Thesis Dissertation

CROSS-ECOSYSTEM PACKAGES ACROSS SOFTWARE ECOSYSTEMS

Spyros Droussiotis

University Of Cyprus

COMPUTER SCIENCE DEPARTMENT

May 2026

UNIVERSITY OF CYPRUS DEPARTMENT OF COMPUTER SCIENCE

Cross-ecosystems packages across software ecosystems

Spyros Droussiotis

Supervisor
Eleni Constantinou

Thesis submitted in partial fulfilment of the requirements for the award of degree of bachelor's in computer science at University of Cyprus.

May 2026

Code of Conduct

I declare that this dissertation and any accompanying software are my own original work, conducted in full compliance with the University of Cyprus Code of Conduct for Research (https://www.ucy.ac.cy/legislation/kodikas-deontologias-kai-politikes/), and with full accountability on my part for the accuracy, integrity, and validity of all content."

Acknowledgments

I would like to thank my supervisor, Professor Eleni Constantinou, for her continuous guidance. Her mentorship has provided me with all the insight need to accomplish this research

Abstract

Cross-ecosystem packages are software components that are distributed across multiple software package ecosystems, and a number of software projects rely on their functionality. Although existing literature was focused on the evolution of technical and social aspects of these packages, less attention has been given to how dependent software projects actually use them at source code level. While dependent packages provide metadata about their dependencies, this information does not fully explain how these packages interact with the cross-ecosystem packages in practice. This thesis investigates how dependent packages use cross-ecosystem packages by analyzing method usage in source code. A dataset of 10,369 dependent packages from npm and RubyGems was collected and examined using static source code analysis. The results identify the reference forms used to access cross-ecosystem packages and reveal differences between method usage frequency, method diversity, and method co-usage. This study provides useful insight on how dependents employ cross-ecosystem packages' functionality and highlights differences in usage across packages and ecosystems.

Contents

Section 1

Introduction

1.1 Background and Motivation

In the field of software development, several elements support and accelerate the development process, including software libraries, frameworks, and software ecosystems. The term software ecosystem refers to interactions among multiple actors within a shared technological platform leading to the development, evolution, and maintenance of software projects and services [8]. These ecosystems are not only collections of projects, but also include tools, platforms, and communities. Examples of software ecosystems include package managers for various programming languages such as npm for JavaScript, RubyGems for Ruby, and PyPI for Python. Software ecosystems have attracted significant attention because of value they provide; they enable faster development and collaboration between developers by interacting around a shared platform.

Software ecosystems are commonly viewed as isolated entities with their own boundaries and development environments, which often leads to projects being developed and distributed in a specific language. However, as software development has become increasingly multilingual [19], some projects may extend to more than one software ecosystem; these projects are referred to as cross-ecosystem packages [1]. As software development has become increasingly interconnected, the importance of these packages has grown significantly, making them a key factor in modern software development. For this reason, studying these

packages is vital for understanding how software dependencies operate across ecosystem boundaries.

Prior research has explored both the technical and social aspects of cross-ecosystem packages [2,3], including patterns of shifting programming languages during their development and the influence of developers on the evolution of these projects. However, one aspect that remains underexplored is how these widely used cross-ecosystem packages are utilized by the packages that depend on them. Understanding this aspect is important because it can provide a clearer picture of their impact on modern software development and reveal the actual role of cross-ecosystem packages in dependency networks.

1.2 Research Objective and Questions

The objective of this thesis is to navigate how cross-ecosystem packages are being utilized by their dependents. The study aims to uncover how the cross ecosystems packages are being used by their dependents to understand the role these packages play in dependent source code.

To address this objective, the thesis is guided by the following research question.

RQ: How do the dependents use the cross-ecosystem packages?

The need for this research stems from the fact dependency metadata is accessible, but it does not clearly state how actively a cross-ecosystem package is used. The metadata can show that a relationship exists but does not explain how the package is accessed inside the source code.

Specifically, the research question aims on identifying how cross-ecosystem packages are referenced, how frequently their methods are invoked, and how diverse their detected method usage is. In addition, the study examines co-usage, which refers to cases where methods from the same package appear within the same code block.

1.3 Methodology Overview

To address these research questions, this thesis adopts an empirical approach based on the analysis of cross-ecosystem packages and their dependents. The required data were collected from relevant software ecosystems to examine how cross-ecosystem packages are utilized in practice. Through this analysis, the aim is to discover how these packages are used to better understand their role.

This thesis was based on 338 cross-ecosystem packages across the ecosystems of npm and RubyGems. These ecosystems were selected as they provided the dependency metadata needed for this study. The initial dataset [5] was based on Libraries.io, which is a free service where it collects open-source packages [4]. Once the metadata was collected, the process of extracting the dependents source code followed. Using static source code analysis, 10,369 dependent packages were examined, and 457,256 recorded method usages were extracted.

1.4 Thesis Structure

The thesis begins in Section 2, introducing fundamental background information to understand the research topic. Section 3 presents related work on covering software ecosystems, cross-ecosystem packages, and software dependencies. Section 4 describes the methodology in detail, and Section 5 presents the results of the methodology. In section 6 the results are discussed and finally Section 7 concludes the thesis and outlines possible directions for future work.

Section 2

Background

2.1 Software ecosystem definition

Software Ecosystems (SECOs) have attracted vast amounts of attention due to their significant role in software development, but there is not a universally accepted definiton of what a software ecosystem is. Jansen et al [7] referred to a SECO as “a set of businesses functioning as a unit and interacting with a shared market for software and services, together with the relationships among them” while on the other hand Lungu et al. [9] defined SECOs as “A software ecosystem is a collection of software projects that belong to an organization and are developed in parallel by the organization” which covers a more technical aspect. In their systematic literature review, Manikas et al. [8] observed that many published studies do not include a definition of what a software ecosystem is while those that do, present differentiated definitions. Despite the differences, the authors concluded that the definitions, even though different, all had three elements in common. First, they referred to a form of “common software” within an ecosystem. Second, they view an ecosystem as a business, and the third is that the SECOs have “connecting relationships” among the entities within them. After this observation, Manikas et al [8] concluded their own definition for software ecosystem “as the interaction of a set of actors on top of a common technological platform that results in a number of software solutions or services”.

Overall, a software ecosystem can be described as a shared technological platform where software projects are developed, maintained and evolved. It encourages collaboration amongst different actors in the same environment and operates as a business-oriented network in the market that delivers software and services.

Even though software ecosystems are defined in different ways in the literature, they share common features that support their growth, promote collaboration and encourage innovation. They possess characteristics of natural ecosystems like mutualism, communalism and symbiosis, thus explaining the term software ecosystems. Their architectural infrastructure provides security, reliability, evolution management and interface stability, thereby creating a stable and supportive environment for development. [12]

Additionally, some software ecosystems usually follow the open-source development model where the source code is freely available for anyone to view, modify and share. This model enhances collaboration among developers within the community, enabling them to develop and maintain software components together. It also increases innovation by allowing these components to be extended and improves transparency as every modification or addition is publicly visible. [12]

Moreover, SECOs have internal and external developers, both of whom contribute to the software system. The distinction between them lies in whether they are part of the platform team; they both work to achieve a common target, but each benefits differently from the system. Nevertheless, their interaction is of great importance as it shapes the platform's evolution. [10] Another important feature of software ecosystems is the existence of common technological platforms. They provide a shared technical foundation on projects within the ecosystem and in some cases, they have multiple layers. The common platform supports reusable components, allows the expansion of architecture, and often includes common tools and services [10].

2.2 Package Managers

With the rapid evolution of software development, various types of software ecosystems have emerged, each playing a unique role in supporting development. These ecosystems range from open-source to domain-specific and application-oriented ecosystems, each depending on their own communities and technologies. One of the most important categories is package management platform ecosystem, they are package-based ecosystems who are often open-source focusing on distribution, reuse and management of software packages.

Package management platforms are software ecosystems that provide repositories for software packages and automate the process of installing and managing them. Some examples are npm for JavaScript, PyPI for Python, RubyGems for Ruby and Maven Central for Java where each manages packages for their respective programming languages. A systematic literature review (SLR) from Hou et al [11] has referred to package managers as a core mechanism that automatically manages the installation and upgrade of a package on an operating system. The significance of these ecosystems is that they offer reusable packages which developers can integrate into their projects, making software development faster and more efficient. Beyond hosting packages, these ecosystems offer collaboration through common tools and communities. Furthermore, these ecosystems are usually treated as separate environments where their boundaries do not overlap, and their packages are developed, maintained and evolved within a single ecosystem.

2.3 Definition of cross-ecosystem packages

While most packages are limited to a single ecosystem, some extend across the boundaries of their original ecosystem and are distributed across multiple software ecosystems. These packages are known as cross-ecosystem packages. Constantinou et al. [1] identified such packages by observing whether packages in at least two software ecosystems shared the same GitHub repository, considering this to represent the same cross-ecosystem package.

Unlike regular packages, cross-ecosystem packages create connections between separate ecosystems. Another key difference is that a regular package is typically developed and maintained within a single ecosystem, whereas a cross-ecosystem package may be maintained for distribution across multiple ecosystems. In addition, a cross-ecosystem package has a broader reach as it exists in multiple ecosystems which leads to being available for a larger number of developers.

The existence of cross-ecosystem packages can be defined by the need to utilize same or similar functionalities available across multiple programming languages. Moreover, developers probably prefer their package to be accessible regardless of the ecosystem where a project exists. By being distributed in multiple ecosystems, it supports wider software reuse, reaching a larger developer base and reflects the increasing interconnecting software development.

In modern software development there are multiple reasons for cross-ecosystem packages to exist, but it is what they actually offer that makes them a highly significant element in this field. The importance of cross-ecosystem packages lies in their ability to connect multiple distinct ecosystems while increasing software reusability and shared functionality. They spread tools, libraries and functions across ecosystems thus creating connections that extend beyond a single ecosystem.

2.4 Package Dependencies

Interconnections are a material aspect of software development because they reveal how software components relate and rely on one another. Since packages provide reusable functionality and components depend on them, thus forming dependency relationships within an ecosystem. Jaisri et al [14] described a package dependency as the reliance of a software library on an external module to function correctly. Generally, package dependency can be defined as the reliance of one package on another, for it to operate as expected, thereby creating a network of dependency relationships within an ecosystem.

A useful distinction in package dependencies is between direct and transitive dependencies. A direct dependency is a package explicitly declared and required by another package, In contrast, a transitive dependency is a package indirectly required through one or more direct dependencies. This distinction is significant because a package may rely not only on direct dependencies, but also on additional packages introduced through those dependencies.

Dependency relationships contributed to the complexity of software ecosystems. Packages may rely on one another directly or indirectly, therefore forming dependency networks within the ecosystem in which they are distributed. These networks can expand over time as packages are added or removed. Their complexity can be shaped by the number of direct and transitive dependencies, since these relationships influence how packages are connected.[19] These dependency networks play a significant role in software development because they help developers understand which packages rely on the package they maintain and manage. This knowledge is crucial when changes are introduced to a package, since they may affect dependent packages and potentially cause them to no longer function as intended. As a result,

understanding dependency networks is essential for improving maintenance, supporting compatibility and reducing the risk of failures within an ecosystem, especially in the case of cross-ecosystem packages, that may affect multiple ecosystems at the same time.

In conclusion, software components and packages that rely on other packages to operate correctly create dependency relationships within or beyond an ecosystem forming dependency networks. These networks allow developers before introducing any changes, to have the knowledge which packages will be affected and reducing the risk of failure across ecosystems. However, the existence of a dependency relationship does not necessarily reveal how a package is being used in practice, therefore is important to examine the method usage within the dependent package source code.

2.5 Method usage and co-usage pairs

For the purposes of this thesis, method usage refers to the invocation of methods provided by a package within their dependent source code, allowing the analysis of how package functions are used in practice. In this context, method usage focuses on the actual API calls made to the methods exposed by the package rather than the declaration of the package as a dependency. Such usage may occur in different code elements, including function bodies, declared assignments, object declarations, and other programming constructs. Therefore, studying method usage enables a deeper understanding of how dependent packages, especially cross-ecosystem ones, make practical use of their external package functionalities.

