Software Bill of Materials in Open Source C/C++ Projects: An Exploratory Study

Hangbo Zhang
University of Hawaii at Manoa
hangbo@hawaii.edu

Abstract

Understanding the dynamics of project dependencies is pivotal for assessing software maintenance and evolution. This study aims to investigate the update frequency, common library changes, and reasons behind dependency modifications in C/C++ projects. Leveraging version control history and software bill of materials (SBOM) data from GitHub repositories, we analyze release cycles, commit logs, and dependency manifests. Our methodology involves the selection of projects meeting language-specific criteria, retrieval of SBOM information using the GitHub REST API, and construction of a SQLite database for structured analysis. By addressing Research Questions (RQs) concerning update frequency, common library changes, and reasons for modifications, this research sheds light on software maintenance strategies, responsiveness to emerging issues, and alignment with industry standards.

Keywords: SBOM, C/C++, dependencies, Github REST API

1. Introduction

In today's software-centric landscape, comprehending the intricate interplay of dependencies within a project stands as a pivotal pillar for ensuring security, compliance, and transparency. As software ecosystems burgeon in complexity, the imperative for robust documentation of these dependencies has never been more pronounced. This is precisely where the Software Bill of Materials (SBOM) assumes its significance.

Originating from the manufacturing industry, where a bill of materials (BOM) served as an inventory roster of sub-assemblies and components within a parent assembly [8], the SBOM emerges as the software equivalent—a foundational construct vital for fortifying software supply chain security. Functioning as a nested inventory, the SBOM meticulously enumerates the software components comprising a project. It delineates these components, furnishing pertinent information and delineating the supply chain relationships interlinking them [10]. Essentially, the SBOM functions as a comprehensive ledger, cataloging all components, libraries, frameworks, and software artifacts incorporated within a project. Its utility lies in furnishing invaluable insights into the project's composition, empowering developers, stakeholders, and security professionals to navigate vulnerability management, licensing compliance, and risk mitigation strategies with informed precision.

However, despite the manifold benefits of SBOMs, their adoption trajectory remains somewhat subdued. A significant proportion of existing third-party software or components—whether open source or proprietary—lack accompanying SBOMs. Ideally, SBOM generation should commence early in the software development life cycle, gradually accruing enriched information through subsequent stages [20]. The absence of a unified package manager exacerbates this issue in the realm of C/C++ development, rendering SBOM generation an arduous task. Nonetheless, during our preliminary investigations, we discerned that GitHub's built-in SBOM feature seemed poised to provide insights into C/C++ projects, prompting us to delve deeper into its potential.

This document serves as an instructional blueprint for conducting an SBOM study on C/C++ projects hosted on GitHub. Through meticulous analysis, extraction, and documentation of dependencies and associated metadata, our objective is to address the following research questions (RQs):

RQ1: How frequently do developers update project dependencies?

Understanding the cadence of updates to project dependencies is pivotal for gauging project maintenance and evolution. Through scrutiny of version control history, commit logs, and release notes, we endeavor to discern the frequency with which developers integrate dependency updates. Factors such as the project's development tempo, the exigency of security patches, and the availability of new features or bug fixes in upstream dependencies influence update frequency. Furthermore, examining patterns of dependency updates over time offers insights into the project's overarching maintenance strategy and its responsiveness to emergent issues and advancements within the software realm.

RQ2: What are the common libraries that undergo changes?

Unveiling common libraries subject to changes provides valuable insights into evolving project requisites, technological trends, and community-driven development dynamics. By parsing through version control history and dependency manifests, we aim to pinpoint libraries frequently updated or replaced. These may encompass core dependencies like frameworks, utility libraries, or platform-specific components. Understanding the rationale behind these changes can illuminate the project's adaptability to evolving technologies, performance enhancements, or shifts in development paradigms.

RQ3: What are the reasons behind these changes?

Delving into the motives driving dependency changes furnishes contextual understanding of the project's development trajectory and decision-making ethos. By dissecting the rationale underlying dependency alterations, we aspire to glean deeper insights into the project's prioritization of stability, security, performance, and feature enhancements. Furthermore, this exploration sheds light on the project's alignment with industry standards and best practices, elucidating its strategic evolution within the software landscape.

2. Methodology

The methodology employed for establishing the database aimed to ensure systematic organization and comprehensive coverage of Software Bill of Materials (SBOM) information extracted from GitHub repositories. Through meticulous project selection criteria, data retrieval procedures, and database construction steps as shown in Figure 1, we laid the foundation for a robust dataset conducive to in-depth analysis and inquiry.

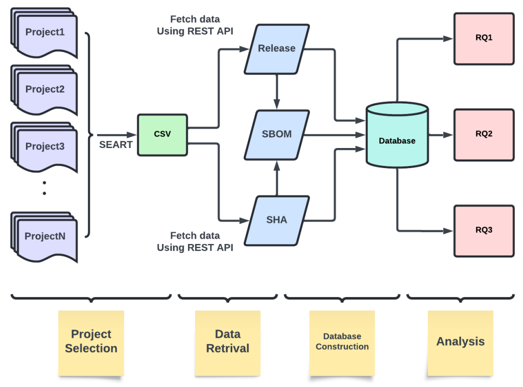

Figure 1: Methodology

2.1. Project Selection Criteria

The selection of projects for inclusion in the database followed specific criteria to ensure relevance and diversity. We are using the SEART[4], a GitHub Search Engine to sample repositories to use by using several combinations of selection criteria, to get the projects from GitHub based on the following criteria:

- **Language:** Projects written in C/C++ were considered to align with the specific language focus.
- **Release Count:** Projects with at least 100 releases were selected to capture a sufficient number of data points for analysis.
- **Not a Fork:** Only original projects, not forks, were included to maintain data integrity.
- **Recent Activity:** Projects with commits within the last 6 months were chosen to ensure the inclusion of actively maintained projects.

