Thesis Dissertation

SOCIAL ASPECTS OF EVOLVING CROSS-ECOSYSTEM PACKAGES

Marina Chatzimylou

University Of Cyprus

COMPUTER SCIENCE DEPARTMENT

May 2025

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

Social Aspects of Evolving Cross-Ecosystem Packages

Marina Chatzimylou

Supervisor

Eleni Constantinou

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

May 2025

Acknowledgments

I would like to express my gratitude to my supervisor, Professor Eleni Constantinou, for her continuous guidance and tremendous help during this research. Her mentorship has been invaluable over the last year, and her feedback was always constructive and insightful. I am especially thankful to her for introducing this topic to me and for teaching me countless new things throughout this process.

I would like to extend my heartfelt thanks to my family and friends for their constant support, encouragement and understanding during this journey. Their presence helped me stay focused and motivated, even on the days that this research felt more difficult than ever.

Abstract

This thesis explores the social aspects of evolving cross-ecosystem packages, a relatively underexplored topic in software engineering. Cross-ecosystem packages are software components that are distributed across multiple software ecosystems. While prior research was focused on technical characteristics and usage patterns, this study aims to investigates the social aspects of the packages' development. A dataset of 381 repositories was analysed using Python-based tools and methods such as identity merging, bot detection, Truck Factor and Core/Peripheral developer classification. The results lead to eight distinct social patterns of developer activity, ranging from single developer maintenance to multi-developer collaboration with distinct language roles. Furthermore, a number of correlations and insights into how social structures influence software evolution were also found when the relationship between technical and social patterns was analysed. The study concludes with suggestions for future research directions.

Contents

Section 1

Introduction

1.1 Background and Motivation

Modern software development is heavily reliant on shared libraries and collaborative tools. These libraries and tools are typically centered around ecosystems that give developers the ability to share reusable components within their programming communities. Some well-known software ecosystems are NPM, Maven and PyPi. [1]

As time goes by, more and more projects aim to support multiple languages, resulting in the emergence of cross-ecosystem packages. These packages are distributed across multiple ecosystems and often share a unified codebase that is hosted on GitHub. They also maximize visibility and reusability across different platforms. [2], [3]

Even though their number is relatively small, they have a significand role in the software development community. They are broadly used across multiple ecosystems, and they exist as dependencies in many other projects. However, their maintenance introduces technical and coordination challenges. Each packaging system (ecosystem) has its own unique packaging format, metadata standards and many more. To successfully maintain a cross-ecosystem package, a developer must coordinate all these different elements while keeping the behavior and the functionality consistent across all supported ecosystems.

While studies explored the topic of cross-ecosystems packages from a technical view [4], [5], with patterns like ecosystem migration being discovered, human and social factors involved in

these packages' development have yet to be researched. Most prior work focuses on dependencies and usage metrics, failing to understand how developers interact with the system and their fellow contributors across ecosystems.

This thesis aims to minimize this gap through analysis of social structures and behaviors of the developers involved in each package, with the main focus being on how these structures and behaviors correlate with the technical evolution of these projects.

1.2 Methodology Overview

This study examines 381 cross-ecosystem packages that span through 5 major ecosystems: NPM, PyPi, Maven, CRAN, and RubyGems. The dataset was based on prior work by Orphanos [5], where he defined 7 distinct programming language usage patterns. In this thesis, the social factor is added by analyzing developer contributions per ecosystem over time.

GitHub repository metadata for each package were collected and processed using GitHub REST API and PyDriller. The metadata included commit history and modified files information. Then, a series of analysis steps followed with Identity Merging, Bot Detection, Truck Factor and Core/Peripheral classification being some of them.

The analysis resulted to the identification of 8 distinct social patterns. Some examples are "Single Developer," "One Developer per Language," and "Multiple Developers Leading the Project". These patterns were then compared to the programming language usage patterns found by Orphanos to try and understand the connection between them.

1.3 Thesis Structure

The thesis begins in Section 2 by introducing key background concepts essential to the understanding of the researched topic. Some concepts introduced are software ecosystems, along with their benefits and challenges, and cross-ecosystem packages. The significance of the 5 major ecosystems used in this thesis is also explained.

In Section 3, important previous related work is presented, covering both software ecosystems' and cross-ecosystem packages' research. This includes studies that define the characteristics and the challenges of software ecosystems and explore the dynamics of cross-ecosystem packages.

With Section 4 the methodology that was followed is explained in detail with all minor and major steps taken to retrieve the final results. All programs built and all manual interventions on the dataset are mentioned and explained.

In section 5, the 8 social patterns, that were discovered through the methodology process, are explained using usage scenarios and repository cases as examples. The distribution of technical patterns in the social pattern is mentioned as well. Finally, in this section the research questions of the thesis are described.

Section 6 includes the discussion of how technical and social patterns are connected and how a social pattern can explain a technical one. It is also discussed how the dataset is a key factor to the resulting patterns and how repositories are distributed across the observed social patterns.

Finally, Section 7 concludes the thesis by providing a summary of everything that was explored in the research and includes suggestions for ways to expand the topic in the future.

Section 2

Background Knowledge

2.1 Software Ecosystem Definitions

While there is still not one universally accepted definition for what software ecosystems (SECOs) are, many researchers provided their own interpretations on the topic and what it represents. Some gave a more technical definition with Lungu[1] stating that software ecosystems are "Collections of software projects that are developed and evolve together in the same environment", while some had a more abstract approach with Malcher et al.[6] defining software ecosystems as such: "A software ecosystem (SECO) is an interaction, communication, cooperation, and synergy among a set of players. Depending on the actor's type of interaction with others, each one can play a role.".

Manikas and Hansen [7] conducted a systematic literature review (SLR) of the research done on software ecosystems. In their review, they collected and analyzed all the definitions of the term SECO and summarized them in one. In all the definitions they analyzed they noticed three main elements: Common Software, Business, Connecting Relationships. By combining these elements, the resulting definition is: "The interaction of a set of actors on top of a common technological platform that results in a number of software solutions or services. Each actor is motivated by a set of interests or business models and connected to the rest of the actors and the ecosystem as a whole with symbiotic relationships, while, the technological platform is structured in a way that allows the involvement and contribution of the different actors.".

2.2 Software Ecosystem Characteristics

Even though software ecosystems have multiple definitions in the literature, they still share a set of common characteristics. To this day, there has been limited research on the characteristics of SECOs. Lettner et al. [8] analyzed several studies done on SECO and managed to identify the following characteristics:

1. Actors & Participation

- Internal and External Developers/Platform Enabling Outside Contributors: The development is made available for both internal and external developers to work on, each of them possibly having a different goal for working on the project.
- Outside contributors included in main platform: Suggestions done by third-party contributors can be integrated into the main platform.

2. Architecture and Core Structure

- Common technological platform/Shared core assets: All contributors share a common technological basis that also includes reusable features/components, already set standard solutions, and a framework solution.
- Controlled central part: Internal developers are responsible for developing and controlling the platform's core, while external developers use the platform to work on client applications.

3. Variability and Derivation

- Variability-enabled architectures: The platform allows the application development to be tailored to the user's needs and the external developers can use core assets to do it.
- Automated and tool-supported product derivation: Automation and tool-supported mechanisms are used to develop a product, giving the ability to external developers and users to customize their product themselves.

4. Tools and Infrastructure

- Tools, frameworks, and patterns: They support the developers in managing the development and the evolution of the platform along with its elements.
- Distribution channel: An interface that allows external developers and users to use the platform. External developers submit their solutions in the channel, and the users search and get the solutions they need.

2.3 Software Ecosystem Benefits & Challenges

Software ecosystems have both positive and negative aspects. Neither of them outweighs the other, and both are equally important in understanding software ecosystems.

Starting with benefits, software ecosystems provide an effective way to minimize substantially the cost of software development and its distribution. Due to their collaborative nature, they allow the developers to share their knowledge and their work with each other. This way, code reuse can be used more frequently to decrease the production time, consequently decreasing the development cost. SECOs also promote the successful cooperation between actors to develop and grow software by providing a common platform and preexisting knowledge. [9]

On the other hand, creating a positive environment between the actors can be quite challenging. Everyone involved has their own knowledge and background, making it hard sometimes to find common ground. Keeping the relationships between the developers well defined and representing their individual knowledge effectively in ecosystem models is particularly important for the longevity of the project. [9]

Other challenges that have been identified concern more technical aspects. Some key concerns are in the architecture: maintaining the stability of platform interfaces, supporting their evolution, and many more. Also, when it comes to organizations, they must avoid heterogeneity of software licenses and systems evolution so they can reduce dependency risks and keep their documentation up to date with the law. Organizations are also concerned to find ways to differentiate themselves from their competitors and stand out in the market. [9]

Focusing again back on software ecosystems' collaborative nature, most of the time ecosystems contain geographically distributed projects. Technical and socio-organizational barriers can pose a great challenge when it comes to collaboration and communication among the developers. This can cause some setbacks in the development process and negatively impact the efficiency. Also, the lack of sufficient infrastructures and tools that promote social interactions, decision-making, and development in both open-source and proprietary ecosystems can slow down the development and weaken the community engagement. [9]

2.4 Important Software Ecosystems

This thesis focuses on 5 main ecosystems: Maven, PyPi, NPM, Rubygems, and CRAN. Recent surveys and studies show that JavaScript, Python, and Java are among the most popular languages used by developers all over the world [10]. Table 2.1 shows the popularity per language in 2024 based on a survey conducted by StackOverflow. The importance of the aforementioned languages in the programming world is shown by their high ranks for the past few years [11]. Even though R and Ruby are not as popular, see the last and third to last rows in Figure 2.1, they were also chosen because of their prevalence in our cross-ecosystem package dataset.