A more specific aspect of method usage is the concept of co-usage pairs, which refers to methods that are invoked together within the same block of code. Examining co-usage pairs helps reveal how different methods of a package are combined in practice by dependent packages. This results in providing a more detailed understanding of how package functionality is used in real software development scenarios.

Section 3
Related work

3.1 Research on Software Ecosystems

Previous studies have investigated software ecosystems with the purpose of explaining their complex environment. These studies targeted different aspects of SECOs such as their features, benefits and challenges, technical and social infrastructures either on an individual ecosystem or in general. These studies generated a clearer understanding of how software ecosystems evolve through time and their significance in software development.

Jansen et al [7] examined the business nature of software ecosystems, concluding on how to develop software units with strategic choices so they can be adaptable in new markets and business models. On the other hand, the study of Mens et al [15] focused on understanding the evolution of software ecosystems. Giving insight on how large developer communities maintain or evolve massive collections or large software systems over long periods of time and suggest better tools and techniques for more efficient maintenance.

Lungu et al [9] followed the process of reverse engineering on software ecosystems exploring concepts inside and outside of a system. From their examination they emphasized on the relevance of dependencies among modules inside a system, the collaboration between developers within a community and the evolution of components inside and outside a system. In addition, Software Ecosystem: Features, Benefits and Challenges by Joshua et al [13] had the purpose of detecting the main features of software ecosystems and introducing their benefits and challenges. Investigating ecosystems that follow the open-source model like, Google Android and Eclipse ecosystems, they formed a clearer view on ecosystems that enables research communities and developers to follow more explicit route towards new research.

Manikas et al [8] conducted a systematic literature review on software ecosystems from 90 studies with the intention of presenting common patterns between multiple definitions and providing their own. Moreover, they attempted to reveal software ecosystems aspects that were not explored, research flaws from previous studies, and describe the software ecosystem framework. Overall, they highlighted software ecosystems as a new area of high prospective and value for the industry and as a research field.

While researching the field of SECOs for almost twenty years, researchers offered a deeper insight into these environments that continuously evolve. As a result, it enables researchers to dig deeper within the ecosystems to further analyze their interconnected relationships.

3.2 Research on cross-ecosystem packages

Prior research also examined the characteristics and evolution of cross-ecosystem paxkages. Constantinou et al [1] investigated cross-ecosystem packages across 12 different package managers. Their goal was to uncover characteristics of cross-ecosystem packages to enhance even further the knowledge on these packages. From their work, they observed that in contrast to the considerable number of packages, a limited amount is distributed across multiple SECOs. In either case, they have a significant effect on packages within the ecosystem they are distributed in and especially to packages that depend on them.

Kannee et al [15] conducted an empirical study with the goal of examining how much cross-ecosystem libraries are shared between ecosystems. The study included approximately 4200 GitHub projects expanding across five ecosystems like PyPI, CRAN, Maven, Rubygems and NPM. They investigated the contributions of these software libraries and concluded that a notable number of contributions are received from one main ecosystem but also receive contributions from ecosystems that the library is not distributed. Furthermore, they discovered that these libraries are multilingual regarding programming language usage. This results in third-party libraries being replaced as developers unavoidably may need to switch programming language or maintainers to publish different versions for different ecosystems.

A thesis by Orphanos [2] examined cross-ecosystem packages that expand across five main ecosystems like NPM, PyPI, CRAN, RubyGems and Maven. The purpose of this thesis was

to analysis programming language usage patterns with the aim of identifying patterns of usage in the evolution of these packages. From the results he discovered seven usage patterns enriching even more the knowledge of dynamically language usage within the cross-ecosystem packages. On the other hand, Chatzimylou [3] thesis followed the same dataset as Orphanos [2] but focused on the behavior and collaborations among developers across multiple ecosystems and programming languages. Analyzing repositories that extend across the five major ecosystems, she detected eight unique patterns of social activity. The results display developers with higher percentages of contributions affect more the evolution of these packages and highlight the associations between social and technical patterns from the two theses.

3.3 Research on Package Dependency

Previous studies have also investigated package dependencies to understand how they affect software ecosystems. Decan et al. [19] examined seven packaging ecosystems with the goal of recording the dependency networks growth, changeability, reusability, and fragility. From their investigation they observed that these networks tend to grow overtime influenced by the number of packages and their updates. In addition, they highlighted that a small number of packages had the most reversed dependencies, while the majority were dependent.

Moreover, Decan et al [20] examined how dependencies evolve overtime in their large ecosystems and how do they affect resilience of these ecosystems. From their analysis, they concluded that dependency constraints can lead to issues and reduce beneficial dependency updates. Additionally, a minority of dependencies with strict constraints may not get adjusted for a large period when a dependent package releases a new version.

Section 4

Methodology

This thesis is an empirical study of cross-ecosystem packages and their dependents in the npm and RubyGems ecosystems. The research investigates the dependents' source code to identify the functionalities provided by the cross-ecosystem packages, thereby creating a clearer path to understanding how these packages are utilized within the ecosystems in which they are distributed. First, an existing dataset was used to uncover the relevant packages present on these platforms. Following this, package metadata and dependent packages were collected from the official registries of both ecosystems. The information and dependents were extracted using API requests, web navigation, and web scraping. Once the collection of the dependents was completed, the process of their source code extraction was initiated to perform static code analysis. The analysis was conducted by converting the source code into a syntax tree, supplying a path to discover the cross-ecosystem packages usage within their dependents.

4.1 Initial Scope and Data Selection

During the preliminary stage of the research, the initial scope of this study included five major software package ecosystems such as npm, PyPI, RubyGems, Maven and CRAN. To

identify cross-ecosystem packages, a dataset containing approximately 1.1 million packages across the five platforms was obtained from the study of Kannee et al. [16,17] This dataset was selected because it provided a large collection of package information within platforms intended for this study, making it suitable to determine packages that are distributed in multiple ecosystems. During dataset validation , an inconsistency was found related to Maven entries, as the package names were incomplete, which led to the replacement of the dataset. The final dataset was retrieved from an archived Zenodo dataset constructed by Jeremy Katz [5], containing information from 34 package managers supported by Libraries.io.

In addition, prior to the main analysis, the initial software ecosystems to be considered in the research were the five ecosystems mentioned above While examining the availability of dependent packages, PyPI did not display any information about the dependent packages in its official registry, hence it was removed from the final analysis. In contrast, Maven had available dependent package data; however, no permitted programming method was identified for collecting data, so it was also not considered further. Following the exclusion of these ecosystems, the number of relevant cross-ecosystem packages in CRAN was limited; consequently, it was removed from the scope. Therefore, the final analysis was limited to npm and RubyGems.

As a result, the final design of this thesis was shaped by the dataset of Jeremy Katz [5]. Although the original dataset contained packages from multiple ecosystems, this study focused only on npm and RubyGems platforms.

4.2 Data Collection.

Following the selection of the final dataset and ecosystem scope, the data collection was divided into three phases. Initially, the dataset was processed to identify packages distributed in both npm and RubyGems platforms. These packages were classified as cross-ecosystem packages, which served as the starting point of the methodology. The next phase involved collecting dependent package data from the corresponding registry source, to determine declared dependencies on the cross-ecosystem packages. Finally, the source code of the selected dependents was retrieved and stored to support later analysis. These phases provided all the required data for the remaining stages of methodology, allowing the analysis to delve deeper into the matter of understanding better how the cross-ecosystem packages are used.

Before starting the data collection process, the dataset created by Jeremy Katz [5] was investigated and only relevant information for this study was kept. The dataset provided multiple CSV files containing data from the open-source repository Libraries.io. This dataset included files for projects name and platform, tags, versions, dependencies, repositories, repository dependencies and projects with related repository fields. Only the file containing information about the projects name and platform was retained to identify the cross-ecosystem packages. The dependencies CSV file was excluded because the dataset was created in 2020, meaning that the dependency information could therefore be outdated for the purposes of this research

The selected csv file provided information for 34 package managers and approximately four million packages. The format of the CSV was structured with the following columns; ID, Platform, Name, Created Timestamp, Update Timestamp, Description, Keywords, Homepage URL, Repository URL, Version Count, Source Rank, Latest Release Publish Timestamp, Latest Release Number, Package Manager ID, Dependent Project Count, Language, Status, Last synced Timestamp, Dependent Repositories Count and Repository ID. As a result, this csv file supplied all the required information for the detection of packages distributed in both ecosystems.

4.2.1 Cross-Ecosystem Packages.

After selecting the appropriate CSV file from the dataset, the data collection process focused on preparing the package records required for the analysis. Although, the file contained package information for multiple ecosystems, only entries from npm and RubyGems were retained, as these ecosystems formed the final scope of the study. In addition, packages whose repository URLs did not correspond to a GitHub repository were excluded. This workflow produced approximately one million package records from the selected platforms that were associated with a GitHub repository.

Following the selection of the records, duplicates were filtered out and the remaining records were grouped by their repository URL. This approach provided the insight to identify packages that were distributed across more than one platform. Packages linked to an invalid repository such as “http://github.com”, “https://github.com”, “http://github.com/” and “https://www.github.com” were removed, as they do not point to a specific repository. To

collect the cross-ecosystem packages, repositories that were linked only to one platform were dropped, therefore keeping only the ones linked to the two platforms. As a result, a collection of 676 packages, corresponding to 338 cross-ecosystem packages between npm and RubyGems was produced. The identified packages were extracted into a separate CSV file while retaining the data from the dataset they originated from. This ensured that potentially necessary information, required in subsequent parts of the methodology, was preserved.

4.2.2 Dependent Packages

Once the cross-ecosystem package dataset was established, the next phase of the data collection process focused on retrieving dependent packages. This step was necessary because identifying the selected packages alone does not indicate the extent to which they are used by other packages. Therefore, dependent packages were collected to provide essential data needed to support the examination of used methods from the cross-ecosystem packages.

Dependent package collection was one of the most challenging phases of the research, as the required information was not available in a consistent way across the different methods approached. Although Libraries.io was initially examined as it presented an API call offering a package's dependents and metadata in a JSON format, the API was unavailable as it returned the message "Disabled for performance reasons". For this reason, the collection was carried out using the available official registry data from npm and RubyGems. Based on the mechanism the registries provided their dependents, multiple approaches were applied with the aim of uncovering and collecting the required data.

The npm website allows crawler access to package metadata under controlled conditions, such as limiting the request rate to one request per second or less. Therefore, the scraping process was designed with delays in order to meet the requirements of the npm registry. In contrast, RubyGems registry provided the required data in JSON format through API calls. In RubyGems, this information is presented as reverse dependencies, which refers to packages that rely on the selected package therefore, they were treated as dependent packages. From each selected cross-ecosystem package, necessary metadata was collected including the package's latest version and their dependents list. Moreover, for each dependent package information regarding their version, dependency type on the cross-ecosystem package and

when they were last modified was collected. In addition, when available, the required version of the dependent package was collected as it may have been useful in later stages.

From this workflow, 12,452 dependents and their associated metadata were collected. Cross-ecosystem packages for which no declared dependents were discovered, were not represented in the dependent package records, since they could not contribute to method-usage analysis. The resulting records were stored in a CSV file that combined information about the selected cross-ecosystem packages and their associated dependents. The CSV file includes columns such as Package Platform, Package Name, Package Version, Dependent Name, Dependent Version, Dependent Type, Dependent Last Modified, and Required Version.

To improve further the reliability of the dependent package dataset, a typosquatting-based filter was applied. Typosquatting is a malicious attack where packages are named similarly to existing ones, thereby misleading developers into installing malicious software. However, similar package names are not always malicious; they may also result from accidental naming similarities or legitimate package variations. The purpose of this filter was to flag possible typosquatted packages in the collected dataset to avoid analyzing method usage of suspicious cross-ecosystem packages. This filter procedure employed the edit distance to compare cross-ecosystem and dependent package names against package names from the dataset by Jeremy Katz [5] within the same platform. Package pairs with an edit distance of one were classified as high-risk typosquatting candidates, while pairs with an edit distance of two were classified as medium risk.