Following the application of these criteria, the resultant list of projects was downloaded as a CSV file for subsequent processing and analysis. By adhering to these rigorous selection criteria, we aimed to construct a robust dataset that encompasses a diverse array of C/C++ projects on GitHub, facilitating comprehensive insights into the evolution of SBOM within the context of software development.

2.2. Data Retrieval

To acquire Software Bill of Materials (SBOM) information spanning between releases, we employed the GitHub REST API as our primary data collecting method [6]. Initially, a GitHub Fine-grained personal access token [5] was generated to facilitate authentication and access to the GitHub database. Subsequently, we leveraged data from a CSV file generated in the preceding step, extracting repository owner and name details for each project. This enabled precise retrieval of project-specific information.

Utilizing the extracted repository owner and name, we constructed request URLs to retrieve all releases associated with each project, utilizing the GitHub REST API endpoint for releases [14]. Upon obtaining the release tags, we proceeded to gather comprehensive release information. However, variations in project structures necessitated additional scrutiny. In certain cases, the retrieved tag SHA lacked comprehensive commit information, prompting us to augment our approach by verifying tag SHAs through the REST API endpoint for tags [13]. This ensured acquisition of essential commit SHA details.

With a comprehensive collection of commit SHAs for each project's releases, arranged in descending chronological order, we transitioned to the final phase of this step. We facilitated the extraction of SBOM changes between consecutive releases by utilizing the GitHub REST API endpoint for dependency reviews [12]. The resultant SBOM differences were meticulously cataloged and saved as JSON files, serving as vital inputs for subsequent database construction and analyses.

Through this systematic approach, we ensured the accurate and systematic retrieval of SBOM information, facilitating informed decision-making and enhancing project management capabilities.

2.3. Database Construction

In order to facilitate systematic analysis and storage of Software Bill of Materials (SBOM) data extracted from GitHub repositories, we constructed a relational database using SQLite. This section outlines the process of database construction, detailing the schema design and data population procedures.

Schema Design: The database schema comprises four interconnected tables, each serving distinct roles in organizing SBOM-related information. Detailed structure and description is showed in Table 1.

1. `repo_table`: It serves as the cornerstone of the database, stroing metadata pertaining to individual repositories.

2. `diff_table`: It captures the commit SHA for all the releases for all projects.

Database Table	Columns	Description
repo_table	repo_name	name of repo
	repo_owner	owner of repo
	repo_url	link of repo
	commits	# of commits
	releases	# of releases
diff_table	diff_file	file name
	base_sha	earlier commit SHA
	base_tag	earlier tag
	base_date	date
	head_sha	newer commit SHA
	head_tag	newer commit tag
	head_date	date
	repo_name	forgien key
change_table	change_id	ID
	change_type	add/remove
	manifest	manifest file
	ecosystem	-
	name	name of dependency
	version	-
	package_url	-
	license	-
	source_repo_url	-
	scope	develop/run time
	vulnerability	-
	diff_file	forgien key
notes_table	release_tag	release version tag
	sha	commit SHA
	repo_name	forgien key
	release_note	notes
	commit_message	messages

Table 1: Database Structure

3. `change_table`: It stores detailed information about individual changes within SBOMs.

4. `notes_table`: It stores all the release notes and commit messages for each change.

Data Population Procedure: The database construction process involves iteratively populating the aforementioned tables with information extracted from SBOM-related files, CSV repositories, and GitHub API endpoints. The procedure encompasses the following steps. Establish a connection to the SQLite database using Python's sqlite3 module. Execute SQL commands to create the repo_table, diff_table, change_table and notes_table if they do not already exist. Read repository metadata from a CSV file containing details such as repository name, owner, URL, commits, and releases. Populate the repo_table with repository metadata extracted from the CSV file. Traverse through directories containing SBOM difference files (CSV format) for each repository. Extract SBOM difference details and insert them into the diff_table, linking each entry to its corresponding repository in the repo_table. Traverse through directories containing SBOM change JSON files for each repository. Extract SBOM change details and insert them into the change_table, linking each entry to its corresponding SBOM diff in the diff_table. Utilize the GitHub API to fetch release notes associated with each tag. If successful, extract the release note and commit message for the corresponding SHA. Insert the retrieved release notes and commit messages into the notes_table, linking each entry to its corresponding repository and tag. Implement error handling to manage exceptions such as missing or invalid data during the data population process. Commit all changes to the SQLite database and close the connection, ensuring data integrity and persistence.

By meticulously organizing SBOM-related information within a relational database and supplementing it with release notes from GitHub, we establish a structured framework for subsequent analysis and querying, enabling comprehensive insights into software dependencies, changes, and release information across GitHub repositories.