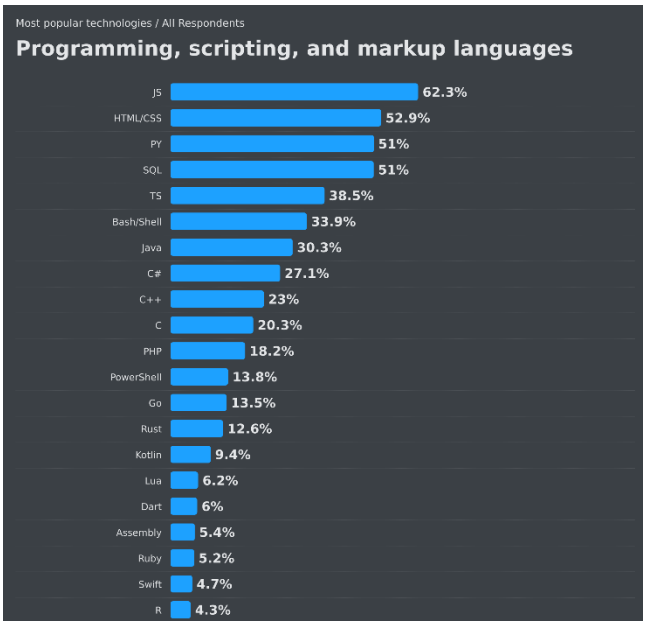

Figure 2.1 Language popularity [12]

In addition, their corresponding packages (Maven, PyPi, NPM, Rubygems, CRAN), considering their package volume, are among the most widely adopted software ecosystems by developers [13].

Maven (Java): Maven is a build automation and project management tool mainly used for Java projects. Its main purpose is to simplify the process of building, packaging, and managing dependencies by using a standardized approach based on the Project Object Model, also known as POM. To use Maven, developers use the mvn command-line tool (CLI). [14]

PyPi (Python): PyPi, short for Python Package Index, is the official repository of software for Python. Its main purpose is to serve as a central hub for developers to discover and download open-source Python packages by entering a keyword in the search bar. Developers can also

upload their own packages onto the platform. To interact with the platform, developers must use the pip CLI. [15]

NPM (JavaScript): NPM is the package manager for Node.js. Just like PyPi, it is a public registry for open-source Node.js packages, available for developers to upload and update their own, find ones they are looking for, and download them by using the npm CLI [16]. With over 5 million packages available, NPM is currently the largest software registry in the world. [13]

Rubygems (Ruby): Rubygems is the package manager for the Ruby language. It is used to distribute, manage, and install gems (libraries) that extend functionalities in Ruby applications. Each gem contains a Ruby program or a library along with its metadata. Gems can be installed using the gem CLI. [17]

CRAN (R): CRAN stands for Comprehensive R Archive Network. It is the main repository for R packages, and it is used to install, update, and access a variety of statistical and data analytics tools, as well as other libraries that extend the core functionality of R. The only way to interact with CRAN is through R or RStudio. [18]

2.5 Defining Cross-Ecosystem Packages

Until now we defined software ecosystems describing them as self-contained communities that focus on a single language or platform. Cross-ecosystem packages take this concept a step further and are described as software components that are published across multiple software ecosystems, such as PyPi, Maven, and others.

They are typically hosted on a common platform, such as GitHub, that allows them to share a common source code base. To have this shared repository can make coordinated development and versioning across ecosystems a lot simpler for the developers involved. Additionally, for the packages to work in this multi-ecosystem environment and with each ecosystem having its own technical requirements, the package is adapted and released in all the different formats that are required so it can fit into each unique package management system. [2], [3]

Cross-ecosystem packages tend to have more community engagement compared to regular packages in the ecosystems, with more stars being noted, more forks, and more contributors [2]. This heightened attention though, is not always focused equally on all the languages or platforms that the package is distributed to. This case of asymmetry in ecosystem support

usually results in one ecosystem being maintained or updated more than the others and creating a huge difference in the percentage of appearance of each ecosystem in the package. [2], [3]

In conclusion, cross-ecosystem packages are an important evolution of software ecosystems and their impact on modern software development is vital. Their ability to enable code reuse and functionality sharing between ecosystems pushes the limits of what software development can reach. By researching and understanding this topic and its trends, we can find valuable ways to support the ongoing evolution of software development and contribute to the limited but enlightening research that has been done up to this day on these specific packages.

Section 3

Related Work

3.1 Research on Software Ecosystems

Over the past 2 decades, software ecosystems have been a hot topic for research communities with the first to introduce the term being Messerschmitt & Szyperski[19]. In 2005, the topic gets academic attention and today there are more than 900 publications indexed.[6]

Manikas and Hansen [7] conducted a systematic literature review on published SECOs research papers. They analyzed the SECO definitions across literature and created a definition that summarizes all different versions that exist, studied publication trends and looked for types of contributions in SECO research. On the other hand, Malcher et al. [6] did a tertiary study on 22 secondary studies on software ecosystems and built a thematic model that organizes the research into 5 major areas: social, technical, business, management, and assessment.

Lungu [1] worked on reverse engineering SECOs so he can understand them better by getting a single project and extending it to the abstraction level of a software ecosystem. He focuses on the structure of the projects, their dependencies and the behavior of the developers. Lettner et al. [8] conducted a case study on an industrial automation software ecosystem (ISECO) aiming to identify the characteristics SECOs have in industrial settings. They extract already mentioned traits from literature and propose their own that are based on the analyzed setting.

3.2 Research on Cross-Ecosystem Packages

Cross-ecosystem packages are a recent term in the research community. There has been little published research and what is known on the topic is very limited. Fortunately, the few

published works are very insightful and helpful for this research and provided great help in understanding the concept of cross-ecosystems.

Constantinou et al. [2] investigated cross-ecosystems and their characteristics across 12 ecosystems. They found that cross-ecosystem packages are significantly smaller in number compared to the total amount of packages included in each ecosystem. Despite their limited number, these packages have a very big impact on the rest of the ecosystems due to dependency links they have across multiple packages. Another conclusion in their research is that cross-ecosystems packages usually span to 2 ecosystems and developers have the tendency to maintain only one of them better. The research highlights the impact of cross-ecosystems packages and some potential risks that can appear due to the asymmetrical maintenance of the ecosystems, an example being security risks due to old, unmaintained dependencies.

Kannee et al. [3] investigated cross-ecosystem packages to find the depth of which libraries intertwine ecosystems in terms of dependencies and contributions by developers. This research complements the research done by Constantinou et al. [2]. They focus on 5 ecosystems and their findings show a large number of dependencies between 4 out of 5 ecosystems with some of them exhibiting higher dependence levels. This leads to the conclusion that these libraries tend to have more dependencies compared to libraries that come from a single ecosystem. They also studied the contribution activity of the developers and found that the majority of them only work on a single ecosystem, while a significant number of developers belong to none of the ecosystems. On the other hand, only a small number of developers are active on all ecosystems.

Paramitha et al. [4] aim to create a language-agnostic categorization protocol that enables the analysis of software in cross-ecosystem packages. Their goal is to find common ground between ecosystems with different functional purposes and allow them to be compared on the same categories. They selected Java libraries that had known security issues and had 2 assessors to manually determine the categorization of each library. When they disagreed, a third assessor determined the category of the library. Through this whole process they managed to categorize Java libraries based on Python categories (PyPi topics). They want this manual process to be foundation for security analysis between ecosystems, training automated tools to categorize packages from multiple ecosystems and make analysis and comparison between packages/libraries from different ecosystems easier for developers.

Orphanos [5] analysed a set of cross-ecosystem packages that are distributed across 5 major ecosystems: Maven, NPM, PyPI, RubyGems, and CRAN. The purpose of his thesis was to extract programming language usage patterns, to uncover trends and understand the

behavioural patterns and evolution of cross-ecosystem packages. His research led to the discovery of 7 distinct patterns: Base language with support of other language, Parallel support in both languages, Interchanging support in both languages, Language Migration, Attempt Success, Attempt Failure, Unclear Pattern. The recovered patterns are explained below.