To reduce false positives, name comparisons were limited to names of the same length to ensure structural similarities and meaningful results. Pairs linked with the same repository were excluded from the filtering, since they are linked to the same software project. For npm-scoped packages, pairs that are associated with the same scope were also excluded, as they belong to the same organization. The output of the typosquatting filter was stored in a separate CSV file containing the package platform, checked package name, checked package type, matched package name, edit distance, and assigned typosquatting risk. Since the filter was based on name similarity alone, 133 cross-ecosystem packages were manually reviewed as they were high risk candidates for typosquatting. The manual review was based on the overall number of downloads reported by the corresponding platform. None of these packages were removed during the review, as the examined packages had a substantial

download count and did not appear suspicious. Packages with limited downloads had no dependent packages and therefore were already absent from the dependent package dataset.

Additionally, dependent packages whose latest published version was marked as pre-release were removed from the dataset. Versions containing identifiers such as alpha, beta, rc, pre or preview were treated as pre-release versions, 320 out of 12,452 dependent packages were removed, leaving 12,132 dependents. This filter was applied to ensure the analysis focused on stable package releases and limit noise introduced by experimental, unstable and incomplete packages. In contrast, packages with their latest published version being their initial version, for example, "0.1.0", remained in the dataset. The purpose of not removing these packages was to avoid eliminating packages that do not follow the semantic versioning strictly.

4.2.3 Dependent Packages Source Code

Upon filtering the dependent packages' data, a more trustworthy dataset was formed enabling the extraction of reliable dependent source code. The source code files were retrieved through API requests corresponding to the official registry package archives. For npm packages, the source code was retrieved using the URL 'https://registry.npmjs.org/{package_name}/-/{package_name}-{version}.tgz' with the relevant package name and version inserted to corresponding field. On the other hand, Rubygems packages were downloaded using "https://rubygems.org/downloads/{gem_name}-{version}.gem" where gem name and version identified the selected package archive. This approach obtained compressed files containing the source code required for the analysis. The extraction process was based on the compression type, and the extracted files were stored locally.

Through manual investigation of a limited amount of dependent package source code, the source code of the RubyGems cross-ecosystem packages was crucial in order to identify method usages from their dependents. Therefore, Rubygems cross-ecosystem packages were extracted using the same approach. This procedure supported the identification of modules and class names of cross-ecosystem packages used by their dependents.

As a result of this workflow, the downloaded packages were organized by their type (cross-ecosystem or dependent) and platform. Additionally, scoped packages create errors due to the

usage of special characters; their names were encoded and added to the dataset, so their file names were documented.

Packages that could not be extracted successfully were marked, together with their linked error details. This contributed to the detection and exclusion of packages affected by technical extraction problems and reducing the risk of analyzing incomplete or incorrectly extracted source code files. During the extraction process, 67 dependent packages could not be extracted, and 1,696 packages did not contain any source code files. In total, 10,369 packages were successfully extracted and kept for the analysis.

Consequently, a modified CSV file was formed, preserving previously collected dependent package data while adding the encoded dependent package file name where encoding was required. The file included fields marking packages with extraction failure and their associated error details.

4.3 Static Analysis of source code

A static source code analysis was conducted on the 10,369 downloaded dependent packages to examine whether the declared cross-ecosystem dependencies were actually used in the source code. The metadata only indicates the existence of dependency relationships, but it does not show how the cross-ecosystem package is referenced within the dependent package. Therefore, the analysis focused on identifying import and require statements and method usage calls related to cross-ecosystem packages. This made it possible to distinguish packages with only a declared dependency and packages that actually reference cross-ecosystem packages in their implementation.

Instead of relying on regular expressions that detect only simple textual patterns and may possibly miss or incorrectly match source code constructs, this study followed a more complex approach. The method employed was the syntax-tree-based analysis supporting the navigation of the source code structure and enhancing the ability to successfully identify imports, require statements, and invoked methods.

4.3.1 Dependency Reference Detection

The aim of this analysis was to detect cross-ecosystem package references in the source code of the dependents and capture assigned aliases. This approach followed the same general structure but was adapted according to the programming language being analyzed, such as Ruby, JavaScript, and TypeScript. JavaScript and TypeScript share similar import patterns, though TypeScript offers additional forms. Ruby, on the other hand, uses different mechanisms for referencing external packages. As a result, the import analysis was adjusted according to each programming language keywords for loading packages without altering the final output.

In RubyGems packages, two types of dependency-related constructs were considered. The first type consisted of dependency declarations in the .gemspec file, such as “add_dependency”, “add_runtime_dependency” and “add_development_dependency”. These declarations indicate that package requires another gem, allowing the dependent package to access modules and methods from that gem. However, this does not indicate that external packages are utilized. The second form refers to packages on file using source code loading statements such as “require”, “include”, “extend”, ”prepend”, and “gem”. The second type of referencing was also examined for variable assignments. When a loaded package was assigned to a local variable, the assigned variable name was recorded for later analysis. Both types were captured as they provide evidence of how Rubygems packages employ their declared dependencies.

In contrast, npm packages followed a more direct approach because JavaScript and TypeScript source files commonly contain explicit dependency references. These references were identified through multiple import forms including “require”, “import”, “named imports”, “namespace imports”, wrappers, and “side-effects", while “import_alias” was the additional form for TypeScript. Named imports and namespace imports were highly significant because they indicated whether a dependency was imported under a local variable. This allowed the detection of direct references of cross-ecosystem packages on their dependent source files while also capturing assigned aliases and identifiers to the references.

On both platforms, multiple reference forms were examined during static analysis. Since Ruby, JavaScript, and Typescript use different mechanisms for referencing dependencies,

Table 1 provides a brief description of each npm reference form considered in the analysis, while Table 2 provides the corresponding description for RubyGems reference forms.

Import/Refence Form	Description
require	Reference through require statements
import	ES module import statement
Name import	Specific functions imported directly form the package
Namespace import	Package imported under an identifier
Side-effects	Imported without assigning it to a variable
wrapper	Reference inside a wrapper or transformed import structure
Import alias	A package function import using an alias

Table 1: Import/Reference Forms Description for JavaScript and Typescript

Import/Refence Form	Description
add_dependency	Dependency declaration in .gemspec file
add_runtime_dependency	Runtime Dependency declaration in .gemspec file
add_development_dependency	Development Dependency declaration in .gemspec file

require	Reference through require statements
include/extend/prepend	Extend module functionality
gem	Declares RubyGems package dependency

Table 2: Import/Reference Forms Description for Ruby

Source files without references were handled differently depending on the platform. In RubyGems packages, dependencies may be made available through autoloading mechanisms without a direct reference. To reduce the risk of eliminating source files with useful method usages for this research, files without any explicit references were not excluded. In contrast, npm source files without any detected dependency references were excluded from the method usage detection, since JavaScript and TypeScript usages were traced through direct references.

A crucial aspect of the reference detection was the identification and collection of identifiers and aliases assigned to imported packages for each source file. Since dependents may use a dependency through a local alias rather than its original name, recording these assignments enabled a more accurate analysis. This allowed later method calls to be associated with the correct cross-ecosystem package even if it was linked to a different name. Additionally, the detected reference forms were collected with their linked aliases, creating import-form and alias pairs. For Ruby packages without any found references, the reference form and alias were marked as “None”, whereas direct references without any alias were marked as “unassigned”. These pairs provided valuable information for understanding how a dependency relationship is declared in the implementation of the dependent package

4.3.2 Method Usage Detection

With the dependency references aliases and identifiers recorded for each dependent source file, the static analysis centered on discovering method usages. This part of the analysis

examined how dependent packages employed the referenced cross-ecosystem packages in their source code files. Method calls associated with the detected imports, required statements, declarations, and aliases were collected in order to provide insight into how the cross-ecosystem packages are used by their dependents.

Before method usage detection could be performed for RubyGems dependents, the module names associated with the referenced cross-ecosystem packages were extracted. The module names were collected from the cross-ecosystem package source code following the syntax-based tree approach. This was a necessary step because Ruby code often references functionality through these modules, meaning the method usages may appear through these module names even when no direct loading statement is present in the source file.

Based on the collected module names and the local aliases, the source files of RubyGems dependents were examined to detect method calls. Method calls were recorded when the caller matched an identified module name or an associated alias. For each detected usage, both the caller and the invoked method were recorded. This workflow provided the fundamental information required for understanding how dependents actually implement their dependency relationship with cross-ecosystem packages inthe RubyGems ecosystem.

For npm dependent packages, captured variables and aliases from each source file were used as anchors for detecting method usage from each source file. Method calls associated with these names, along with their invoked methods, were collected in order to provide a traceable link to the referenced cross-ecosystem packages and their used methods.

Even though method calls were investigated in different ways, additional information regarding the method usage was extracted using the same approach. The method usages were distinguished based on the block of source code they were identified in. These blocks refer to programming construct types, including functions, methods, modules, and classes. Method usages not located inside any constructs were marked as “Global”, indicating their appearance at the top level of the source file. In addition, named blocks were recorded using their assigned name, whereas unnamed blocks were recorded as “anonymous”. For each block, the starting and ending line numbers were recorded, as well as the line number of the method usage within the block. Unique identification tokens were generated for each block by combining the block name with start and end line numbers. These tokens provided unique

distinctions between method usages, enabling a more detailed in-depth analysis of the employed methods.

4.3.3 Static Analysis Output

The static analysis produced 457,256 method usage entries from 4,202 dependent packages and stored them in a structured CSV file. Each row represented a detected method usage in the related cross-ecosystem package utilized inside the dependent package. The columns were selected to contain information regarding the analyzed package, the referenced package, the source file, the detected reference form, the invoked method, and code block data. This data allowed each identified method to be traced back to the dependent package, source file, and exact code block in which it appeared. The structure of this output dataset is summarized in Table 3.

Column	Description
Package Platform	Ecosystem of the analyzed dependent
Scanned Package	Dependent package analyzed
Imported Package	Reference cross-ecosystem package
File Name	Source file the usage was detected
Language	Programming language of the source file
Import Form	Type of reference detected
Import As	Alias/Local name assigned to the reference
Caller	Object/Module/Class/Alias making the call
Method	Invoked Method
Block ID	Identification token created

Block Name	Block name identified
Block Type	Programming constructed type
Usage Line	Line's number the method usage was detected
Start—end Block Line	Starting and Ending line number of the detected block

Table 3: Method usage CSV file column description

4.4 Method Usage and Co-Usage Analysis

Once the method usage dataset was established, the records were analyzed to identify how cross-ecosystem packages were used by their dependent packages. The examination divided the data into several dimensions of method usage, including detected reference forms, individual method usage, and co-usage pairs within the same block. By exploring these the analysis provided a more detailed understanding of method usage and contributed to addressing the main research objective.

The dimension regarding individual usage aimed to uncover multiple aspects of how referenced cross-ecosystem packages were used, particularly their usage frequency. This analysis examined which cross-ecosystem package appeared most frequently in method usage records. This examination was completed by grouping the extracted data using the package platform and the imported cross-ecosystem package. This analysis helped identify cross-ecosystem packages that are actively used by their dependents rather than just being referenced as dependencies.

Beyond total usage frequency, the analysis also examined the number of distinct methods used from each cross-ecosystem package. The purpose of this process was to investigate the breadth of the method usage pool associated with each cross-ecosystem package within its dependents. For this step, the data set was grouped by package platform and imported cross-ecosystem package, and the number of unique methods was calculated for each group. This exploration enabled the study to identify which cross-ecosystem packages used a wider variety of functionality and which were mainly accessed through a limited set of functions.

With the individual examination completed, the analysis shifted its focus to method usages that appear within the same code blocks. By identifying these pairs, the research moved beyond isolated method calls and investigated how dependent packages combined functionality from their referenced dependencies.