3. Data Analysis

In order to effectively address our research inquiries, we adhere to a meticulous preprocessing protocol applied to both the `change_table` and `diff_table` datasets. It is imperative to meticulously filter out extraneous data artifacts, such as lock files ('yarn.lock', 'package-lock.json'), which are automatically generated during software development processes and lack direct developer intervention. This preliminary filtration process ensures the integrity and relevance of our subsequent analyses. Consequently, we obtain two refined datasets: the `filtered_change_table` and `filtered_diff_table`, both devoid of non-essential data elements. These refined datasets serve as the foundational basis for our comprehensive analytical investigations, fostering a rigorous and scientifically sound approach to our research objectives

3.1. RQ1: How frequently do developers update project dependencies?

To address RQ1, we've implemented a systematic methodology to determine the release gap for each project, resulting in the creation of the `gap_table`. Here's how we achieved this:

Initially, we extracted all repository names from the `repo_table`. These repositories served as the foundation for subsequent data retrieval and analysis. Leveraging these repository names, we then retrieved the corresponding difference files from the `filtered_diff_table`. These files document variations between successive releases and are denoted numerically to indicate their position in the release sequence. For example, if a project named "project1" experiences dependency changes between the most recent and the previous release, the associated difference file would be labeled as `project1_diff0.json`. Similarly, `project1_diff1.json` would capture differences between the previous release and the one before it, and so forth.

By examining these difference files, we were able to ascertain the release gap for each project. The presence of changes within these files allowed us to determine the release gap, providing valuable insights into the temporal evolution of project dependencies. Finally, the calculated release gap data was compiled and inserted into the `gap_table`, enabling comprehensive analysis and exploration of temporal dynamics across projects.

3.1.1. Number of Releases with Changes The distribution of the number of releases with changes exhibits a pronounced right skewness. The majority of projects manifest minimal changes in their releases, with a median of 0. However, a subset of projects demonstrates a comparatively high number of changes, as evidenced by the maximum value of 613. Shown in Table 2.

Count	**Min**	**Q1**	**Median**	**Q3**	**Max**
number of changes for all projects					
314	0	0	0	8	613
number of changes without zero change projects					
144	1	3	9	22.5	613
release gaps					
3002	1	1	2	5	462

Table 2: Five-Number Summary

To refine our analysis, we initially filtered out repositories with zero releases containing changes, resulting in a dataset of 144 repositories. Subsequently, we applied the Interquartile Range (IQR) method to identify and remove outliers, resulting in the exclusion of 13 repositories. This meticulous approach aims to enhance the robustness of our analysis by focusing on repositories with substantial changes while mitigating the influence of extreme values.

Upon reevaluating the filtered dataset, the following insights emerge as shown in Table 3:

- **Dataset Size:** The filtered dataset comprises 131 repositories, providing a more targeted perspective on repositories with non-zero changes.
- **Descriptive Statistics:**
  - **Mean:** The mean number of changes is 12.41, indicating a moderate average across repositories.

Count	Mean	Std Dev	Min	Q1	Median	Q3	Max	IQR
number of changes without outliers								
131	12.41	12.62	1	3	8	16.5	51	13.5
release gap without outliers								
2672	2.59	2.26	1	1	2	3	11	2

Table 3: Fine detaited Five-Number Summary without outliers

- **Standard Deviation:** A wide standard deviation of 12.62 implies a considerable spread in the data, signifying variability in the number of changes.
- **Interquartile Range(IQR):** The IQR of 13.5 reflects the spread of the middle 50% of repositories, underscoring significant variability in the central distribution.

This refined analysis provides a scientifically rigorous exploration of the distribution of changes in repository releases. The meticulous filtering and outlier removal steps contribute to the reliability and interpretability of the observed patterns, offering valuable insights into the frequency and variability of changes across the examined repositories.

3.1.2. Release Gap (All Projects)

The distribution of release gaps, similar to the number of releases with changes, follows a right-skewed pattern. While the majority of projects exhibit relatively modest release gaps, a subset of projects demonstrates more extended intervals, with the maximum release gap reaching 462 as shown in Table 2.

To enhance the robustness of our analysis, we applied the Interquartile Range (IQR) method to identify and subsequently remove outliers from the release gap dataset. This meticulous curation resulted in a refined dataset, reducing its size from 3002 to 2672 instances.

Upon reevaluating the filtered dataset, the following insights emerge as shown in Table 3:

- **Dataset Size:** the filtered dataset now comprises 2672 instances, providing a focused perspective on repositories with distinct release gap characteristics.

- **Descriptive Statistics:**
  - **Mean:** The mean release gap stands at 2.59, indicating a moderate average duration between successive releases.
  - **Standard Deviation:** With a standard deviation of 2.26, there is a moderate level of variability in release gap values across the dataset.
  - **Interquartile Range(IQR):** The IQR, measured at 2.0, delineates the central spread of the middle 50% of release gap values, indicating a relatively concentrated distribution around the median.

Summary for RQ1: We observe a **right-skewed pattern** for both number of releases with changes and release gap, indicating that while the **majority of projects undergo minimal changes in each release with short intervals**, a subset of projects experiences more significant alterations. This variability underscores the diverse nature of dependency updates across repositories.

