environment. 1

technical debt The implied future cost (in extra work or reduced code quality) incurred by choosing a quick or sub-optimal implementation today instead of a more robust solution. 11, 16

Acronyms

API Application Programming Interfaces. 12, 14, 15

CLI Command Line Interface. 6

DSL Domain Specific Language. 6

JSON JavaScript Object Notation. 13

MTBU mean time between updates. 14–16, 21, 26, 27

OSS Open Source Software. 2, 5, 9, 10, 12, 21, 22, 31, 32

OSSF Open Source Security Foundation. 3, 9, 14–16, 18, 32

POM Project Object Model. 6, 13

PyPI Python Package Index. 6, 14

VCS Version Control System. 9

Bibliography

[CKK+18] Garvit Rajesh Choudhary, Sandeep Kumar, Kuldeep Kumar, Alok Mishra, and Cagatay Catal. Empirical analysis of change metrics for software fault prediction. *Computers Electrical Engineering*, 67:15–24, 2018.

[Fou25] The Apache Software Foundation. Introduction to the dependency mechanism, 2025. `https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html` [Accessed: (23.05.2025)].

[Fsc25a] Free and Open Source software community. Gradle, 2025. `https://docs.gradle.org/current/userguide/getting_started_dep_man.html` [Accessed: (23.05.2025)].

[Fsc25b] Free and Open Source software community. npm, 2025. `https://docs.npmjs.com/about-npm` [Accessed: (23.05.2025)].

[Fsc25c] Free and Open Source software community. pip, 2025. `https://pip.pypa.io/en/stable/topics/dependency-resolution/` [Accessed: (23.05.2025)].

[Has08] Ahmed E. Hassan. Automated classification of change messages in open source projects. In *Proceedings of the 2008 ACM Symposium on Applied Computing*, SAC '08, page 837–841, New York, NY, USA, 2008. Association for Computing Machinery.

[Has09] Ahmed E. Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, page 78–88, USA, 2009. IEEE Computer Society.

[KGO+18] Raula Gaikovina Kula, Daniel M. German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. Do developers update their library dependencies? *Empirical Softw. Engg.*, 23(1):384–417, February 2018.

[MPS08] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th International Conference on*

Software Engineering, ICSE '08, page 181–190, New York, NY, USA, 2008. Association for Computing Machinery.

[MV00] Audris Mockus and Lawrence G. Votta. Identifying reasons for software changes using historic databases. In *Proceedings of the International Conference on Software Maintenance (ICSM'00)*, ICSM '00, page 120, USA, 2000. IEEE Computer Society.

[Par94] David Lorge Parnas. Software aging. In *Proceedings of the 16th International Conference on Software Engineering*, ICSE '94, page 279–287, Washington, DC, USA, 1994. IEEE Computer Society Press.

[WCH+20] Ying Wang, Bihuan Chen, Kaifeng Huang, Bowen Shi, Congying Xu, Xin Peng, Yijian Wu, and Yang Liu. An empirical study of usages, updates and risks of third-party libraries in java projects. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 35–45, 2020.

Appendix

Appendix: Repository URLs

- **Keycloak**: `https://github.com/keycloak/keycloak` (last accessed 2 June 2025)
- **Apache Tomcat**: `https://github.com/apache/tomcat` (last accessed 2 June 2025)
- **ZAP**: `https://github.com/zaproxy/zaproxy` (last accessed 2 June 2025)
- **checkmk**: `https://github.com/Checkmk/checkmk` (last accessed 2 June 2025)
- **cve-search**: `https://github.com/cve-search/cve-search` (last accessed 2 June 2025)
- **IntelOwl**: `https://github.com/intelowlproject/IntelOwl` (last accessed 2 June 2025)
- **YETI**: `https://github.com/yeti-platform/yeti` (last accessed 2 June 2025)
- **SpiderFoot**: `https://github.com/smicallef/spiderfoot` (last accessed 2 June 2025)
- **Caldera**: `https://github.com/mitre/caldera` (last accessed 2 June 2025)

Appendix: Artefact Availability

All code, data, and supplementary materials for this thesis are publicly available at: `https://github.com/Konstiu/DependencyAnalysis`