For the achievement of this goal, the dataset was grouped by the package platform, the scanned package, imported package, file name, and block id, hence the groups created included all the functions that appear in each block. From the grouping results, all method combinations were generated for each block. These combinations formed co-usage pairs, and their occurrences were counted across all blocks. This helped identify the most frequent co-usage relationships across the dependent packages. Furthermore, distinct co-usage pairs were extracted from each package with purpose of inspecting the variety of methods used within the same construct.

Moreover, the analysis also examined the import or reference forms used to access each cross-ecosystem package. The aim of this exploration was to gain a clearer understanding of how dependent packages reference cross-ecosystem packages before using their methods. For the completion of this task, the method usages were grouped by their import form. Therefore, examining these forms helped identify the numerous ways a dependency is declared and grants access to its methods.

Overall, this section described the various perspectives applied for investigating individual usage, co-usage and import forms from the extracted method usage dataset. The individual usage examination focused on methods that were most frequently used and how widely the package's functionality was accessed. The same measures influenced co-usage examination; however, it extended the view by identifying methods appearing together within the same block of code. Imports form analysis targeted the ways a cross-ecosystem package was introduced into their associated dependent source files. Therefore, these provided fundamental answers to the research question.

4.5 Development Tools and Libraries

This section describes the tools and libraries used to support data collection and the analysis process. The methodology was implemented in the Python programming language, while Jupyter Notebook was used as the development environment for implementing, testing, and executing data collection and analysis scripts. Additional libraries were used for data processing, web scraping, file management and static source code analysis to support the execution of the research methodology.

Pandas was one of the main libraries utilized throughout the study, as most of the data collected was stored and processed in CSV files. This library supported the processing stage, enabling datasets to be loaded, filtered, cleaned, and stored in a structured format. Additionally, the os module was employed to construct file paths, access directories, and check whether files or folders existed and organize the collected data in a consistent folder structure. These libraries had a crucial role in data manipulation, file navigation and storage, hence enhancing the overall performance of this research.

Moreover, the Selenium library provided all the functionalities needed for web navigation and web scraping. It facilitated automated navigation of registry web pages and the extraction of relevant dependent information from HTML content. The Requests library enabled interactions with official registry APIs, allowing package metadata, dependent package information, related information and source code to be retrieved automatically. These tools contributed to the gathering of various types of information and data needed, therefore served as critical elements of this study.

For the static source code analysis stage, the tree_sitter_language_pack module was selected. Unlike the other libraries used for data collection, this module was used to parse downloaded source code files and represent them as structured syntax trees. In addition, it offered methods for navigating these trees and made it possible to identify imports and method usages from the source files. Tree_sitter_language_pack enabled the extraction of method usage and import statements, supporting the analysis of dependent packages' source code.

Section 5

Results

This section presents the results of the source code analysis of dependent packages across npm and RubyGems, the selected ecosystems for this research. Based on the generated method usage dataset, several perspectives were examined to understand how dependent packages interact with cross-ecosystem package functionalities.

5.1 Overview of Method Usage Analysis

This chapter provides an overview of the final dataset in this research. This dataset was produced through static code analysis of the dependent packages on the two platforms. Table 4 summarizes the crucial findings, including the numbers of packages processed, packages with valid source code, packages with detected references, reference cross-ecosystem packages, and the final recorded method usages extracted. This overview clarifies the scope of the extracted dataset and the produced results in the following sections.

	NPM	RubyGems	Total
Dependent packages processed	6,277	4,092	10,369
Dependent packages with detected references	2,043	2,159	4,202
Referenced cross-ecosystem packages	45	62	107
Method Usage Recorder	405,969	51,287	457,256

Table 4 Method Usage Dataset Summary

As shown in Table 4, a total of 10,369 dependent packages were processed across npm and RubyGems. From these packages, detectable references to cross-ecosystem packages were found in 4,202 dependent packages. These references were associated with 45 packages from npm and 62 from RubyGems; in total 107 cross-ecosystem package references. The static analysis produced 457,256 documented method usages from these two ecosystems. The majority were accounted from npm with an amount of 405,969 recorded method usages, whereas RubyGems accounted for 51,287 method usages. A difference that can be spotted from the two ecosystems is that Rubygems contained slightly more referenced packages, while npm produced a substantially larger number of method usage records.

5.2 Individual Method Usage Results

Following the overview of the final dataset, this section presents the results of the individual method usage analysis. The purpose of this was to examine how dependent packages use the selected cross-ecosystem packages at the source-code level. Each detected usage represents a method call associated with an imported, required or otherwise referenced cross-ecosystem package.

The results are divided into two parts. First, the method usage frequency is examined to identify which cross-ecosystem packages were used most often by their dependents. Second, the number of distinct methods is inspected to show the variety of the invoked methods employed for each package.

Method usage frequency and distinct method usage measures offer different perspectives regarding the accessed methods. The frequency counts the total number of detected method references, including repeated usages of the same method. In contrast, the distinct aspect counts the number of unique methods captured for each cross-ecosystem package.

5.2.1 Cross-Ecosystem Packages with the Most Method Usage

Before investigating method usage on the package level, the total number of detected method usages was compared between npm and RubyGems. This supplies an overall view of how method usage was distributed across the two ecosystems.

Figure 1 presents the percentage of the total detected method usages from npm and RubyGems. This figure shows that npm accounted for a larger percentage (88%) of the total detected method usages rather than RubyGems (11%). This indicates that, within the collected dataset, npm dependents contained more method calls to the cross-ecosystem packages. However, the result should be interpreted together with the package level analysis ,hence to provide further insight for a better understanding.

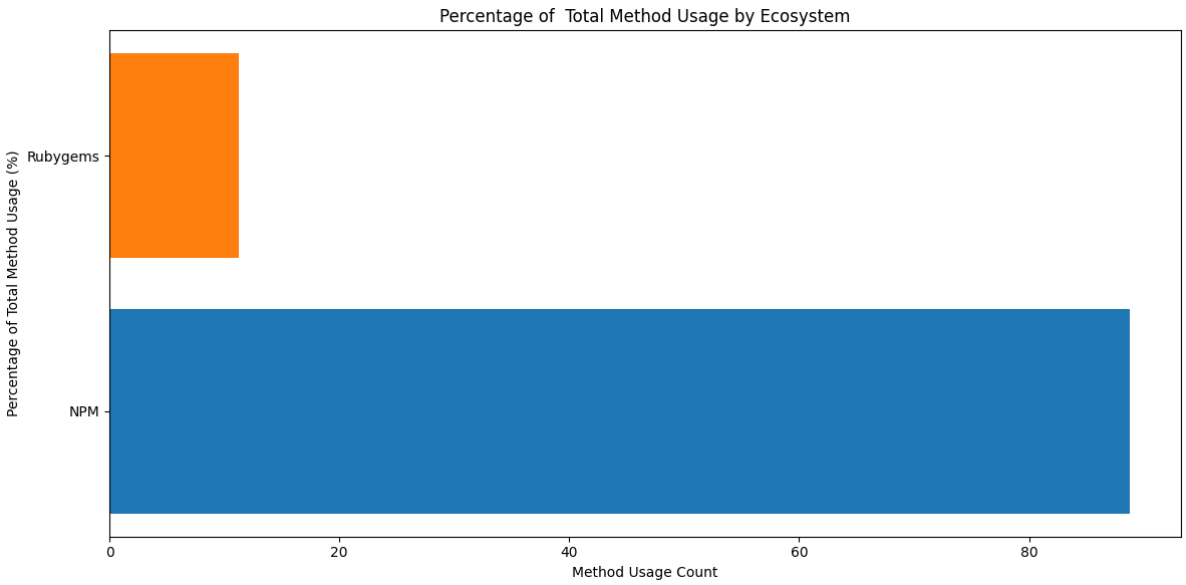

Figure 1: Percentage of detected method usages by ecosystem

The first individual usage result examined which cross-ecosystem packages appeared most frequently in the method usage entries. This was measured by counting the total number of detected method usages linked with each imported cross-ecosystem package. The purpose of this result was to identify which packages were mostly used at the source code level by their dependent packages.

Figure 2 presents the ten cross-ecosystem packages with the highest number of method usages using a logarithmic scale to highlight the large gap while also being visually presentable. This provides a general view of which packages had the overall highest usage frequency across the dependents.

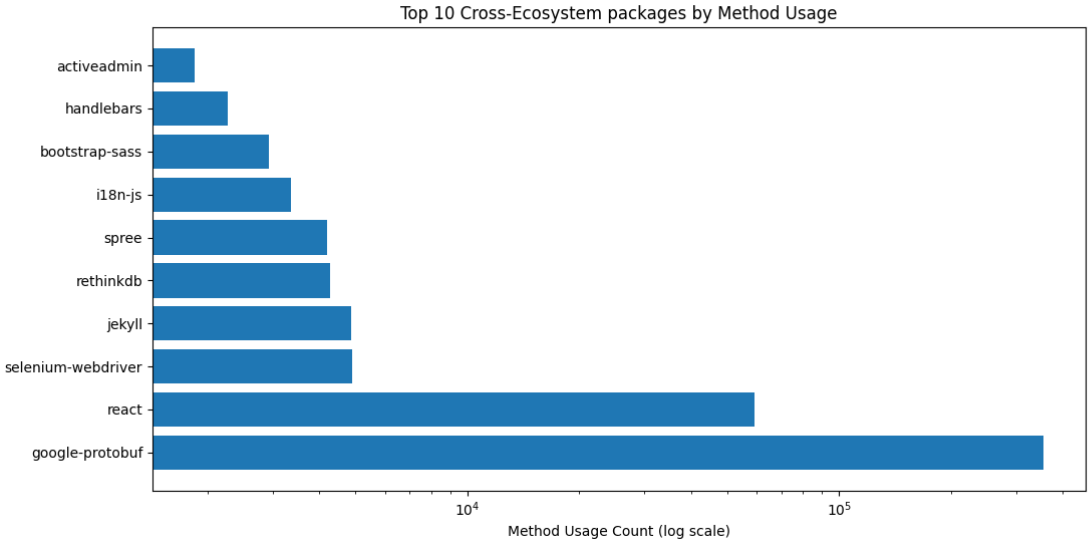

Figure 2: Top cross-ecosystem packages by overall recorded method usage frequency

As shown in Figure 2, 'google-protobuf' recorded the highest overall method usage count, clearly exceeding the remaining packages. This reveals that a substantial proportion of method usages were associated with this package. In addition, a significant amount of method usage was linked to 'react' while the other packages contributed less.

For a more detailed analysis, the packages were grouped by their platform to determine from which ecosystem the highest ranking package frequencies originated from. The aim of this separation was to distinguish whether cross-packages from npm, RubyGems, or both contributed to their overall frequency.

The following figure presents the highest ranking cross-ecosystem packages identified in Figure 2, with their method usage frequency broke-down by platform. This provides a clearer view of how recorded method usage was distributed between npm and RubyGems. For packages with the same name in both ecosystems, such as 'google-protobuf', the overall count in Figure 2 includes method usage from both platforms, whereas Figure 3 separates it. Packages that appear under different names across ecosystems such as 'react/react-source', the corresponding package from the other platform is also included for comparison. Both names are shown on the label, with the npm package name on the left, while RubyGems name is on the right.