3.1.3. Findings

The analysis reveals that developers update project dependencies with varying frequencies. By examining the distribution of the number of releases with changes, we observe a right-skewed pattern, indicating that while the majority of projects undergo minimal changes in each release, a subset of projects experiences more significant alterations. This variability underscores the diverse nature of dependency updates across repositories.

Moreover, the investigation into release gaps elucidates the temporal dynamics of dependency changes. The distribution of release gaps follows a similar right-skewed pattern, with the majority of projects exhibiting relatively short intervals between successive releases. However, a subset of projects demonstrates more extended release gaps, suggesting differing update cadences among repositories.

By applying rigorous filtering and outlier removal techniques, the analysis enhances the reliability and interpretability of the observed patterns. The refined datasets provide focused insights into repositories with distinct characteristics in terms of the frequency and temporal distribution of dependency updates.

In conclusion, the findings offer valuable insights into the frequency of dependency updates among software projects, facilitating a deeper understanding of the release process and informing future research and development efforts in software engineering practices.

3.2. RQ2: What are the common libraries that undergo changes?

To address RQ2, we devised a methodical process to construct the `common_lib` table, leveraging data from both the `filtered_change_table` and `filtered_diff_table`. Here's a detailed outline of our approach:

We began by extracting the distinct repository names from the `filtered_diff_table`. For each project, we executed SQL queries to retrieve the top five most frequently added and removed libraries, storing them in separate lists along with their respective occurrences. Subsequently, we inserted this data into the `common_lib` table, which records the top five added libraries with their occurrences, the top five removed libraries with occurrences, and the overall top five most used libraries.

Additionally, we computed the total occurrence of the top five libraries for each change type across all projects. Furthermore, we identified the top five libraries for each change type that were most commonly used across all projects, along with the number of projects utilizing them.

Dependency	**Added**		**Removed**	
	Usage	Projects	Usage	Projects
(actions)/checkout	1746	125	947	94
()/upload-artifact	488	82	259	58
()/cache	274	41	111	23
()/setup-python	272	30	147	21
()/download-artifact	188	46	88	26
junit	144	3	98	3
semver	13	4	5	1
debug	11	2	6	1

Table 4: Occurrence of Dependencies Filtered

3.2.1. Library The analysis of Table 4 column “Added” highlights significant observations regarding the ’Added’ change type. The most frequently utilized dependency is ’actions/checkout,’ evidencing its pervasive usage across 125 distinct projects. The accumulated total occurrences of 1746 underscore its integral role in project development, suggesting a consistent need for updates across diverse development processes. ’actions/checkout’ is a GitHub Actions workflow that enables developers to checkout or clone the source code repository of their project into the execution environment of a GitHub Actions workflow. In other words, it’s a step in the workflow that ensures the relevant source code is available for subsequent tasks or actions in the workflow. the high occurrences and consistent updating of ’actions/checkout’ across projects suggest that it is a fundamental and frequently utilized part of the GitHub Actions workflows in the examined repositories. Developers often update it to benefit from new features, improvements, or to adapt to changes in their projects.

Following closely, ’actions/upload-artifact’ ranks as the second most utilized dependency, registering 488 occurrences distributed across 82 projects. This indicates a widespread adoption of this dependency, albeit at a slightly lower frequency than ’actions/checkout.’ ’actions/upload-artifact’ is another GitHub Actions workflow that enables developers to upload and share files or artifacts between different jobs within the same workflow, or even between different workflows. This action allows for the transfer of build outputs, test results, or any other files that are relevant to the workflow. the high occurrences of ’actions/upload-artifact’ across projects indicate that developers are frequently using this action to share important artifacts during their workflows. The total occurrences and the number of projects using it suggest that it plays a significant role in the continuous integration or deployment processes of these repositories.

Securing the third position is ’actions/cache,’ with 274 occurrences spanning 41 projects. This comparatively lower distribution implies a more specialized use, potentially within projects requiring specific caching functionalities. ’actions/cache’ is a GitHub Actions workflow that provides caching mechanisms to speed up workflows by storing and retrieving files or dependencies between workflow runs. This action is often used to cache dependencies, build artifacts, or other files that can be reused across multiple workflow runs to avoid redundant work. the high total occurrences of ’actions/cache’ across projects indicate that developers frequently utilize this action to optimize their workflows by caching and reusing certain files or dependencies. The number of projects using it suggests its widespread adoption in the GitHub Actions workflows.

Intriguingly, ’semver’ and ’debug’ exhibit notable total occurrences of 423 and 272, respectively. However, their lower usage across projects suggests a distinct pattern. These dependencies, characterized by higher occurrences and lower project adoption, may denote a frequent need for updates within specific development contexts. ’semver’ typically refers to Semantic Versioning, a versioning scheme used in software development to convey meaning about the underlying code changes. Semantic Versioning (SemVer) follows a three-part version number format: MAJOR.MINOR.PATCH. the high total occurrence

(423) of 'semver' across projects may suggest that developers are actively utilizing Semantic Versioning in their projects. This could indicate a commitment to following a standardized versioning approach, which helps communicate the nature of changes in software releases. The dependency 'debug' is a JavaScript debugging utility inspired by Node.js core's debugging technique. It is designed to work in both Node.js environments and web browsers. Developers commonly use this utility to facilitate the debugging process during software development. Considering its high total occurrence of 272, it appears to be widely utilized across projects. The frequent updates to this dependency suggest that developers actively maintain and enhance their debugging capabilities, emphasizing the importance of robust debugging practices in the software development life cycle.