Pattern 1: Base language with support of other language
This pattern reflects projects with one language more dominant than the other. The rest of the languages show light support compared to the base language.

Pattern 2: Parallel support in both languages
This pattern reflects projects that have similar activity over time in both languages. There is no dominating language.

Pattern 3: Interchanging support in both languages
This pattern reflects projects that do not have a dominating language, but each language has higher commit frequency on different time periods than the other. There is a "switch" in the attention given to each language.

Pattern 4: Language Migration
This pattern reflects projects that have a successful transition from one main language to another. There is a period where they overlap.

Pattern 5: Attempt Success
This pattern reflects projects that have a successful integration of a new programming language into the project's development workflow.

Pattern 6: Attempt Failure
This pattern reflects projects that fail to integrate a new programming language into the project's development workflow resulting to its abandonment.

Pattern 7: Unclear Pattern
This pattern reflects projects with unclear language patterns. They show lack of notable trends or consistent usage dynamics, making it impossible to sort them into a particular category.

Section 4

Methodology

This section outlines the methodology process that was followed to retrieve the results of this research. Steps like data collection, data formatting and data cleaning were necessary to ensure reliable and valid results. The main goal was to find social patterns based on developer activity from selected repositories. The selected repositories represent cross-ecosystem packages that are distributed across five major software ecosystems: Maven, NPM, PyPi, RubyGems, and CRAN. All the scripts that were developed during this process are written in Python, a programming language that is widely used in mining software repositories and data analytics.

4.1 Data Collection

Before starting the collection of the initial data that were needed for this research, the first thing that had to be done was to determine which repositories were useful for the study and could be considered as cross-ecosystem packages. To achieve this, prior work by Orphanos [5] was

used, where the focus was on the 5 previously mentioned ecosystems. The cross-ecosystem packages and their technical patterns were obtained from the replication package developed by Orphanos which was based on data originally collected by Kannee et al. [3] during his study. This dataset was chosen because it included the technical patterns that are required in the analysis phase of this thesis.

The CSV that was provided by Orphanos included the Repository URL, the Platform Count that represented in how many ecosystems the package is found, the Platforms it is part of, and the Pattern that Orphanos categorized it in during his research. For this first step of the process, the only information that was needed was the Repository URL and it was used to extract the commit information of the projects.

To retrieve the complete commit history of each repository, the GitHub REST API[1] was utilized. A personal token was required for authentication and as input was given the CSV that is mentioned previously. The commits were extracted using pagination, up to 100 commits per request, to eliminate any issues regarding the APIs rate limit. Commit history by default is extracted in nested Json format. Each output was flattened using the pandas.json_normalize function so commit fields like author_login and commit_author_email could be easily stored, accessed, and processed in a structured CSV file. By the end of the script execution, each repository had its own individual CSV file containing all the extracted commit metadata provided by the GitHub API. The following fields are selected from the complete set of extracted data for further analysis, focusing only on information needed to the study:

- Commit information
    - sha, commit_message, url
- Author and committer information
    - author_login, author_type
    - committer_login
- Commit author and committer information
    - commit_author_name, commit_author_email, commit_author_date
    - commit_committer_name, commit_committer_email, commit_committer_date

Each of these fields has a vital role in understanding the commit process and each user's role in the project. Each commit has a unique identifier (sha), a commit message that explains the changes that the commit introduced and a url that acts as a link to the commit on GitHub. Along with the standard commit information, the data extracted provide an insight in the users

[1] https://docs.github.com/en/rest

involved. Each author and committer have their own unique identifiers (login), along with their user type (User/Bot), name, email address and the date they completed their respective role in the commit process. Author is the one that wrote the code or did the changes, while committer is the one who saved the changes in the repository, meaning that they did the commit. In most cases, the two are performed by the same person.

Now that each repository's commits were extracted, they could be used to extract the files that were modified in each commit and their corresponding details by utilizing the PyDriller library [20]. To be able to get the modified files data, the repositories had to be cloned locally and by traversing each commit all recorded data on the modified files were saved in a zipped CSV file. The zipped CSV format was used instead of a regular CSV for better storage management, as in some cases the extracted DataFrames could reach thousands of rows. The saved CSV files included the following fields of interest among others:

- Commit Information
    - Commit Hash (previously mentioned as sha)
- File Modification Details
    - File Name
    - Added Lines
    - Deleted Lines

The modified files metadata dive deeper in terms of the information obtained on commits, with each row of the final CSV giving information on each file changed in each commit of the repository. For each commit, identified by the Commit Hash (also referred to as sha during the commit extraction), the dataset includes the File Name, the lines added, and the lines deleted from the file.

During the extraction process, a few errors ultimately led to the exclusion of some repositories from the dataset. The said errors were "Memory Error", "CalledProcess Error" and "Value Error". Memory Error happened because some DataFrames were too large for the memory to process. CalledProcess Error usually happened because the repository that PyDriller was looking for was no longer available. Finally, Value Error happened when repositories had incomplete or invalid Git histories causing PyDriller to raise an error during the traversal of the repository. Out of 524 repositories that were provided for data extraction, only 386 were successfully processed.

4.2 Formatting Initial Data

After collecting the raw commit and modified files data, the next step was to format them to get a unified dataset that was suitable for the following steps of analysis. This step includes merging datasets and enhancing them with other important attributes.

The first step of this section involved establishing the base structure of the dataset. The CSV extracted from the file modification data was set as a base for selected column from the Commits CSV to be merged into. The columns that were merged into the modified files CSV were author_login, commit_author_email, commit_committer_date and url. The merging was done based on sha which ensured that each of the merged columns were matched to their respective modified files.

Next, some additional processing was done to enhance the dataset. One key information that was missing from the merged CSV was the language of each file. The language is very important in the research since it is the metric to categorize the cross-ecosystem package into its ecosystems. The language was extracted from the File Names by analysing their extensions and matching them with their language by looking into a language map. The mapping includes two columns, one for the Extension and one for the Language it represents. Additionally, the commit date was normalized in the format YYYY-MM-DD.

A subset of the data that were enhanced, were saved in a new CSV file. The fields included in the CSV are: Commit Date, Project, author_login, commit_author_email, File Name, Language, Added Lines, and Deleted Lines. Project was constructed by the url of the repository to keep only the section author/repository_title to be able to recognize which repository each file is part of. This more refined subset was created to better understand how files were changed, which languages were used, and who made the changes. During this process, 4 other repositories were removed from the database due to an issue with their commit metadata missing key elements, leading to 382 repositories being left for processing.

4.3 Identity Merging

To ensure an accurate representation of the contribution of each developer, it is important to make sure that all different aliases of the same physical person are discovered and merged into one. This process is widely known as Identity Merging.

Over the years several identity merging algorithms have been published, such as Bird et al. [21], Kouters et al. [22] and Geominne and Mens [23]. This thesis follows a hybrid approach with manual inspection of the data to avoid common errors like overlooked matches while following the existing algorithms' steps.

The merging process begins by loading the commit data of the repository that is processed at the moment. The information that is used during this process is the author's and committer's name, email and login. The emails are normalized to a common format to avoid emails that use different forms of the at sign ("@"). Then all names, emails and logins are tokenized, removing commonly used terms like generic domains (e.g. "gmail.com") and keeping only the terms that are important to create the identity map.

The strategy that is used to identify potential identity matches is a token-based similarity approach. Potential identities are constructed by comparing the overlap between the tokens of the identity that is processed and the other identities. If a sufficient number of common terms is found the identity and its information are proposed as a potential match followed by the compatibility distance calculated using Levenshtein distance.

The matching process is divided into 3 parts: author-based identity suggestions, committer-based identity suggestions and all logins appearing in the repository. Author-based identity suggestions are potential author identities that match, committer-based identity suggestions are potential committer identities that match, and all available logins are all author logins that are found in the CSV file. The user is asked in all 3 stages to enter a login for each identity that does not have an identity in the existing data. If no matching identity was found after all 3 stages, then the author's name was added as the login.

The resulting merged identities are added as a separate column in a new CSV file along with other commit information like sha, author and committer names, emails, dates, types and logins.

4.4 Bot Detection

Bot detection is another important step in data cleaning. Automated Bots can resemble regular human developers in terms of naming and activity patterns, making them indistinguishable at first glance. Including these accounts in the analysis can lead to inaccurate results that do not represent true human interactions. To eliminate such cases, an adapted version of the BoDeGiC Bot Detection tool was employed to better fit the structure and the need of this study. The tool

was originally developed by Mehdi Golzadeh during his PhD research at University of Mons as part of the SECO-ASSIST Excellence of Science initiative [24], [25]. The modifications made are explained at the end of this section.