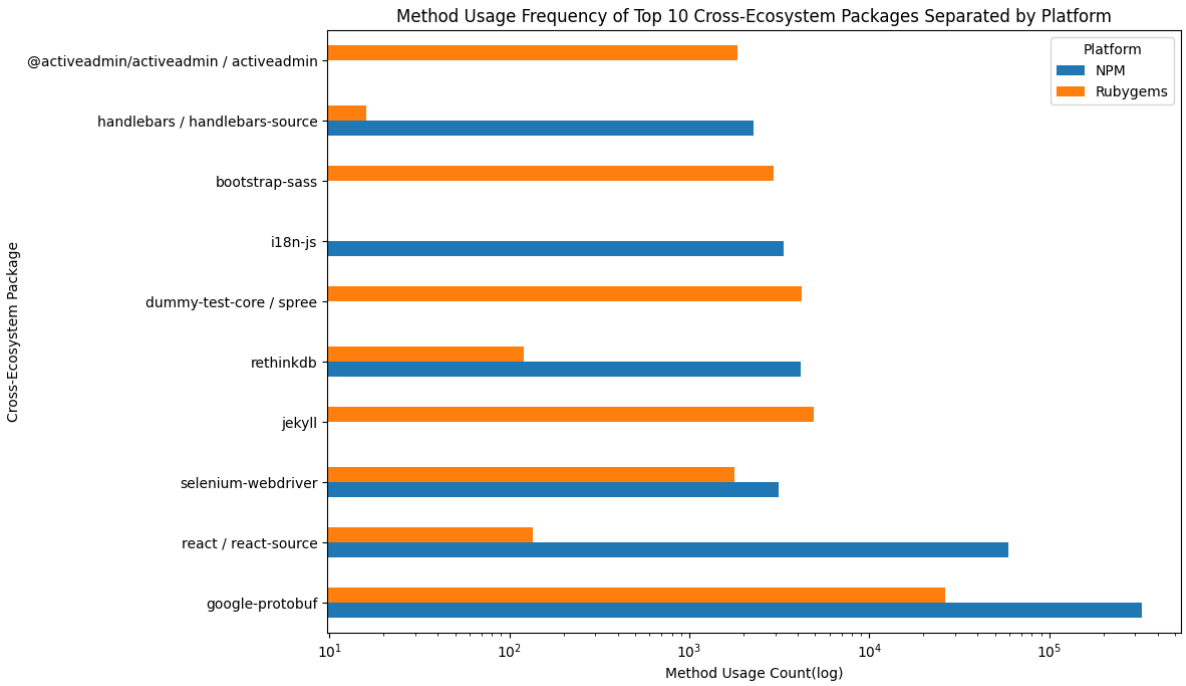

Figure 3: Top 10 cross-ecosystem packages method usage frequency by platform

Figure 3 shows that the frequency of method usage for these packages was not spread evenly between npm and RubyGems. The 'google protobuf' was the most dominant, with the largest number of recorded method invocations appearing in npm dependents. Although this package also recorded method usage in RubyGems, the difference between the two platforms was substantial, indicating that its overall dominance originated from the npm dependents source code.

Several other packages also appeared in both ecosystems, but with different usage patterns. For example, react/react-source' had a much higher frequency in npm than RubyGems, as well as 'handlebars/handlebars-source' and 'rethinkdb'. On the other hand, 'selenium-webdriver' displayed a more balanced distribution across the two ecosystems. For the remaining packages, method usage was concentrated within one ecosystem. Packages namely as 'activeadmin', 'bootstrap-sass', 'jekyll', and 'spree' were mainly associated with RubyGems, whereas 'i18n-js' was strongly associated with npm.

5.2.2 Distinct Methods Usage per Cross-Ecosystem Package

The second examination at the individual level focused on the number of distinct methods the dependent packages applied from cross-ecosystem packages. This perspective provided a wider view of package usage by showing whether the dependents relied on a small set of repeated methods or accessed a larger set of functionalities. Unlike total method usage frequency, which can be strongly affected by method reuse of the same methods, distinct method usage shows the variety of methods used from each package.

Figure 4 shows the packages with the most unique methods detected in the dataset. This allows the results to highlight the range of functionality accessed from each ecosystem package rather than total number of invoked methods

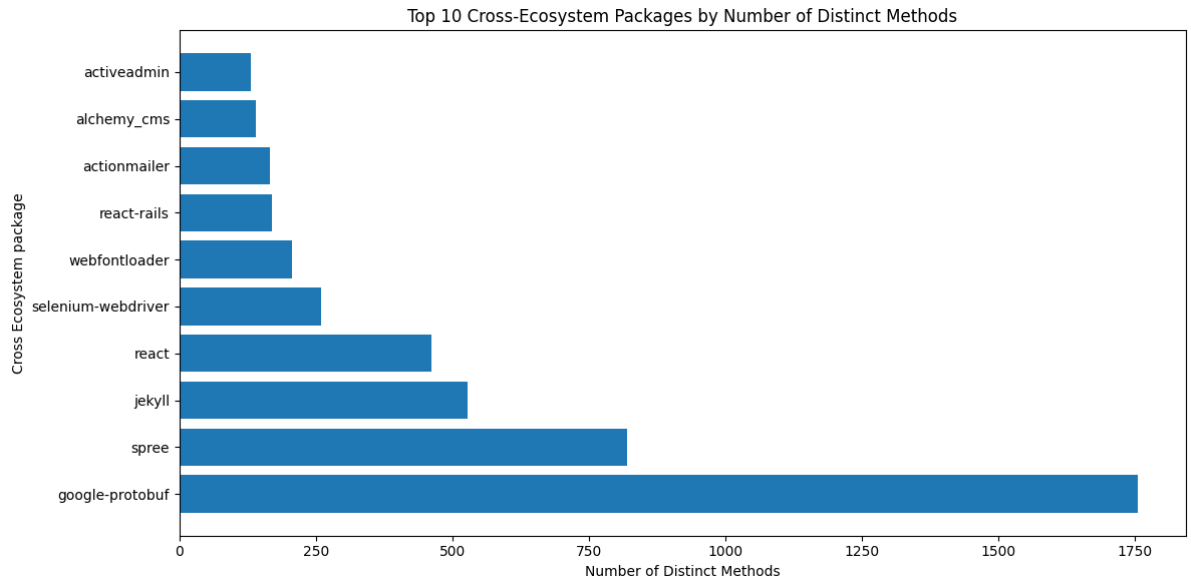

Figure 4: Top 10 package with the most unique method usage

The distinct method results reveal that ‘google-protobuf’ had the highest number of distinct methods among the analyzed cross-ecosystem packages. Similarly, packages that appeared with a high method usage frequency, such as ‘react’, ‘jekyll’, ‘spree’, selenium-webdriver', and ‘activeadmin’ also appear in this examination. In contrast, some packages that were not included in the frequency analysis, such as ‘webfontloader’, ‘actionmailer’ and

‘alchemy_cms’ were observed in this examination. As a result, distinct method usage provided a different perspective from the total frequency, since it emphasized diversity rather than only the number of method calls.

Following the findings presented in Figure 4, the examination then focused on how the distinct methods are distributed across both platforms. Figure 5 follows the same structure as Figure 3, separating results by platform and including the corresponding packages from the other platform for comparison. This allows the distinct method counts of the highest ranking packages from the previous figure to be compared between npm and RubyGems.

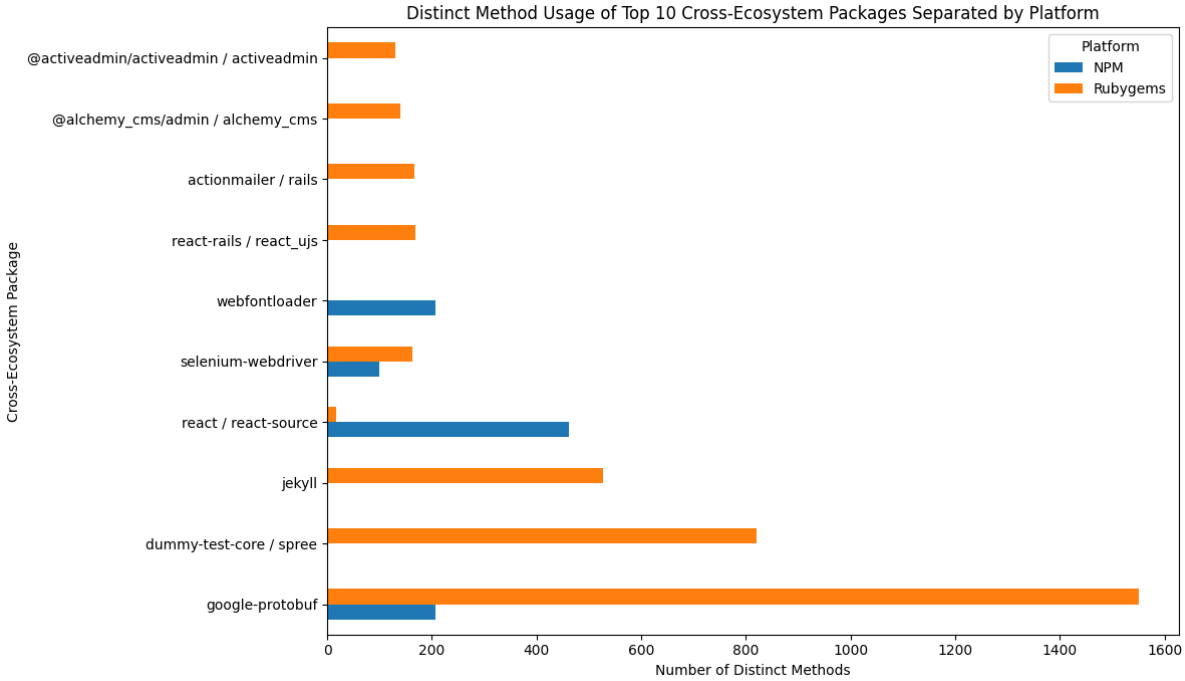

Figure 5: Top 10 cross-ecosystem packages distinct method usage by platform

Figure 5 shows that distinct method counts were not evenly distributed across ecosystems. Interestingly, ‘google-protobuf' had more distinct method access in RugyGems, even though it had significantly less usage frequency. In contrast, this package on npm had limited diversity in the method calls compared to the considerable proportion of total method usages.

Additionally, the ‘react’ package in npm had a higher number of distinct methods in npm than RubyGems. Moreover, the ‘selenium-webdriver’ showed a more balanced distribution across the platforms. In the method usage frequency results, npm displayed a slightly higher frequency for ‘selenium-webdriver’, in the distinct methods result, RubyGems showed a

slightly higher number of distinct methods. The remaining packages show that their distribution of the variety set strongly corresponds to one platform, while several of these packages were associated with RubyGems.

5.3 Method Co-Usage Results

After examining individual method usages, the results focused on method co-usage. This part of the results aimed to identify method usage pairs that appeared within the same code block and the cross-ecosystem packages they originated from. The package level co-usage results show which cross-ecosystem packages produced the highest number of method combinations. In this way, co-usage provides a more contextual view of dependency usage by showing which methods were used together during implementation.

5.3.1 Cross-Ecosystem Packages with the Most Co-Usage Frequency

The first co-usage result examined the cross-ecosystem packages with the highest total co-usage frequency. The frequency represents the sum of all detected co-usage pair occurrence within each package. This identifies the packages whose methods were most frequently used together inside the same code blocks.

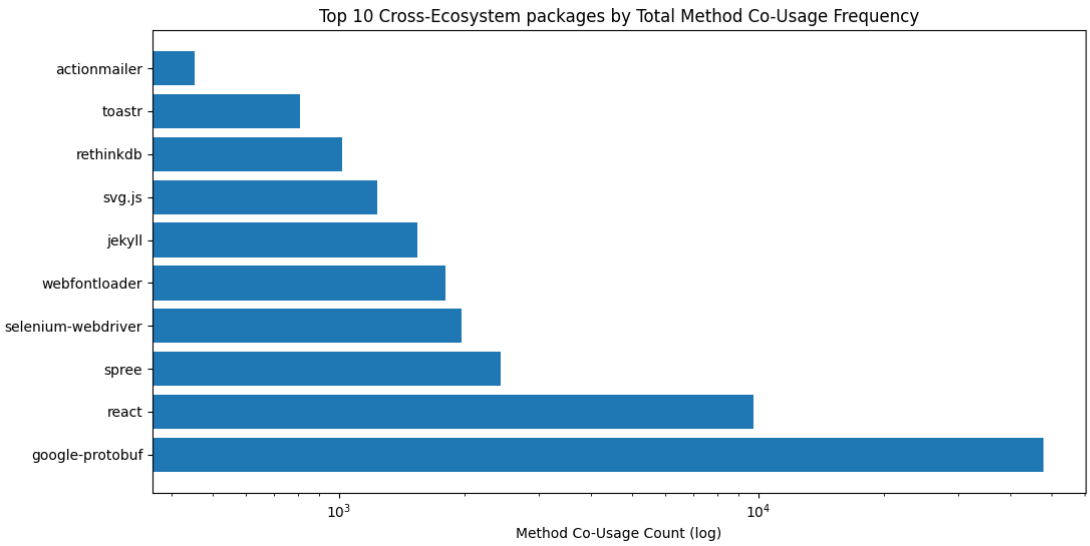

Figure 6: Top cross-ecosystem packages with most Co-usage Frequency

As shown in Figure 6, ‘google-protobuf’ had the highest co-usage frequency among the analyzed cross-ecosystem packages. Several packages that were present in the highest method frequency results also appeared in this analysis, including ‘react’, ‘spree’, ‘selenium-webdriver’, ‘jekyll’ and ‘rethinkdb’. Moreover, packages such ‘svg.js’ and ‘toastr’ which were not among the highest ranking packages in individual methods usage results, appeared in the co-usage analysis.