Conversely, 'actions/setup-python' and 'actions/download-artifact' display a lower frequency of updates. Despite their higher project adoption (30 and 46 projects, respectively), the lower total occurrences imply a less dynamic updating pattern. This behavior may be indicative of dependencies integral to a project's core functionality, requiring infrequent updates.

Transitioning to Table 2 column "Removed", which pertains to the 'Removed' change type, 'actions/checkout' emerges once again as the predominant dependency. With 947 occurrences across 94 projects, its prevalence aligns with the trends observed in the 'Added' change type. The continuity of 'actions/checkout' in the 'Removed' change type further emphasizes its dynamic role, both introduced and retired across diverse projects.

The second-highest occurrence in the 'Removed' change type is attributed to 'actions/upload-artifact,' which records 259 instances across 58 projects. This mirrors its position in the 'Added' change type, suggesting a consistent lifecycle pattern.

Similar behavior is noted in 'semver,' 'debug,' and 'actions/setup-python,' which maintain comparable patterns to their 'Added' change type counterparts. This congruence in behavior reinforces the notion that certain dependencies exhibit consistent updating dynamics, irrespective of their introduction or removal. Moreover, dependencies such as '@types/node,' 'actions/cache,' and 'microsoft/setup-msbuild' exhibit similar behavior as those mentioned eariler in the added change type.

actions/checkout has a total of 32 releases range from Aug 1 2019 to Oct 17 2023 with some version of releases archived. The average of added of this dependency is 14 which indicate the project that using this dependency update regularly as the new releases comes out. 93 out of the 125 projects have the version less than 4.0 while the highest version is 4.1.1.

Ecosystem	**Usage (# of repos)**
actions	128
pip	31
npm	24
nuget	19
cargo	13
rubygems	8
gomod	5
maven	4
composer	1

Table 5: Ecosystems

actions/upload-artifact has a total of 23 releases range from Aug 1 2019 to Jan 2023. 65 out of the 82 projects have the version less than 4.0 while the highest version is 4.3.1.

semver has a total of 11 releases. 6 out of the 16 projects have the version less than 7.3.5 while the highest version is 7.6.0.

actions/cache has a total of 48 releases. 16 out of 41 project have the version less than 3 while the highest version is 4.0.0.

debug has a total of 37 releases. All of the 13 projects has the version higher than 4.3.1 while the highest version is 4.3.4 (9 projects out 13).

actions/setup-python has a total of 39 releases. 10 output of the 30 projects has the version less than 4 while the highest version is 5.0.0.

actions/download-artifact has a total of 20 releases. 8 out of the 46 projects has the version less than 3 while the highest version is 4.1.3. 27 out of the 46 project has the version 3.

3.2.2. Ecosystem GitHub Actions is an automation and workflow framework provided by GitHub. It allows you to define custom workflows, including automated tasks and processes, directly within your GitHub repository. These workflows are triggered by events, such as pushes, pull requests, issues, or scheduled events. GitHub Actions is often used for continuous integration (CI), continuous deployment (CD), testing, and other automation tasks.

npm, or Node Package Manager, is a package manager for JavaScript programming language. It is the default package manager for Node.js, a popular runtime for server-side and networking applications. npm is used to install, share, and manage packages (software libraries) in the Node.js ecosystem.

NuGet is a package manager for the Microsoft development platform, primarily used with the .NET framework and related technologies. It helps developers discover, install, and manage libraries, frameworks, and tools necessary for .NET development. The term "NuGet" is often used to refer to both the package manager tool and the repository where packages are hosted.

RubyGems is the package manager for the Ruby programming language. It serves as a tool for distributing and managing Ruby libraries and programs. RubyGems simplifies the process of installing, updating, and managing dependencies in Ruby projects.

Cargo is the package manager and build system for the Rust programming language. It is an integral part of the Rust ecosystem, providing tools for managing dependencies, building projects, and handling various aspects of the development process.

pip is the package installer for Python, and it plays a crucial role in managing Python packages. It simplifies the process of installing, upgrading, and managing third-party libraries and tools for Python development.

gomod is the module system introduced in Go (or Golang) starting from version 1.11 to manage dependencies in a Go project. It provides a way to define and manage dependencies at the module level, making it easier to build and share Go code.

Apache Maven is a widely used build automation and project management tool in the Java ecosystem. It simplifies the process of building, managing, and organizing Java projects. Maven uses a declarative XML-based configuration to define project settings, dependencies, and build phases.

Summary for RQ2: The analysis shows widespread use of GitHub Actions workflows for CI/CD, testing, and automation. Additionally, the presence of popular libraries such as "junit" and "semver" underscores their critical role in software development. Frequent updates in key dependencies reflect dynamic CI/CD pipelines, emphasizing ongoing workflow optimization.

3.2.3. Findings The majority of dependencies belong to the "actions" ecosystem, indicating a prevalent usage of GitHub Actions workflows in the examined repositories. This ecosystem includes essential workflows such as "actions/checkout" and "actions/upload-artifact," which facilitate various aspects of CI/CD, testing, and automation processes within GitHub repositories.