The tool predicts if an identity is User or Bot by analysing their commit message patterns that extracts from the cloned repositories. For each contributor it calculates features, for example Number of unique message patterns and Gini index (message diversity), that will help during the prediction process. Just like in the merge identity process the messages are compared using overlap between sets of words (Jaccard similarity) and Normalized Levenshtein distance that calculates how many edits are required to get one message and transform it to another. The application of these measures provides a clear view of which messages are similar or repetitive. To group the similar in context messages, the DBSCAN (Density-Based Spatial Clustering of Applications with Noise) algorithm is applied to identify the distinct "patterns" that a contributor follows in their commit messages. The fewer the patterns the most likely to be a Bot, as they are known for their repetitive and template-like messages.

Once all the features are calculated they are inputted in a pre-trained machine learning model that was trained to distinguish Bots from Users. The results are saved back to the dataset as a new column that separated the predicted types from the preexisting ones.

To ensure that no contributor got assigned more than one type, the tool was modified to group all developers with multiple aliases as singular entities using their merged identities. If an identity was already assigned a type, then all other identities were assigned the same and the contributor was excluded from the prediction process. Also, the tool was adapted to automatically assign the type of User to contributors with less than 10 commits, a predefined threshold by the original developer for more accurate predictions to avoid false positives since it is unlikely that bots contribute this little in a project.

4.5 Data Setup for Analysis

At this stage, the dataset was almost ready for the final phase of the analysis. The only thing that was left was to be formatted correctly to match the expected input that the tools used in the following steps required. To get the dataset to the desired format, actions like merging columns from multiple files and formatting datasets were implemented.

The first step to getting the desired format was to create the base dataset with the following columns: hash, author, date, added, removed, current, author_type, Language. To get this dataset, final data from multiple previous steps needed to be merged into one common CSV. Key datasets in this process were the modified files metadata, the final identities after identity merging and bot detection, and the initial data after formatting. Before any columns were merged, all contributors flagged as Bot were filtered out. This ensured that only human developers were included in the following analysis, allowing the results to accurately reflect human behavior.

Considering that almost none of the expected columns exist in the input CSVs with these headers, the columns that were merged had to be temporarily renamed to fit the desired format:

- Modified Files:
    - Commit Hash => hash
    - Added Lines => added
    - Deleted Lines => removed
    - File Name => current
- Final Identities:
    - merged_identity => author
    - commit_committer_date => date
    - author_type => author_type
- Formatted Initial Data
    - Language => Language

All columns, except Language that was matched based on the File Name, were matched based on the Commit Hash. The final merged results were saved in a new CSV file.

The final step in formatting the new dataset was to keep only the commits that contributed to the languages relevant to each repository's ecosystems and group them based on the month they were committed. All other commits done on unrelated languages were filtered out to keep the results focused only on the ecosystem related contributions. The repositories were mapped to their known ecosystems through a reference CSV file that included information on the ecosystems each repository contributed to. Then, the filtered data were grouped based on the Year-Month section of the date and saved in a new CSV file that would be used as direct input in the Abandonment Threshold, Truck Factor and Core/Peripheral calculations.

4.6 Abandonment Threshold

A key aspect of understanding developer contribution behavior in an ecosystem is detecting when a developer abandons the language. This section outlines how the abandonment threshold for developer inactivity was identified. The methodology follows the Abandoner Threshold Sensitivity Analysis introduced by Avelino et al. [26] in their research on the abandonment of open-source projects.

The script's main purpose was to compute the activity gaps of all developers across all projects. To calculate the durations of all the gap between each pair of consecutive commits, the script extracted the commit dates of all contributors and when one's total commits were less than two the developer was excluded from the process.

To get more accurate results, multiple thresholds were tested: 1, 3, 6, 9, 12, 18, 24 months. Each threshold was evaluated based on the metrics Precision, Improvement and Harmonic Mean [27]. Precision calculates how many active developers were identified correctly, Improvement calculates how many previously misclassified developers were correctly identified in the current threshold and Harmonic Mean balances the two metrics to calculate one final score that reflects the overall effectiveness of each threshold.

Threshold (months)	Precision	Improvement	Harmonic Mean
1	0.4582	0	0
3	0.5738	0.2134	0.3111
6	0.6808	0.2511	0.3669
9	0.7494	0.2148	0.3339
12	0.8035	0.216	0.3404
18	0.8656	0.3161	0.4631
24	0.9038	0.2838	0.4319

Table 4.1 : Abandonment Threshold Analysis Results

Among all the thresholds that were tested, the one that scored the highest in Harmonic Mean and was chosen as the abandonment threshold was 18 months, as seen in Table 4.1.

4.7 Truck Factor

The concept of Truck Factor (TF) refers to the minimum number of developers that need to be hit by a truck (stop contributing) for the project to be in serious danger of abandonment. They are the developers that hold the knowledge of the project, and their departure leaves the project vulnerable. This metric is used to determine how prepared a project is to deal with this situation.[27]

In this research, an existing implementation of the Truck Factor algorithm [28] was adapted to match the circumstances and requirements of this study. The algorithm that was used is developed by HelgeCPH as a part of the truckfactor PyPi package [29] and draws inspiration from Avelino et al.[24] research on calculating Truck Factors.

The modified script processes the data that have been separated by month and language during the previous steps. For each month, the main developer(s) of each file are identified by calculating their total contribution to it, up to that point of time, based on the number of lines they added to it. The developer(s) with the highest contribution are considered the main developer(s). Once this process is complete, each developer is assigned how many files they are responsible for as the main developer.

Truck factor is determined by discovering the smallest group of developers that combined own at least 50% of the files. This calculation is done for each calendar month that the project had commits. The logic allows developers that are not currently active (on the month processed) to still be attributed to the Truck Factor title when no developer owns more files than them and is able replace them.

To know which developers are no longer active, the abandonment threshold was implemented. The Truck Factors each month are given 1 of 3 states: Active, Inactive or Abandoned. When the TF contributed that month was given the state Active, when they did not contribute but they had activity in the next month or their activity gap was less than the threshold they were given the state Inactive and when they had no future activity and their activity gap was longer than the threshold they were given the state Abandoned along with the language(s) they stopped contributing to. The output was saved in a separate CSV file, and it contained the Truck Factors and their state for each month.

4.8 Core/Peripheral Algorithm

To have a better understanding of the main developer dynamics in the ecosystem, the Core/Peripheral Classification approach was implemented. This approach states that a small group of Core developers has 80% of the total activity while the rest 20% is taken by less active, Peripheral developers. This logic was introduced by Robles and Gonzales-Barahona [30] their study on Libre projects and developer turnover.

The classification is performed by analyzing the commit frequency per month of all developers in the languages involved. Each month, each developer's number of commits is computed and once all developers' activity is gathered, developers are sorted in descending order based on their number of commits up to that point in time. Their combined cumulative activity is then calculated and once the cumulative activity reaches 80% the process stops. The subsets of developers that their activity was part of that percentage are returned as the Core developers while the rest as Peripheral Developers for that month.

Just like Truck Factor, the developers are given states of activity based on the abandonment threshold, as again when no new Core Developer is active enough to replace an inactive Core Developer, the inactive one appears as Core in their inactive months as well.

The final results are saved in a new CSV file that includes all Core and Peripheral developers per month, as well as their state of activity.

4.9 Final Data Formatting

The final step of the Data Preparation process was to unify all the key data together for each repository. These data include the Truck Factor and Core/Peripheral output as well as the final dataset that was used by Orphanos [5] during his research of programming language usage patterns. During a manual inspection of the dataset provided, it was noticed that 1 repository did not exist, resulting in its exclusion from the database. The final number of repositories available for analysis is now 381.

Each dataset was individually added and formatted to align based on the Year-Month column. Each one was given a unique extension like (language)_Core, (language)_TF and (language)_Technical for better distinction in data analysis.

To improve readability, a basic coloring pattern was followed for all datasets. When the month was not empty, meaning that the dataset had a value for that month, the cell was shaded green, and when the cell was empty, meaning no activity that month, it was shaded red. This method allowed a more comfortable view of the combined data by eliminating the overwhelming sensation that large amount of data with no prominent features give. An example of the combined data is presented in Table 4.2.

Year-Month	Javascript_Core	Python_Core	Javascript_TF	Python_TF	Javascript_Tech	Python_Tech
2017-06	cekees	cekees	cekees	cekees	No Gap	No Gap
2017-09	cekees	cekees	cekees	cekees	No Gap	No Gap
2019-12		cekees, davidbrochart		davidbrochart	Gap	No Gap
2021-04	cekees, davidbrochart	cekees, davidbrochart	cekees	davidbrochart	Gap	Gap
2017-07					Gap	Gap
2017-08					Gap	Gap

Table 4.2: ipymesh Repository Final Data Format example

Table 4.2 shows the formatted final data of the ipymesh[2] repository. Each row represents a Year-Month period, providing information on both Javascript and Python ecosystems. The "Core" columns list all Core developers that were identified during the Core/Peripheral Classification process while the "TF" columns list all developers detected as Truck Factors during that time period. The "Tech" ("Technical") columns represent the commit activity, "Gap" for no commits and "No Gap" for commits, of each period and were used by Orphanos [5] to detect the technical patterns. In this case, the columns are used for comparison of the social and technical aspects and as a tool to assist in the detection of the social patterns.