To explore this part of the results further, the following figure presents the separation of the co-usage frequency by platform. This figure follows the same structure used for comparison of packages between npm and RubyGems. The purpose of this examination is to provide more details about the co-usage frequency by the dependent packages.

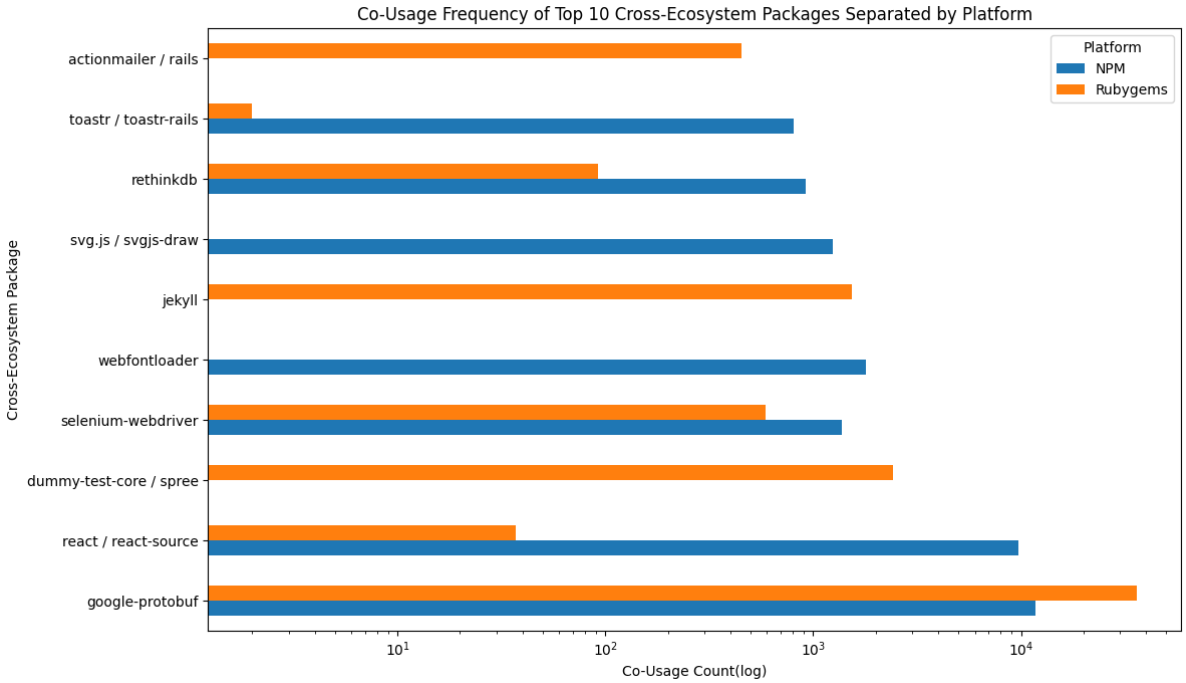

Figure 7: Top cross-ecosystem package with the highest method co-usage frequency

These results, present that although ‘google-protobuf’ in RubyGems had the highest method-co usage frequency, it did not have a dominant role. Additionally, the co-usage frequency between the corresponding platforms showed slight differences, in contrast with previous results, which exhibit major gaps.

Furthermore, the ‘react / react-source' exhibits the same pattern as previous results, showing a higher frequency than the package in RubyGems. The ‘toastr/toastr-rails’ and ‘rethinkdb’

show that the associated package in npm produced more method co-usages, but ‘toastr/toastr-rails’ had significant difference in the frequency. The ‘selenium-webdriver’ package showed a smaller difference between npm and Rubygems, with co-usage frequency appearing more balanced across the two platforms. The remaining packages did not present any co-usage methods on both platforms, therefore suggesting that their contribution was strongly in one ecosystem.

5.3.2 Cross-Ecosystem Package with Most Distinct Co-Usage Pairs

The next co-usage result targeted the various methods of co-usage pairs associated with the cross-ecosystem packages. While total co-usage frequency measured how often method pairs appeared, the distinct co-usage pair aimed to uncover how many different combinations were captured from each package. This difference helped identify packages whose functionality was combined in more diverse ways within dependent source code, rather than packages driven mainly by repeated occurrences of the same pairs.

The following figure presents the top ten cross-ecosystem packages with the highest number of distinct method co-usage pairs. Each value represents the number of unique pair combinations detected for a package, regardless of the co-usage frequency measured in the previous section.

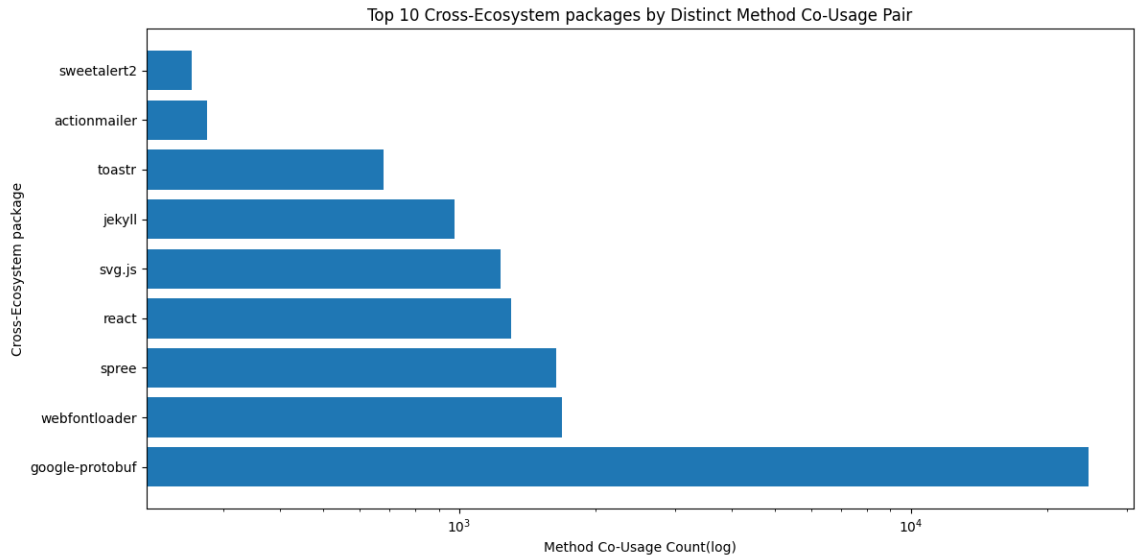

Figure 8: Top 10 packages with most diverse co-usage pairs

Figure 8 shows that ‘google-protobuf’ regains a dominant role, with a significant difference based on the distinct method co-usage results. Some packages such as ‘sweetalert2’ and ‘actionmailer’ had significantly fewer unique co-usage pairs compared with higher ranking packages. The remaining packages present smaller differences in their distinct pairs.

To provide more insight, the analysis examined these packages separately based on distinct co-usage pairs in each ecosystem. Figure 9 follows the same structure as Figure 3 and presents this examination to explore a different aspect of the previous result.

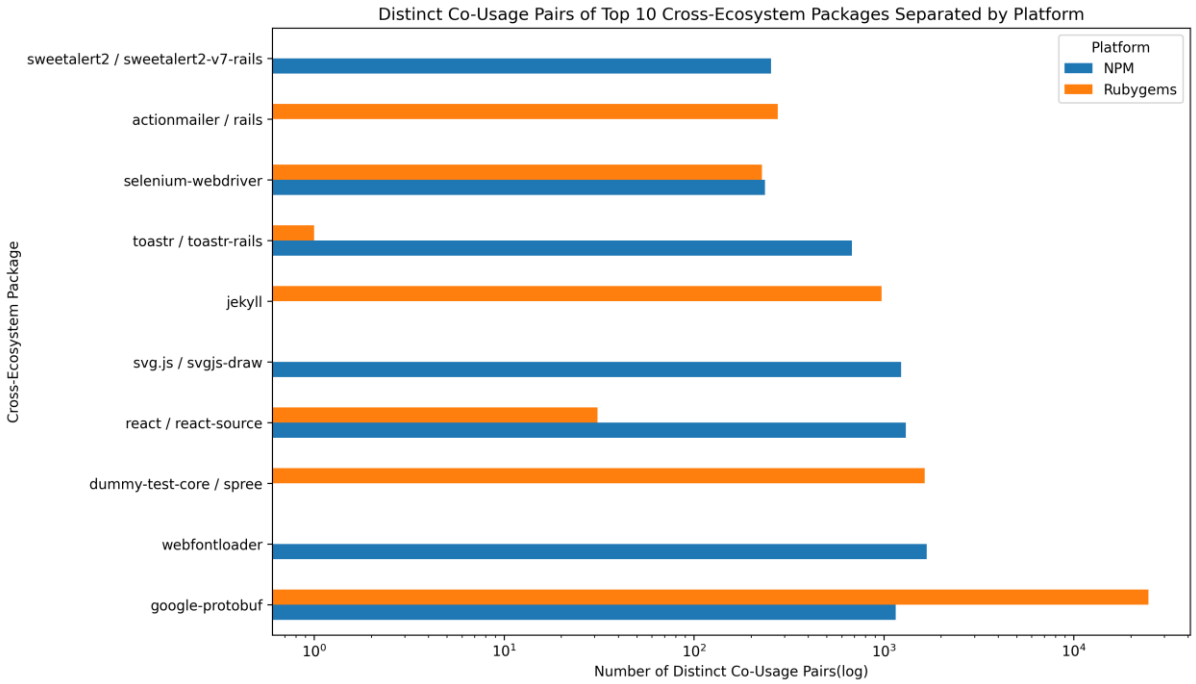

Figure 9: Top 10 packages with most distinct co-usage pairs

The results indicate a considerable number of unique method co-usage pairs generated by ‘google-protobuf’ in RubyGems even though it produced significantly fewer individual usages compared to npm. Although the npm package presents a lower range of diverse co-usage pairs, there is a noticeable difference between the numbers of individual distinct methods and co-usage pair variety.

Similarly to previous results, react/react-source’ and ‘toastr/toastr-rails’ showed higher distinct co-usage pair counts in npm than their associated RubyGems package. Moreover, the

‘selenium-webdriver’ had a balanced distribution of distinct pairs in both ecosystems. The remaining packages such as ‘jekyll’, ‘actionmailer/rails’, ‘dummy-test-core/spree’ were mainly associated with RubyGems while ‘webfontloader’, ‘sweetalert2/sweetalert2-v7-rails’ were primarily linked to npm.

5.3.3 Heatmap Analysis of Specific Packages

After examining the co-usage frequency and distinct co-usage pairs, concurrence heatmaps were used to provide a more detailed view of selected packages' method co-usages. These heatmaps show for each selected package the unique methods appearing within the ten most frequent co-usage pairs. The rows and columns represent methods that appear in those pairs, and each cell represents the usage volume of the corresponding pair. This made it possible to observe whether co-usage was concentrated around a small number of dominant pairs or distributed across several method combinations while also providing a view of the unique methods within these frequently used pairs.