Versatility of Package Managers: The presence of diverse package managers such as npm, NuGet, RubyGems, Cargo, pip, and gomod underscores the multi-language and multi-platform nature of the analyzed projects. Each package manager caters to specific programming languages and ecosystems, reflecting the diverse technological landscape of modern software development.

Popular Libraries and Frameworks: Certain dependencies, such as "junit" and "semver," are widely used across projects, indicating their importance in software development. These libraries provide essential functionalities for testing, versioning, and other critical aspects of software engineering.

Dynamic Updating Patterns: Dependencies like "actions/checkout" and "actions/upload-artifact" exhibit high frequencies of updates across a significant number of projects. This dynamic updating pattern suggests a continuous integration and deployment (CI/CD) pipeline, where developers frequently modify and optimize their workflows to streamline the development process.

Specialized Use Cases: Dependencies like "actions/cache" and "actions/setup-python" demonstrate a more specialized use, with fewer occurrences but significant project adoption. These dependencies are likely tailored to specific project requirements, such as caching dependencies or setting up Python environments, highlighting the diverse needs of software projects.

Overall, the analysis provides valuable insights into the common libraries and ecosystems driving dependency changes in software projects. Understanding these patterns is essential for developers to make informed decisions about dependency management, ensure project stability, and optimize development workflows for enhanced productivity and efficiency.

3.3. RQ3: What are the reasons behind these changes?

To address RQ3, we initiated by extracting release notes and commit messages for each release of every project. Subsequently, we conducted preprocessing on the messages by eliminating empty messages, URLs, commit SHAs, and special characters, including numbers. We leveraged the STOPWORDS from WordCloud [9] supplemented with additional words to filter out non-essential data. Utilizing functionalities from NLTK [18], including word_tokenization, WordNetLemmatizer, and ngrams, we tokenized all messages, lemmatized verbs, and generated unigrams, bigrams, and trigrams.

This meticulous preprocessing pipeline ensures

Table 6: Top 10 frequently occurring unigrams, bigrams, and trigrams in release notes/commit message

Unigram	Count	Percentage	Bigram	Count	Percentage	Trigram	Count	Percentage
fix	26103	3.78%	arm dts	986	0.14%	fix memory leak	616	0.09%
add	8092	1.17%	memory leak	892	0.13%	version bump package	554	0.08%
use	6601	0.96%	fix error	844	0.12%	fix use free	427	0.06%
test	4514	0.65%	add miss	740	0.11%	fix refcount leak	335	0.05%
support	4203	0.61%	fix memory	690	0.10%	fix error handle	305	0.04%
update	3973	0.57%	use free	602	0.09%	fix data race	297	0.04%
error	3639	0.53%	return value	589	0.09%	drm amd display	280	0.04%
change	3489	0.50%	version bump	587	0.08%	arm dts qcom	280	0.04%
check	3345	0.48%	fix miss	570	0.08%	data race around	275	0.04%
release	3307	0.48%	error handle	568	0.08%	null pointer dereference	241	0.03%
others	623777	90.27%	others	683974	98.98%	others	687431	99.48%

the extraction of pertinent information from release notes and commit messages, facilitating comprehensive analysis and interpretation to address RQ3. This approach enables us to uncover underlying trends, patterns, and reasons behind the observed changes, providing valuable insights into the evolution of the software projects.

The `notes_table` contains of 2824 entries each has release notes and commit messages. We have generated the word-cloud of the most frequent words that are provided by the developers in terms of the reason of dependency change. To verify the finding, we have conducted a manual review on random selected 339 projects with 95% confidence level and 5% margin of error.

3.3.1. Unigram

The most frequently occurring unigram, "fix," emerges prominently within the corpus, with a substantial count of 26,103 instances, representing 3.78% of the total unigram count. This statistical prevalence underscores a significant emphasis on issue resolution and bug fixing within the software projects under analysis.

Moreover, beyond "fix," other notable unigrams such as "add," "use," "test," and "support" are observed, each contributing distinctively to various facets of software development and maintenance. For example, "add" connotes the incorporation of novel functionalities, while "use" implies resource or tool utilization. Similarly, "test" and "support" underscore the critical roles of testing procedures and assistance provision within the project ecosystem.

While the majority of the unigrams predominantly pertain to specific project issues or features, intriguingly, certain instances do allude to dependency alterations. Noteworthy examples include references to dependency updates or introductions, such as "*Fixes a linker error on iOS due to a new dependency introduced with a bgfx update*"[3] and "*Add deb package for Ubuntu 22.04*"[15]. These occurrences illuminate the interplay between software modifications and the underlying dependencies, suggesting a nuanced understanding of the software's evolution within its broader ecosystem.

3.3.2. Bigram

Among bigrams, the recurrent occurrence of "arm dts" stands out prominently, with a count of 986 instances, representing 0.14% of the total bigram count. This statistical prominence suggests a focused attention on ARM architecture and device tree sources within the software projects under examination.

Additionally, notable bigrams such as "memory leak," "fix error," and "add miss" are identified, each emblematic of common challenges and actions encountered throughout the software development and maintenance lifecycle. For instance, the presence of "memory leak" signifies a persistent technical hurdle, while "fix error" implies concerted efforts to rectify software bugs or oversights.