4.10 Pattern Detection

To extract possible social patterns, a manual inspection of all final results was done. All noticeable characteristics found that could lead to candidate social patterns were written down. This process was time consuming due to the volume of the data that was needed to be manually inspected so a script was created to help with the pattern detection process. When all final patterns were confirmed, another script was constructed to automatically assign the repos their matching social pattern.

[2] https://github.com/erdc/ipymesh

4.10.1 Developer Activity Percentage Calculation

The first script was created to assist during the configuration of the final set of social patterns detected. Based on its final results, multiple patterns were extracted and confirmed. Its main purpose is to calculate each developer's relative contribution per language.

For all repositories, their formatted final data are given as input to be further analysed. It focuses only on columns including the _Core and _TF extensions, with all _Technical columns being discarded from the dataset to be analysed. Terms such as "Gap", "No Gap" and NaN were also discarded keeping only months that at least one developer was active (green cells).

For each developer, their active months, either as Core or TF, are calculated and then divided by the total months that the language was active. This division returns a percentage indicating the developer's presence in the language and their role in the project. Developers with activity percentage greater than 90% in a language could be considered a strong contributor to that language while a developer with less than 25% contribution could be considered a minor or occasional contributor. Cases where a developer was a strong contributor to all languages were also found with these developers being considered as strong contributors in the project and not just in a specific language.

The results were compiled in a separate CSV file per repository, providing a clear view of each developer's distribution of effort in the project. The following table, Table 4.2, provides an example of the output results of the script.

Language	Developer	Contribution (%)
Javascript	bruth	100
Javascript	naegelyd	29.63
Python	bruth	100

Table 4.2 Activity Percentage Calculation example

With the final data presented in this format, the pattern detection process became significantly more manageable and less time consuming. The process resulted to 8 social patterns, that will be further explained in Section 5.

4.10.2 Repository Social Pattern Assignment

With all social patterns found the next step is to classify each repository with its right patterns. For each pattern, distinct criteria were set in place to easily be able to categorize the repositories. Due to some patterns not having uniquely distinct criteria and repositories matching more than one patterns, a pattern hierarchy was implemented to ensure that each repository was assigned to only one pattern. The hierarchy was carefully constructed to give the best results in terms of pattern match based on the actual repository activity.

Just like the Activity Percentage Calculation script, this script has the final formatted data per repository given as input and all columns other than the Year-Month, Core and TF being ignored. For each repository the script calculates 3 metrics: the total number of active months per language, the set of contributors active in each language, and the specific months in which each contributor was active. These 3 metrics are used in all pattern detection methods to determine whether a repository matches the current pattern criteria. If the repository is a match, the pattern is logged in a CSV file and if it is not a match, it moves on to the next pattern detection criterion until the repository is matched.

During this process, some errors were noticed in a small number of repositories where manual intervention was needed. In some cases, the bot exclusion during the data setup for analysis process did not work as expected with some bots still in the dataset. One possible reason for this error is that during the manual inspection and correction of author_type classifications, grammar errors or unintended whitespace characters that were overlooked, did not get addressed during the bot removal process and remained in the dataset. The remaining bots were manually removed and the steps after that were done again with the changed dataset. In other cases, developers with more than one unique login were given a single login for better identification of their true activity. No repository was excluded in this step.

4.11 Research Questions

This section discusses the research questions of this thesis.

- RQ1: How do key contributors spread across programming languages and ecosystems in cross ecosystem packages?

Data Used:

- Truck factor per programming language and ecosystem
- Commits per programming language

Motivation:

This question seeks to understand how key developers are distributed across different programming languages and ecosystems as well as how they interact with them. The goal is to determine whether developers tend to focus on one language or contribute to more, and provide insights into how contributors may influence multiple ecosystems. Some example situations being same group of developers supporting multiple languages causing a risk when the languages are abandoned, as well as distinct group of developers supporting one language each, risking uneven support between the languages.

- RQ2: Can social patterns give an explanation to the observed technical patterns?

Data Used:

- Technical Patterns
- Social Patterns

Motivation: The observed technical patterns might be the result of social situations or patterns. As an example, the language migration technical pattern could be the result of a conscious decision of one developer to support a different language or the result of a new core developer taking over the project and supporting a different language.

Section 5

Results

This section presents the outcome of the pattern detection process. Each pattern is explained in detail individually with repository examples for better understanding. The presentation order of the patterns is the hierarchy that was followed during the categorization of the repositories. The hierarchy prioritizes patterns from the most centralized to the more complex, ensuring that each repository is matched to its most representative pattern. For presentation purposes, only small repositories were chosen as examples.

5.1 Pattern 1: Single Developer

The first and most popular pattern that was found during the pattern extraction is “Single Developer”. This pattern represents the scenario where repositories have a single developer be responsible for the entire activity of the project.

The pattern is detected based on the following **criteria**:

- The project has exactly one unique contributor.
- That contributor is 100% active in all languages, across all tracked months.

The repository is expected to have only one developer active through all active months with no other developer appearing anywhere in the dataset. The developer must have 100% activity based on the Activity Percentage Calculation in all languages appearing in the project. If other developers appear they are expected to have 0% overall activity.

Example Cases:

Case 1 : FireEye Repository[3]

Year-Month	Javascript_Core	Python_Core	Javascript_TF	Python_TF
2019-08	0xjeremy	0xjeremy	0xjeremy	0xjeremy
2019-09	0xjeremy	0xjeremy	0xjeremy	0xjeremy
2020-04		0xjeremy		0xjeremy
2019-10				

Table 5.1.1: FireEye Repository Final Formatted Data

Language	Developer	Contribution (%)
Javascript	0xjeremy	100
Python	0xjeremy	100

Table 5.1.2: FireEye Repository Developer Contribution Percentage

Tables 5.1.1 and 5.1.2, represent the developer activity and contribution distribution in the FireEye repository over time. As shown in Table 5.1.1, 0xjeremy is the sole contributor across all languages as both Core and Truck Factor. Table 5.1.2 further confirms that 0xjeremy holds 100% of the activity in both Python and Javascript, reinforcing the observation of the single contributor.

This qualifies the repository under Pattern 1: “Single Developer” as it meets all the expected criteria of one unique developer that is 100% active in all languages.

Case 2: basic_test_widget Repository[4]

Year-Month	Javascript_Core	Python_Core	Javascript_TF	Python_TF
2018-05	aaronwatters	aaronwatters	aaronwatters	aaronwatters
2019-08		aaronwatters		aaronwatters
2018-06				

Table 5.1.3: basic_test_widget Repository Final Formatted Data

[3] https://github.com/0xJeremy/FireEye
[4] https://github.com/AaronWatters/basic_test_widget

Language	Developer	Contribution (%)
Javascript	aaronwatters	100
Python	aaronwatters	100

Table 5.1.4: basic_test_widget Repository Developer Contribution Percentage

Similar to the FireEye Repository, as seen in Tables 5.1.3 and 5.1.4, basic_test_widget Repository qualifies under Pattern 1 with aaronwatters being the sole contributor and owning the 100% of all activity in all languages.

Distribution of Social Pattern Across Technical Patterns:

This social pattern was detected in all 7 technical patterns with the most repositories matched being noted in Pattern 2 and Pattern 7. Table 5.1.5 shows the exact number of appearances in each pattern.

Technical Pattern	Social Pattern Appearances
Pattern 1 : Base language with support of other language	18
Pattern 2 : Parallel support in both languages	49
Pattern 3 : Interchanging support in both languages	2
Pattern 4 : Language Migration	6
Pattern 5 : Attempt Success	7
Pattern 6 : Attempt Failure	2
Pattern 7 : Unclear Pattern	89

Table 5.1.5: Social Pattern Distribution Across Technical Patterns

5.2 Pattern 2: One Developer per Language

The second pattern found is “One Developer per Language”. This pattern represents the scenario where each language is supported by only one exclusive contributor through the entirety of the project’s duration. No contributor works on any other language than the one they are responsible for. Based on the number of languages appearing in the repository, there is a corresponding number of unique contributors matching the amount exactly.

The pattern is detected based on the following **criteria**:

- There are at least two contributors.
- Each language is handled by exactly one developer, with 100% activity in that language.
- No developer appears in more than one language.
- No minor contributors

The repositories that are under this pattern are expected to have at least 2 contributors active, with each one being responsible for only one language only, meaning they have 100% activity distribution in a different language each. The selection of the number 2 comes from the fact that cross-ecosystem packages require at least 2 different ecosystems to be involved in a single repository to be considered as cross-ecosystem, leading to the conclusion that there are at least 2 languages in each repository and each has one developer supporting it. Based on that. the number of total unique contributors must be exactly the same as the number of languages used in the project. If the number differs, then the repository cannot be detected under pattern 2.