The heatmap analysis focused on selected packages identified in the previous co-usage results, including ‘google-protobuf’ and ‘selenium-webdriver’. The selection was based on findings of previous results, and the reason is explained in more detail before their corresponding heatmaps.

Figures 10 and 11 present the heatmaps of ‘google-protobuf’ in each ecosystem. This package was investigated because it showed contrast in the between results. The main fact about this selection was the limited number of unique methods accessed and the large number of distinct co-usage pairs in npm. In addition, a considered factor was the lower individual frequency, along with a significant amount of co-usage pairs and their diversity on RubyGems.

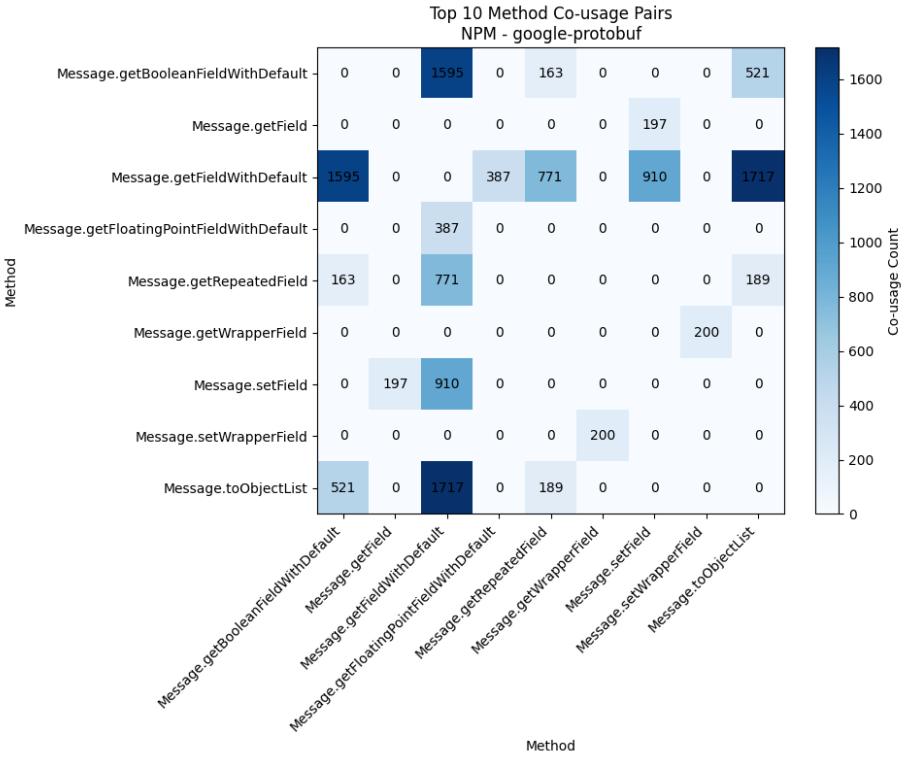

Figure 10: npm 'google-protobuf' heat map

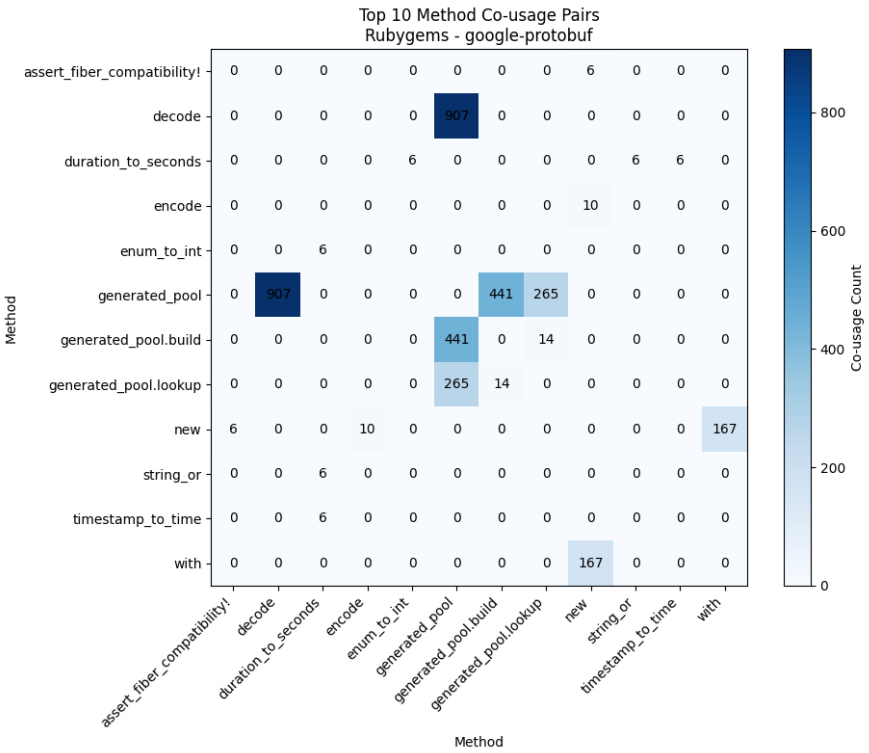

Figure 11: RubyGems 'google-protobuf' heatmap

The heatmaps show a clear difference in the distribution of co-usage pairs between the two platforms. In npm, the most used co-usage pairs are concentrated around a smaller group of methods. In contrast, the RubyGems heatmap displays limited usage among the pairs and a more scattered distribution across co-usage pairs.

The following figures present the heatmaps of 'selenium-webdriver' for npm and RubyGems. This package was selected because the previous individual method usage and co-usage results showed more balanced state across the two platforms.

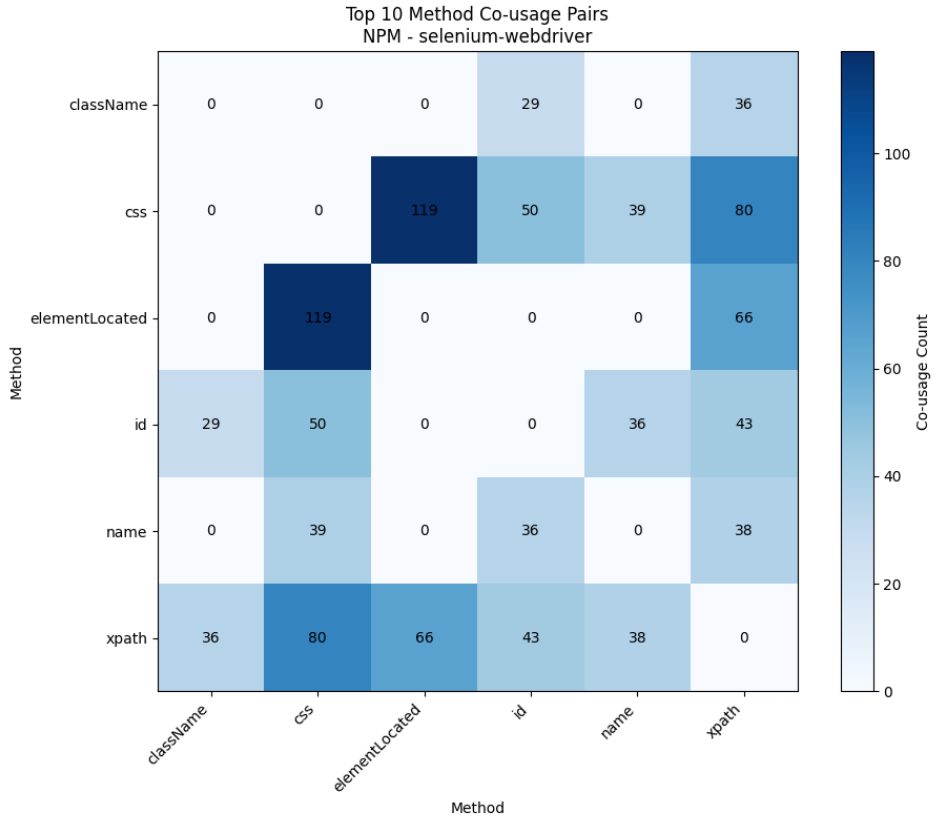

Figure 12: npm 'selenium-webdriver' heat map

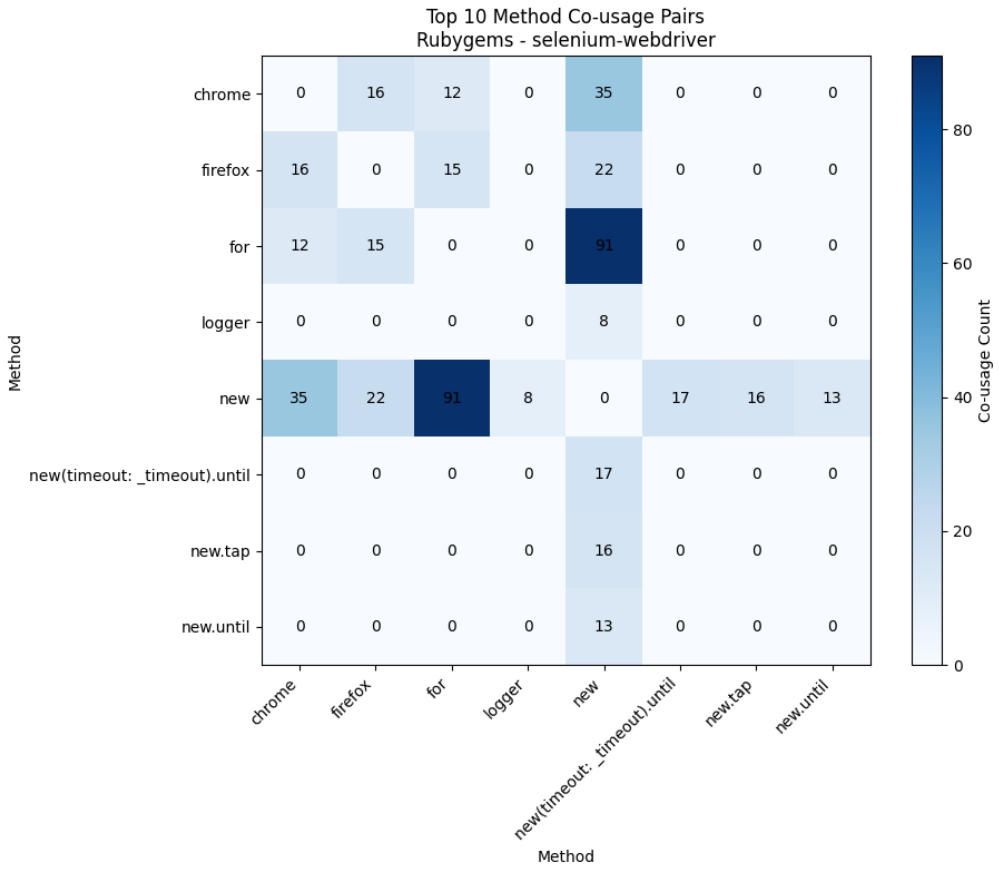

Figure 13: RubyGems 'selenium-webdriver' heat map

Figures 12 and 13 exhibit a more balanced co-usage pattern of 'selenium-webdriver' across npm and RubyGems. In both ecosystems, the co-usage pairs are spread across several method combinations rather than being concentrated around a single dominant pair. In addition, npm corresponding packages showed a smaller number of unique methods within the most frequent co-usage pairs. These results show that co-usage pairs are similarly distributed in both ecosystems, therefore reflecting the balanced state of the packages across all previous results as well as some methods that were slightly repeated in npm in the associated pairs.

5.4 Reference Forms Detected

Using the reference forms described in Tables 1 and 2 from the methodology section, the method usage records were examined according to the import or reference forms associated with each detected usage. This subsection aimed to identify the different types of reference mechanisms found in the dependent source code to present how the cross-ecosystem packages were introduced before their methods were invoked.