While the majority of the identified bigrams are intricately linked to specific project issues or actions, for example, "*ARM: dts: spear1340: Update serial node properties*"[19], "*Fixed a memory leak in ppdOpen*"[1], it is noteworthy that certain instances do allude to dependency changes. For example, within the same release notes, references are made to dependency adjustments, such as "*bpf_doc: Fix build error with older python versions*"[19] and "*Fixed builds when there is no TLS library*"[1]. These mentions underscore the interconnected nature of software modifications and external dependencies, highlighting the consequential impact of dependency management on project stability and functionality.

3.3.3. Trigram

In the trigram analysis, "fix memory leak" emerges as the most frequent, with a count of 616 instances, representing 0.09% of the total trigram count. This statistical prominence underscores the critical importance of addressing memory-related issues within the software projects under scrutiny.

Additionally, other notable trigrams such as "version bump package" and "fix use free" are identified, indicating activities closely associated with version management and resource utilization, respectively. The presence of these trigrams sheds light on the multifaceted nature of software development and maintenance, encompassing not only technical troubleshooting but also strategic resource allocation and optimization.

Moreover, phrases like ”fix refcount leak” and ”drm amd display” are observed, highlighting specific technical challenges and components within the software projects. These trigrams offer insights into the diverse array of issues encountered during the development and maintenance phases, further emphasizing the complexity inherent in modern software ecosystems.

While the primary focus of trigrams may not inherently relate to dependency changes, intriguingly, within the same release notes, mentions of dependency adjustments are intertwined with these technical terms. For instance, while the trigram ”fix memory leak” primarily addresses an internal project issue such as ‘*fix: memory leak when drop db...*”[17], the accompanying mention of ”build: update taospy and taos-ws-py version” within the same release note indicates a concurrent adjustment in dependencies. Similarly, the trigram ”version bump package” directly correlates with dependency modifications, as evidenced by instances such as “*Rust bindings version bump for package 1.3.26*” [2].

Summary for RQ3: The analysis reveals a predominant focus on issue resolution and bug fixing within the software projects. Additionally, while most linguistic patterns pertain to internal project issues, sporadic mentions of dependency changes underscore the interconnected nature of software modifications and ecosystem management.

3.3.4. Findings Analyzing the percentages associated with each n-gram provides insights into their relative importance within the corpus. Unigrams like “fix” and “add” demonstrate relatively high percentages, indicating their significance in the overall text. Conversely, certain bigrams and trigrams exhibit lower percentages, suggesting their occurrence in less common or specific contexts. For example, phrases like “arm dts” and “fix memory leak” may occur less frequently but carry significant importance within the context of the software projects.

In conclusion, the analysis of the top unigrams, bigrams, and trigrams sheds light on the underlying reasons behind dependency changes within the software projects. The prevalence of terms such as “fix” and “add” suggests a primary focus on resolving issues and introducing new features. Common phrases like “arm dts” and “fix memory leak” highlight specific areas of concern, indicating a combination of functional enhancements and technical optimizations.

Furthermore, the occurrence of phrases like “memory leak” and “version bump package” signifies platform-specific modifications and version management activities, respectively. These findings suggest a multifaceted approach to dependency changes, encompassing both functional improvements and technical refinements to enhance software quality and performance.

Overall, the analysis underscores the diverse nature of dependency changes within the projects, encompassing a wide range of motivations and objectives. By identifying and understanding the underlying reasons behind these changes, stakeholders can make informed decisions to optimize software development processes, prioritize development efforts, and ensure the long-term sustainability and reliability of the software projects.

4. Related Work

Dependencies: Wei et al. [16] highlighted the challenge posed by the lack of a standard package format and unified package manager, leading to a scarcity of methods and tools for extracting Software Bill of Materials (SBOM) from large-scale C/C++ repositories. To address this issue, they developed CCScanner, a tool designed to define dependencies in C/C++ projects. Similarly, Ling et al. [7] investigated Centris, a state-of-the-art software composition analysis (SCA) technique tailored for the C/C++ ecosystem, focusing on deriving Third Party Library (TPL) dependencies. Additionally, Yoonjong et al. [11] introduced Cneps, a precise approach for analyzing dependencies among reused components, which addresses challenges such as indistinguishable files and duplicated components.

SBOM: The significance of SBOMs in enhancing software supply chain (SSC) security through transparency, accountability, traceability, and security has been emphasized by Boming et al. [20]. Their study, comprising 17 interviews and 65 surveys with SBOM practitioners, underscores the importance of SBOMs in addressing security challenges within the software supply chain.

While numerous studies have investigated C/C++ TPL dependencies and SBOMs in general, further exploration into the specific domain of SBOM for

C/C++ is warranted.

5. Conclusion

The findings from this study shed light on various aspects of dependency management practices and their implications for software development. Through the exploration of three research questions, we gained insights into the frequency of dependency updates, common libraries undergoing changes, and the reasons behind dependency changes.

Firstly, our analysis of the frequency of dependency updates revealed a wide spectrum of update patterns across software projects. While some projects exhibit minimal changes and stable release intervals, others demonstrate dynamic update cycles with frequent releases. Understanding these patterns is crucial for developers to optimize resource allocation, balance stability with innovation, and ensure the overall health of software projects.