Example Cases:

Case 1 : epf Repository[5]

Year-Month	Javascript_Core	Ruby_Core	Javascript_TF	Ruby_TF
2013-05	ghempton		ghempton	
2013-06	ghempton		ghempton	
2013-07	ghempton	heartsentwined	ghempton	heartsentwined
2013-08	ghempton		ghempton	
2013-09	ghempton		ghempton	
2013-10	ghempton		ghempton	

Table 5.2.1: epf Repository Final Formatted Data

Language	Developer	Contribution (%)
Javascript	ghempton	100
Ruby	heartsentwined	100

Table 5.2.2: epf Repository Developer Contribution Percentage

Tables 5.2.1 and 5.2.2, represent the developer activity and contribution distribution in the epf repository over time. As shown in Table 5.2.1, ghempton and heartsentwined are the sole

[5] https://github.com/GroupTalent/epf

contributors in their respective languages as both Core and Truck Factor. Table 5.2.2 further confirms that ghempton and heartsentwined hold 100% of the activity in Javascript and Ruby respectively, reinforcing the observation of one single developer per language with no other overlapping developers.

Case 2 : hermes-protocol Repository[6]

Year-Month	Javascript_Core	Python_Core	Javascript_TF	Python_TF
2019-01	elbywan	anthonyray	elbywan	anthonyray
2019-02	elbywan	anthonyray	elbywan	anthonyray
2019-03	elbywan	anthonyray	elbywan	anthonyray
2019-04	elbywan	anthonyray	elbywan	anthonyray

Table 5.2.3: hermes-protocol Repository Final Formatted Data

Language	Developer	Contribution (%)
Javascript	elbywan	100
Python	anthonyray	100

Table 5.2.4: hermes-protocol Repository Developer Contribution Percentage

Similar to the epf Repository, as seen in Tables 5.2.3 and 5.2.4, hermes-protocol Repository qualifies under Pattern 2 with elbywan and anthonyray being the sole contributors in their respective languages with 100% activity each.

Distribution of Social Pattern Across Technical Patterns:

This social pattern was detected in only 3 technical patterns out of 7 with 1 matched repository each. Table 5.2.5 shows the exact number of appearances in each pattern.

Technical Pattern	Social Pattern Appearances
Pattern 1 : Base language with support of other language	1
Pattern 2 : Parallel support in both languages	1
Pattern 6 : Attempt Failure	1

Table 5.2.5: Social Pattern Distribution Across Technical Patterns

[6] https://github.com/snipsco/hermes-protocol

5.3 Pattern 3: Single Lead Developer with Passive Support

The third pattern found is “Single Lead Developer with Passive Support”. This pattern represents the scenario where one developer dominates the project with a high activity percentage across all languages, while the remaining contributors contribute minimally.

The pattern is detected based on the following **criteria**:

- One developer has ≥80% activity in each language.
- All other contributors are ≤30% active in every language.

The repositories matched with pattern 3 must have only one developer that dominates the project’s overall activity by more than 80% while the rest of the developers must have less than 30%. This way a clear distinction is made between lead and passive contributors. The 80% threshold was chosen because it reflects clear dominance without requiring 100% activity. The 30% threshold represents the supporting peripheral contributions. Any percentages above that are representing stronger contributors that do not align with the passive support requirement.

Example Cases:

Case 1 : resources-api-v1 Repository[7]

Year-Month	Javascript_Core	Python_Core	Javascript_TF	Python_TF
2017-04		flappysvgbuildbot, niccokunzmann		niccokunzmann
2017-05		niccokunzmann		niccokunzmann
2017-06		niccokunzmann		niccokunzmann
2017-08		niccokunzmann		niccokunzmann
2017-09	niccokunzmann	niccokunzmann	niccokunzmann	niccokunzmann
2017-07				

Table 5.3.1: resources-api-v1 Repository Final Formatted Data

Language	Developer	Contribution (%)
Javascript	niccokunzmann	100
Python	niccokunzmann	100
Python	flappysvgbuildbot	20

Table 5.3.2: resources-api-v1 Repository Developer Contribution Percentage

[7] https://github.com/schul-cloud/resources-api-v1

As seen in Table 5.3.1, developer niccokunzmann is consistently active through the entire project as both Core and Truck Factor in all languages. This high activity behaviour suggests that niccokunzmann is the lead developer. Table 5.3.2 further supports the suggestion with niccokunzmann being 100% in both languages. Developer flappysvgbuildbot on the other hand is a peripheral contributor with only 20% activity that falls under the 30% criteria.

Case 2 : select2-bootstrap-css Repository[8]

Year-Month	Javascript_Core	Ruby_Core	Javascript_TF	Ruby_TF
2013-05	michaek		michaek	
2013-07	michaek		michaek	
2013-08		michaek		michaek
2013-09	michaek	michaek	michaek	michaek
2014-06		michaek		michaek
2015-04	michaek	floriankissling, michaek	michaek	michaek

Table 5.3.3: select2-bootstrap-css Repository Final Formatted Data

Language	Developer	Contribution (%)
Javascript	michaek	100
Ruby	michaek	100
Ruby	floriankissling	25

Table 5.3.4: select2-bootstrap-css Repository Developer Contribution Percentage

Similarly to case 1, in Table 5.3.3 michaek is consistently active through all active months of the project in both languages as Core and Truck Factor while floriankissling is only active for one month as Core only. This observation shows that michaek is the lead developer and floriankissling is the passive support. Table 5.3.4 confirms the observation with michaek having 100% activity in both languages and floriankissling appearing only in Ruby with 25%.

Distribution of Social Pattern Across Technical Patterns:

This social pattern was detected in most of the technical patterns with 4 out of 7 having at least one matched repository. Pattern 7 has the highest number of matches with 13 matched repositories. Table 5.3.5 shows the exact number of appearances in each pattern.

[8] https://github.com/t0m/select2-bootstrap-css

Technical Pattern	Social Pattern Appearances
Pattern 1 : Base language with support of other language	5
Pattern 2 : Parallel support in both languages	3
Pattern 4 : Language Migration	1
Pattern 5 : Attempt Success	1
Pattern 7 : Unclear Pattern	13

Table 5.3.5: Social Pattern Distribution Across Technical Patterns

5.4 Pattern 4: All Developers Fully Active

The fourth pattern found is “All Developers Fully Active”. This pattern represents the scenario where all developers maintain the project with perfect activity percentage across multiple languages. The developers can be active in one or more languages with active months overlaps allowed.

The pattern is detected based on the following **criteria**:

- At least two developers exist.
- Every developer is 100% active in the languages they contribute to.
- At least one developer is 100% active in more than 1 languages.
- No minor contributors

For a repository to be detected under patterns 4 it must have at least 2 active developers with 100% activity in all the languages they appear. The 2 developers were chosen as a distinction between this pattern and pattern 1 where a single developer is fully active in all the languages. To ensure that this repository is not overlap with pattern 2 that requires at least 2 fully active developers as well, the criteria that requires at least one of the developers to be active in more than one language was added. This way the no overlap criteria of pattern 2 is not satisfied. Due to the nature of the pattern, no minor contributors are allowed since their existence automatically changes the pattern that suggests that all developers do not have activity percentages lower than 100%.

Example Cases:

Case 1 : gits Repository[9]

[9] https://github.com/tolstoyevsky/gits

Year-Month	Javascript_Core	Python_Core	Javascript_TF	Python_TF
2016-12	eugulixes, one001	dmitriyshilin, eugulixes	one001	eugulixes
2017-01		dmitriyshilin, eugulixes		eugulixes
2017-02	eugulixes, one001	dmitriyshilin, eugulixes	one001	eugulixes
2017-03		dmitriyshilin, eugulixes		eugulixes

Table 5.4.1: gits Repository Final Formatted Data

Language	Developer	Contribution (%)
Javascript	eugulixes	100
Javascript	one001	100
Python	dmitriyshilin	100
Python	eugulixes	100

Table 5.4.2: gits Repository Developer Contribution Percentage

Gits repository is a perfect example of pattern 4. All developers are consistently active in their respective languages, as seen in Table 5.4.1. one001 is fully active as Core or Truck Factor in Javascript, dmitriyshilin in Ruby and eugulixes in both. eugulixes by being active in both languages satisfies the criteria of at least one developer active in more than one language. Their perfect activity percentages are shown in Table 5.4.2. All of them have 100% activity percentage with no developer having a lower score, suggesting that no minor contributors are involved.