Import/Refence Form	Detected
require	Found
import	Found
Name import	Found
Namespace import	Found
Side-effects	Found
wrapper	Found
Import alias	Not Found

Table 5: Import/Reference Forms Description for JavaScript and Typescript

Import/Refence Form	**Description**
add_dependency	Found
add_runtime_dependency	Found
add_development_dependency	Found
require	Found
include/extend/prepend	Not Found
gem	Not Found

Table 5: Import/Reference Forms Description for Ruby

The detected reference forms indicate that cross-ecosystem packages were accessed via different mechanisms depending on the ecosystem and programming language. In npms' dependents, referenced packages were mainly introduced through import and require

statements, while in RubyGems, from required statements, dependency declarations, and no reference forms which may suggest an autoloading mechanism.

An important observation was that in npm multiple reference forms could appear within the same source file. This means that a dependent package may access a cross-ecosystem package through multiple mechanisms within the same file. The main combinations observed include

- Require + wrapper
- Import + name import + namespace import
- Import + named import
- Import + namespace import

5.4 Answering the Research Question

RQ: How do the dependents use cross-ecosystem packages

Dependents use cross-ecosystem packages in different ways depending on the package, the ecosystem, and the implementation context. The results revealed that dependents not only reference packages as dependencies, but also actively accessed them through method invocations in the source code files. Some packages were used very frequently, while others showed lower frequency, but a wider variety of method calls or co-usage pairs. The findings also show that high usage frequency does not always indicate that many different methods were used, since some packages are mainly accessed through repeated calls to a smaller set of methods. In addition, co-usage results reflect that dependent packages often use methods from the same cross-ecosystem package inside the same programming construct. These findings also revealed that packages with a limited number of accessed methods can still produce many co-usage occurrences, suggesting that the dependent packages combine unique methods within the same programming construct.

Section 6

Discussion

This chapter discusses the findings presented in the results section and explains their meaning in relation to the research question. While the results chapter presented the detected method usage, distinct methods, co-usage pairs, and reference forms, this chapter focuses on interpreting these findings.

6.1 Dependency Data to Method Usage

The results extracted in this thesis are associated with the cross-ecosystem packages method usage, derived from the data collected and analyzed. It is crucial to acknowledge that method usages may differ with different ecosystems and dependent packages due to several factors contributing to the research.

A key factor is the collection of cross-ecosystem packages and their dependents from npm and RubyGems. A different selection of ecosystems may generate different results regarding the frequency and diversity of method calls. Additionally, another main factor is the identification design for import forms and method usage. Different approaches may exhibit different results regarding method invocations.

In conclusion, the results extracted in this thesis provides a profound understanding on how cross-ecosystem packages are used by their dependents. These results reflect the method usage captured by the selected detection approach and may not cover all possible forms of package interaction.

6.2 Dependency Declaration and Source-Code Usage

These results suggest that declared dependency relationships do not always reflect the actual level of interaction between dependents and cross-ecosystem packages. Although the collected dependents were linked with the selected cross-ecosystem packages, many packages did not show evidence of interaction at source code level. This indicates that dependency metadata alone might not have actual usage, since a declared dependency does not necessarily mean that package functionality is directly accessed at the source code files.

This distinction is crucial because dependency presence and source code usage are not the same. A dependent package may be associated with a cross-ecosystem package for several reasons such as configuration, testing, or may have been declared at source code level and is not used. However, this does not clearly mean the package is accessed through method calls. Therefore, examining source code level usage provides a more accurate view of how the dependents interact with their cross-ecosystem packages.

Overall, declared dependencies should be treated as a starting point for analysis rather than proof of usage. The source code analysis added more detail by showing the reference forms and method interactions on the implementation level across cross-ecosystem and dependent packages. Therefore, source code analysis was necessary because dependency declarations alone could not fully explain how cross-ecosystem packages were actually used by their dependents.

6.3 Cross-Ecosystem Packages and Reference Forms

The examination of the reference forms provided insight into how the dependent packages grant access to cross-ecosystem packages functionality. As shown in the results, the detected reference forms varied between npm and RubyGems. This difference suggests that package import forms may be used differently across the two ecosystems. In RubyGems, the packages were often introduced through dependency declarations, which indicate that packages were made available at the project level rather than referenced separately in the source files.

Compared to RubyGems, npm provided an expanded set of import mechanisms. This reveals that npm dependents may introduce cross-ecosystem packages in more ways, using different

reference forms depending on the source file and the required functionality. An interesting observation was that in some files, cross-ecosystem packages were accessed through multiple reference forms. This suggests that npm dependencies were introduced in multiple ways within the same file, possibly accessing separate parts of its functionality or supporting different implementation purposes. This highlights that cross-ecosystem package usage in npm can be structurally diverse. As a result, detected reference forms showed that depending on how a package is accessed in the source code can directly affect how method usage is detected, traced and used.

6.4 Method Usage Frequency and Distinct Methods

Method usage frequency and distinct method count were the two main factors in analyzing method usage among the dependents. The method usage frequency measure focused on how many times methods from cross-ecosystem packages were invoked, while distinct methods centered on the number of unique methods recorded. Applying both measures provided the ability to distinguish between packages between high volume and broader diversity.

From the two measures the results indicated that some packages had high usage frequency with a limited set of distinct methods. In contrast, some packages showed less frequent usage but a larger number of distinct methods. This suggests that some packages may have repeatedly used a small set of methods, whereas others utilized more functionality with fewer calls. One probable reason for these differences in the findings is the importance of specific methods associated with the corresponding package functionality therefore some methods were continuously repeated. On the other hand, developers may access more methods that have a more specific usage, hence are used limited times.

The comparison of npm and RubyGems further supported the distinction between variety and frequency of method usage of these packages. For example, some packages showed higher usage frequency in one ecosystem, but in the other displayed more method diversity. This suggests that one platform may repeatedly use a smaller group of methods, while the other accesses more methods with lower frequency. In other cases, some packages exhibited both higher frequency and a bigger method set in one ecosystem. A third case was observed when method usage appeared mainly in one ecosystem, which reflects either stronger usage in one

platform or limited availability of dependent source code. These differences may be influenced by different factors. One possible factor is a higher percentage of source files written in the analyzed language as cross-ecosystem packages are multilingual as well as the number of declared available dependents. Additionally, the developing purposes of the dependent package may affect the diversity and frequency of method usage.

These findings provided valuable information because relying on both perspectives provides a better understanding of how the cross-ecosystem package use. A package may appear dominant in terms of total method usage, but distinct method results show whether the usage was concentrated or broad. By examining both frequency and diversity, the analysis provided a clearer understanding of how often cross-ecosystem packages are used by their dependents as well as how diverse the functionality was.

6.5 Method Co-Usage in implementation

The method co-usage results provided a different aspect of how dependent packages use cross-ecosystem packages. Although individual method usage shows which methods were invoked, co-usage shows which methods from the same package appeared together within the same code blocks. This aspect is crucial because methods are often used as part of programming constructs rather than as individual calls. Therefore, co-usage helps reveal how dependents combine methods of cross-ecosystem packages.

The co-usage results indicate that individual usage and co-usage do not follow the same behavior. Some packages that did not appear strongly in the individual method usage findings, appeared stronger in the co-usage analysis. This suggests that these packages may not have produce the highest number of individual method calls but their dependents combined multiple methods within the same code block. As a result, co-usage can reveal relationships that are not visible through individual methods. In contrast, packages who appeared strongly in the individual results also appeared strongly in the co-usage findings. This indicates that the packages were not only actively used, but that their methods were also combined within the same programming constructs during implementation. Moreover, the concurrence heatmaps showed that co-usage structure can either be concentrated around a small number of pairs or spread across multiple combinations.

These results may be affected by several elements related to source code composition. One important reason is how the method calls are organized in the programming constructs, for example functions, modules, and code blocks. In cases where several methods from the same cross-ecosystem package appear within the same block, this may generate more co-usage pairs. In contrast if methods are limited and repeated in these blocks, less co-usage pairs would be presented.

Consequently, co-usage analysis indicates that method usage pairs provide a better understanding of how dependents use cross-ecosystem packages during the implementation. The results show that packages are used only through individual calls, can be used in pairs that appear together in the same blocks. These combinations differed across ecosystems and packages showing that the same or related packages support different implementation behaviors in npm and RubyGems.

Section 7

Conclusion

7.1 Conclusion

The research investigated how dependent packages use cross-ecosystem packages at the source code level. The study focused on cross-ecosystem and dependent packages found in npm and RubyGems ecosystems. In this examination 10,369 dependents were analyzed and the associated methods of cross-ecosystem packages were extracted. The extracted method usage dataset was examined with the goal of providing insight into how the cross-ecosystem packages are employed.

The results showed that the usage of cross-ecosystem packages in their dependents cannot be precisely answered by just dependency metadata. Dependents may use these packages through different reference forms, repeated method invocations, distinct method accessed, and method co-usage. Furthermore, these results can be affected by several factors like source code composition, the package architecture structure and the significance of the access method from the cross-ecosystem package.

7.2 Future Work

Although this research provides insight how methods from cross-ecosystem packages are used individually and through co-usage pairs, several opportunities remain for future work. Future studies could build on these findings by applying additional analysis approaches, expanding the number of ecosystems investigating package usage in more depth. This qould provide a cleared understanding of how cross-ecosystem packages are used by their dependents.

References

[1] Constantinou, E., Decan, A. & Men, T. Breaking the borders: an investigation of cross-ecosystem software packages. arXiv:1812.04868v2 [cs.SE] 13 Dec 2018

[2] Orfanou, C. Evolution of cross-ecosystem packages , Undergraduate Thesis, University of Cyprus, 2024

[3] Chatzimylou, M. Social Aspects of Evolving Cross-Ecosystem Packages. , Undergraduate Thesis, University of Cyprus, 2025

[4] https://libraries.io

[5] Libraries.io Open Source Repository and Dependency Metadata https://zenodo.org/records/3626071

[6] https://zenodo.org/

[7] Jansen S., Finkelstein A, Brinkkemper S., A Sense of Community: A Research Agenda for Software Ecosystems 2009 31st International Conference on Software Engineering - Companion Volume

[8] Manikas K., Hanses, K. M. Software ecosystems – A systematic literature review https://doi.org/10.1016/j.jss.2012.12.026

[9] Lung M. Towards reverse engineering software ecosystems 2008 IEEE International Conference on Software Maintenance

[10] Lettner D., Angerer F., Prahofer H., Grunbacher P. A Case Study on Software Ecosystem Characteristics in Industrial Automation Software ICSSP '14: Proceedings of the 2014 International Conference on Software and System Process

[11] Hou F., Jansen S. A systematic literature review on trust in the software ecosystem Volume 28, article number 8, (2023)

[12] Joshua J., Alao D., Okolie S., Awodele O. Software Ecosystem: Features, Benefits and Challenges - Published 2013

[13] Jaisri P., Reid B., Kula R. B., A Preliminary Study on Self-Contained Libraries in the NPM Ecosystem arXci:2406.11363

[14] Mens T., Evolving Software Ecosystems A Historical and Ecological Perspective Volume 40: Dependable Software Systems Engineering

[15] Kannee K., Wattanakriengkrai S., Ropjpaisarnkit R., Kula R. G., Matsumoto K., Intertwining Ecosystems: A Large Scale Empirical Study of Libraries that Cross Software Ecosystems arXvi:2208.06655

[16] Intertwining Communities: Exploring Libraries that Cross Software Ecosystems https://zenodo.org/records/7553341

[17] Kannee K., Kula R. G., Wattanakriengkrai S., Matsumoto K Intertwining Communities: Exploring Libraries that Cross Software Ecosystems arXvi:2303.09177

[18] Yang H., Nong Y., Wang., Cai H., Multi-Language Software Development: Issues, Challenges, and Solutions

[19] Decan A., Mens T., Grosjean., An Empirical Comparison of Dependency Network Evolution in Seven Software Packaging Ecosystems arXiv:1710.04936

[20] Decan A., Mens T., Claes M., An Empirical Comparison of Dependency Issues in OSS Packaging Ecosystems University of Mons, Belgium