Secondly, our investigation into common libraries undergoing changes highlighted the prevalence of GitHub Actions workflows and diverse ecosystems of dependencies. GitHub Actions workflows play a pivotal role in enabling automation and CI/CD processes, while the heterogeneous nature of dependency ecosystems underscores the diversity of tools and technologies used in software development.

Finally, our examination of the reasons behind dependency changes uncovered common themes and patterns in release notes and commit messages. By analyzing textual data, we identified prevalent terms and phrases indicative of bug fixes, feature enhancements, and performance optimizations. Understanding these reasons can inform developers' decisions regarding prioritization, resource allocation, and overall software quality improvement efforts.

In conclusion, this study provides valuable insights into dependency management practices and their implications for software development. By leveraging these insights, developers can make informed decisions to optimize their development workflows, ensure project stability, and deliver high-quality software products to end-users.

6. Future Work

While this study provides valuable insights into dependency management practices, several avenues for future research exist. Firstly, further exploration into the impact of dependency updates on software project stability and performance could provide valuable insights for developers. Understanding the trade-offs between frequent updates and project stability can inform dependency management strategies.

Additionally, investigating the adoption of emerging technologies and frameworks in software projects could offer insights into evolving trends in software development. Analyzing trends in dependency usage and adoption rates across different ecosystems can provide valuable insights into the evolution of software development practices over time.

Furthermore, exploring the role of automated dependency management tools and techniques, such as dependency analysis tools and automated update mechanisms, could help streamline the dependency management process and mitigate common challenges faced by developers.

Overall, future research endeavors aimed at understanding and improving dependency management practices have the potential to significantly impact the efficiency, reliability, and sustainability of software development processes. By addressing these challenges, researchers and practitioners can contribute to the continued advancement of the field and the delivery of high-quality software products to end-users.

References

[1] Apple. *Release v2.3.0.* 2019. URL: https://github.com/apple/cups/releases/tag/v2.3.0.

[2] AWS. *Release v1.3.27.* 2022. URL: https://github.com/aws/s2n-tls/releases/tag/v1.3.27.

[3] BabylonJS. *Release 0.4.0-alpha.45.* 2021. URL: https://github.com/BabylonJS/BabylonReactNative/releases/tag/0.4.0-alpha.45.

[4] Ozren Dabic, Emad Aghajani, and Gabriele Bavota. "Sampling Projects in GitHub for MSR Studies". In: *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021.* IEEE, 2021, pp. 560–564.

[5] Github. *Creating a fine-grained personal access token.* 2024. URL: https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/managing-your-personal-access-tokens#creating-a-fine-grained-personal-access-token.

[6] GitHub. *Create integrations, retrieve data, and automate your workflows with the GitHub REST API.* 2024. URL: https://docs.github.

com/en/rest?apiVersion=2022-11-28.

[7] Ling Jiang et al. "Third-Party Library Dependency for Large-Scale SCA in the C/C++ Ecosystem: How Far Are We?" In: *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2023, pp. 1383–1395.

[8] Jianxin Jiao et al. "Generic bill-of-materials-and-operations for high-variety production management". In: *Concurrent Engineering* 8.4 (2000), pp. 297–321.

[9] Andreas Mueller. *WordCloud for Python*. 2020. URL: https://amueller.github.io/word_cloud/.

[10] Éamonn Ó Muirí. "Framing software component transparency: Establishing a common software bill of material (SBOM)". In: *NTIA, Nov* 12 (2019).

[11] Yoonjong Na et al. "Cneps: A Precise Approach for Examining Dependencies among Third-Party C/C++ Open-Source Components". In: (2024).

[12] Github. *Use the REST API to interact with dependency changes*. 2024. URL: https://docs.github.com/en/rest/dependency-graph/dependency-review?apiVersion=2022-11-28#get-a-diff-of-the-dependencies-between-commits.

[13] Github. *Use the REST API to interact with tag objects in your Git database on GitHub*. 2024. URL: https://docs.github.com/en/rest/git/tags?apiVersion=2022-11-28.

[14] Github. *Use the REST API to create, modify, and delete releases*. 2024. URL: https://docs.github.com/en/rest/releases/releases?apiVersion=2022-11-28.

[15] Rigaya. *Release 7.02*. 2022. URL: https://github.com/rigaya/QSVEnc/releases/tag/7.02.

[16] Wei Tang et al. "Towards understanding third-party library dependency in c/c++ ecosystem". In: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 2022, pp. 1–12.

[17] taosdata. *Release 3.2.0.0*. 2023. URL: https://github.com/taosdata/TDengine/releases/tag/ver-3.2.0.0.

[18] NLTK Team. *Natural Language Toolkit*. 2023. URL: https://www.nltk.org/.

[19] Xanmod. *Release Linux 5.17.2-xanmod1*. 2022. URL: https://github.com/xanmod/linux/releases/tag/5.17.2-xanmod1.

[20] Boming Xia et al. "An Empirical Study on Software Bill of Materials: Where We Stand and the Road Ahead". In: *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 2023, pp. 2630–2642. DOI: 10.1109/ICSE48619.2023.00219.