Case 2 : Poseidon Repository[10]

Year-Month	Java_Core	Javascript_Core	Java_TF	Javascript_TF
2017-12	southeryf, yodo1-yanfeng	yodo1-yanfeng	southeryf	yodo1-yanfeng
2018-01	southeryf, yodo1-yanfeng	yodo1-yanfeng	yodo1-yanfeng	yodo1-yanfeng
2018-02				

Table 5.4.3: Poseidon Repository Final Formatted Data

Language	Developer	Contribution (%)
Java	southeryf	100
Java	yodo1-yanfeng	100
Javascript	yodo1-yanfeng	100

Table 5.4.4: Poseidon Repository Developer Contribution Percentage

[10] https://github.com/Yodo1-backend/Poseidon

Just like gits repository, Poseidon repository is also a great example of pattern 4. In Table 5.4.3 exactly 2 developers are observed, both fully active in at least one language. southeryf is only active in Java while yodo1-yanfeng in both Java and Javascript. Table 5.4.4 further supports the match by confirming that both developers are fully active in their respective languages with 100% activity and yodo1-yanfeng is active in more than one language.

Distribution of Social Pattern Across Technical Patterns:

This social pattern was detected in only 3 technical patterns out of 7. Pattern 7 has the most detections with 5 matches repositories. Table 5.4.5 shows the exact number of appearances in each pattern.

Technical Pattern	Social Pattern Appearances
Pattern 1 : Base language with support of other language	1
Pattern 2 : Parallel support in both languages	3
Pattern 7 : Unclear Pattern	5

Table 5.4.5: Social Pattern Distribution Across Technical Patterns

5.5 Pattern 5: Distinct Lead Developer(s) per Language

The fifth pattern found is “Distinct Lead Developer(s) per Language”. This pattern represents the scenario where each language is dominated by at least one developer. All developers can be active but not dominate in more than one language.

The pattern is detected based on the following **criteria**:

- For every language, at least one developer has ≥70% activity.
- No developer is dominant in more than one language.
- Minor contributors can appear in multiple languages.

All repositories under pattern 5 must have at least one developer per language that dominates it by 70%. Due to the fact that multiple developers are expected to lead one language, 70% was the most suitable threshold that maintained balance between clear domination and realistic collaboration conditions. If a developer leads a language, they cannot lead a second one. They can appear as minor contributors in multiple languages but not more than one as lead.

Example Cases:

Case 1 : jupyterlab-commenting Repository[11]

Year-Month	Javascript_Core	Python_Core	Javascript_TF	Python_TF
2019-08		hoo761		hoo761
2019-09		hoo761		hoo761
2019-10	kgryte	hoo761, kgryte	kgryte	hoo761
2019-11		hoo761, kgryte		hoo761

Table 5.5.1: jupyterlab-commenting Repository Final Formatted Data

Language	Developer	Contribution (%)
Javascript	kgryte	100
Python	hoo761	100
Python	kgryte	50

Table 5.5.2: jupyterlab-commenting Repository Developer Contribution Percentage

Tables 5.5.1 and 5.5.2 act as proof that the jupyterlab-commenting repository is correctly detected as pattern 5. Table 5.5.1 shows kgryte and hoo761 appearing more consistently in Javascript and Python respectively compared to the rest of the contributions. Table 5.5.2 confirms these observations with the 2 dominating developers having 100% activity in their respective languages. Developer kgryte appears again in Python with 50% activity. This lower percentage means that the lead developers do not dominate more than one language even though kgryte appears in both languages, further confirming pattern 5.

Case 2 : wordfilter Repository[12]

Year-Month	Javascript_Core	Ruby_Core	Javascript_TF	Ruby_TF
2013-09	dariusk		dariusk	
2014-03	dariusk, techdubb		dariusk	
2014-09		elib		elib
2014-11	dariusk, puckey	elib, mambocab	dariusk	elib
2015-05	dariusk, jimkang, puckey		dariusk	
2015-10	dariusk, jimkang, puckey, techdubb		dariusk	

Table 5.5.3: wordfilter Repository Final Formatted Data

[11] https://github.com/jupyterlab/jupyterlab-commenting
[12] https://github.com/dariusk/wordfilter

Language	Developer	Contribution (%)
Javascript	dariusk	100
Javascript	puckey	60
Javascript	techdubb	40
Javascript	jimkang	40
Ruby	elib	100
Ruby	mambocab	50

Table 5.5.4: wordfilter Repository Developer Contribution Percentage

Similarly to case 1, Tables 5.5.3 and 5.5.4 prove that the wordfilter repository is detected as pattern 5. In this case, as seen in Table 5.5.3, dariusk and elib seem to dominate Javascript and Ruby respectively. Table 5.5.4 further proves this observation with both developers having perfect contribution scores in their languages. The rest of the developers do not have more than 70% activity in any of the languages. The 2 lead developers to not have overlapping activity in any of the languages meaning that the distinction between lead developers' criteria is satisfied.

None of the cases showed the scenario of one language having more than one lead developer that do not lead any other language due to their small size of the projects, but the scenario was detected in some bigger repositories like coalesce repository[13]. The repository's developer contributions are shown in Table 5.5.5 with the lead developers highlighted.

Language	Developer	Contribution (%)
Javascript	matthewdefazio	100
Javascript	dclemenzi	86.67
Javascript	tbone255	40
Python	dvenkat	94.12
Python	mtknife	82.35

Table 5.5.5: coalesce Repository Developer Contribution Percentage

Distribution of Social Pattern Across Technical Patterns:

This social pattern was detected in most of the technical patterns with 6 out of 7 technical patterns having at least one repository. Pattern 7 has the most detections with 14 repositories matched. Table 5.5.6 shows the exact number of appearances in each pattern.

[13] https://github.com/InCadence/coalesce

Technical Pattern	Social Pattern Appearances
Pattern 1 : Base language with support of other language	9
Pattern 2 : Parallel support in both languages	1
Pattern 3 : Interchanging support in both languages	2
Pattern 5 : Attempt Success	5
Pattern 6 : Attempt Failure	2
Pattern 7 : Unclear Pattern	14

Table 5.5.6: Social Pattern Distribution Across Technical Patterns

5.6 Pattern 6: Multiple Developers Leading the Project

The sixth pattern found is “Multiple Developers Leading the Project”. This pattern represents the scenario where a group of developers dominated the overall project with a high percentage in all implemented languages. This pattern can be seen as a more lenient version of pattern 5 by allowing language overlaps between the lead developers in comparison to pattern 5 that required strict distinction between the leading developers in each language. Only minor contributors here allowed to coexist in multiple languages.

The pattern is detected based on the following **criteria**:

- At least two lead developers.
- Each of the lead developers has ≥70% activity in every language.
- Minor contributors can appear

Repositories that get detected as pattern 6 must have at least 2 developers, for the same reasons that were stated in previous patterns, in the overall project and at least 2 of all contributors must be lead developers. To be a lead developer the contributor must have at least 70% activity in every language that exists in the project. The 70% threshold was chosen as it gives a more realistic result when it comes to multiple lead developers while still keeping a clear distinction between lead and other contributors. Developers with lower percentages can appear as minor contributors. All contributors can exist in any language with no restriction in the language commits overlaps.

Example Cases:

Case 1 : ipypivot Repository[14]

Year-Month	Javascript_Core	Python_Core	Javascript_TF	Python_TF
2017-12	ocoudray, pierremarion23	ocoudray, pierremarion23	pierremarion23	pierremarion23
2018-01	ocoudray, oscar6echo, pierremarion23	ocoudray, oscar6echo, pierremarion23	ocoudray	ocoudray
2018-02	ocoudray, oscar6echo, pierremarion23	ocoudray, oscar6echo, pierremarion23	ocoudray	ocoudray
2018-03	ocoudray, oscar6echo, pierremarion23		ocoudray	
2021-02		ocoudray, oscar6echo, pierremarion23		ocoudray

Table 5.6.1: ipypivot Repository Final Formatted Data

Language	Developer	Contribution (%)
Javascript	ocoudray	100
Javascript	pierremarion23	100
Javascript	oscar6echo	75
Python	ocoudray	100
Python	pierremarion23	100
Python	oscar6echo	75

Table 5.6.2: ipypivot Repository Developer Contribution Percentage

Table 5.6.1 shows a group of developers that appear consistently in both languages as either Core, Truck Factor or both. Developer ocoudray, pierremarion23 and oscar6echo are seen to dominate both languages with no support from any minor contributors in this case. Table 5.6.2 shows the high activity percentages of the three developers in both languages further confirming their status as lead developers. Developers ocoudray and pierremarion23 have 100% overall activity while oscar6echo has 75% overall activity. All three have over 70% activity and are lead developers in every language of the project successfully satisfying all criteria for pattern 6 detection.

[14] https://github.com/PierreMarion23/ipypivot

Case 2 : Centroid Repository[15]

Year-Month	Python_Core	Ruby_Core	Python_TF	Ruby_TF
2014-01	jtroe		jtroe	
2014-02	gregoryjscott, jtroe, thorncp	jtroe, thorncp	gregoryjscott	jtroe
2014-06	gregoryjscott, jtroe, thorncp	ralreegorganon, thorncp	gregoryjscott	jtroe
2014-07	gregoryjscott, jtroe, thorncp	jtroe, ralreegorganon, thorncp	gregoryjscott	jtroe
2014-08	gregoryjscott, jtroe, thorncp		gregoryjscott	
2016-10	gregoryjscott, jtroe, thorncp	jtroe, ralreegorganon, thorncp	gregoryjscott	jtroe

Table 5.6.3: Centroid Repository Final Formatted Data

Language	Developer	Contribution (%)
Python	jtroe	100
Python	gregoryjscott	83.33
Python	thorncp	83.33
Ruby	jtroe	100
Ruby	thorncp	100
Ruby	ralreegorganon	75

Table 5.6.4: Centroid Repository Developer Contribution Percentage

Just like case 1, Table 5.6.3 shows a group of developers dominating all languages. Unlike case 1 not all developers are considered lead in the project. Developers jtroe and thorncp are the only ones satisfying the lead criteria. Developers gregoryjscott and ralreegorganon do not satisfy it, not because of their activity percentage but from the fact that they do not appear in all languages. gregoryjscott contributes only Python and ralreegorganon only in Ruby. Table 5.6.4 clears up the previous observations by showing that all developers are over 70% but only 2 appear twice in the dataset.

[15] https://github.com/ResourceDataInc/Centroid

The above examples do not showcase the involvement of minor contributors in the project due to the small size of the repositories presented, but the scenario was indeed found in the remaining detected repositories.

Distribution of Social Pattern Across Technical Patterns:

This social pattern was detected in only 3 technical patterns out of 7. Pattern 7 has the most detections with 11 matches repositories. Table 5.6.5 shows the exact number of appearances in each pattern.

Technical Pattern	Social Pattern Appearances
Pattern 1 : Base language with support of other language	2
Pattern 2 : Parallel support in both languages	3
Pattern 7 : Unclear Pattern	11

Table 5.6.5: Social Pattern Distribution Across Technical Patterns

5.7 Pattern 7: Single Lead Developer with Strong Support

The seventh pattern found is “Single Lead Developer with Strong Support”. This pattern represents the scenario where one developer dominates the overall project with support from other strong contributors. This pattern can be seen as a more lenient version of pattern 3 by allowing strong developers with moderate percentages, in comparison to pattern 3 that required only minor developers with activity percentage that was under 30%.

The pattern is detected based on the following **criteria**:

- One developer has ≥85% activity in all languages.
- At least one other developer has ≥50% activity in at least one language.

For a repository to be detected as pattern 7 it needs to have at least 2 developers, one lead developer that has more than 85% activity in all languages and a strong contributor that has 50% and more in at least one language. These thresholds where specifically chosen to show the distinction between lead and strong contributors. Developers with 50% and more activity contribute significantly to a project but are not considered as leading the language or the projects. On the other hand, developers with more than 85% activity are extremely strong

contributors automatically considered as leading. In comparison to previous patterns that lead developers were measured with lower percentages, this scenario focuses on the strongest contributors of the project, so a stricter lead criterion was essential in the clear separation of the two groups.

Example Cases:

Case 1 : socker Repository[16]

Year-Month	Javascript_Core	Python_Core	Javascript_TF	Python_TF
2015-01	joar	joar, lundberg	joar	joar
2015-04		joar, lundberg		joar
2015-05		joar, lundberg		joar
2015-06		joar, lundberg		joar
2015-09	joar	joar, lundberg	joar	joar
2015-12		joar		joar
2016-12	joar	joar	joar	joar
2017-01		joar		joar

Table 5.7.1: socker Repository Final Formatted Data

Language	Developer	Contribution (%)
Javascript	joar	100
Python	joar	100
Python	lundberg	62.5

Table 5.7.2: socker Repository Developer Contribution Percentage

In Table 5.7.1 it is immediately noticed that joar is present in all active months. This automatically satisfies the lead developer criterion since it indicates that joar has 100% activity in both languages. What cannot be clear from Table 5.7.1 is whether lundberg is a strong contributor or a lead. Table 5.7.2 give the answer to that with lundberg having 62.5% activity. An activity percentage lower than 85% but higher than 50% classifies lundberg as a strong contributor in Python. Pattern 7 is successfully detected with one lead developer leading all languages and a strong contributor highly committing in at least one language.

[16] https://github.com/5monkeys/socker

Case 2 : jupyterlab-dash Repository[17]

Year-Month	Javascript_Core	Python_Core	Javascript_TF	Python_TF
2018-10		jonmmease		jonmmease
2018-11		jonmmease		jonmmease
2018-12		jonmmease, timkpaine		jonmmease
2019-02		anhvut, jonmmease		jonmmease
2019-03	jonmmease	jonmmease	jonmmease	jonmmease
2019-07	gnestor, jonmmease	anhvut, jonmmease	jonmmease	jonmmease

Table 5.7.3: jupyterlab-dash Repository Final Formatted Data

Language	Developer	Contribution (%)
Javascript	jonmmease	100
Javascript	gnestor	50
Python	jonmmease	100
Python	anhvut	33.33
Python	timkpaine	16.67

Table 5.7.4: jupyterlab-dash Repository Developer Contribution Percentage

In this case, compared to case 1, there are minor contributors as well. Just like socker repository, in Table 5.7.3 it is immediately noticed that jonmmease is always active. The rest of the developers are unable to have their activity percentage correctly estimated from this format, so Table 5.7.4 is used instead. In Table 5.7.4 gnestor has 50% activity and is considered a strong contributor while the remaining developers have less percentages and are considered minor. These developers do not have an important role in the pattern detection unless no strong contributor is present and the representative pattern of the repository changes.

Distribution of Social Pattern Across Technical Patterns:

This social pattern was detected in most of the technical patterns with 6 out of 7 technical patterns having at least one repository. Patterns 1 and 7 have the most detections with 21 and 38 repositories matched. Table 5.7.5 shows the exact number of appearances in each pattern.

[17] https://github.com/plotly/jupyterlab-dash

Technical Pattern	Social Pattern Appearances
Pattern 1 : Base language with support of other language	21
Pattern 2 : Parallel support in both languages	9
Pattern 3 : Interchanging support in both languages	2
Pattern 4 : Language Migration	4
Pattern 5 : Attempt Success	1
Pattern 7 : Unclear Pattern	38

Table 5.7.5: Social Pattern Distribution Across Technical Patterns

5.8 Pattern 8: Pattern Unclear

The last pattern in the hierarchy is “Pattern Unclear”. This pattern represents all repositories that did not match in any of the patterns found and were left with no categorization. There are two types of repositories left in this pattern: the repositories that could match with one of the patterns but didn’t because they had one of the criteria missing, and the repositories that showed unstable behaviours that could not be successfully converted to a pattern.

Types of Unclear Repositories:

Type 1: Could have matched with a pattern but did not due to missing criteria

Case: serverless-rpc Repository [18]

Year-Month	Javascript_Core	Python_Core	Javascript_TF	Python_TF
2019-08	tvaintrob	galbash	tvaintrob	galbash
2019-09	galbash, tvaintrob	galbash	galbash	galbash
2019-10	galbash	galbash	galbash	galbash
2019-11	galbash	galbash	galbash	galbash

Table 5.8.1: serverless-rpc Repository Final Formatted Data

Language	Developer	Contribution (%)
Javascript	galbash	75
Javascript	tvaintrob	50
Python	galbash	100

Table 5.8.2: serverless-rpc Repository Developer Contribution Percentage

[18] https://github.com/galbash/serverless-rpc

Tables 5.8.1 and 5.8.2 represent the developer activity and contribution distribution in the serverless-rpc repository. In Table 5.8.1, it is observed that galbash is consistent in both languages while tvaintrob appears only in Javascript. Table 5.8.2 confirms the observations with galbash leading both languages with 75% activity and more and tvaintrob having 50% activity only in Javascript. Pattern 7 seems to be the closest match, but due to the 75% activity of galbash in Javascript it was not detected. Pattern 7 has an 85% requirement in all languages.

Type 2: Unclear / Unique behaviour that could not be converted to a pattern

Case: myanmar-tools Repository[19]

Year-Month	Java_Core	Javascript_Core	Ruby_Core	Java_TF	Javascript_TF	Ruby_TF
2017-11	sffc	sffc		sffc	sffc	
2018-03			kirankarki			kirankarki
2018-05	sffc	sffc		sffc	sffc	
2018-06	sffc, sven-oly	sffc, sven-oly		sffc	sven-oly	
2018-07	sffc, sven-oly	sffc, sven-oly		sffc	sven-oly	
2018-08	sffc, sven-oly	sffc, sven-oly		sffc	sven-oly	
2018-09	sffc, sven-oly	sffc, sven-oly		sffc	sven-oly	
2018-11	sffc, sven-oly	sffc, sven-oly		sffc	sven-oly	
2019-10	sffc, sven-oly			sffc		
2019-11	sffc, sven-oly	sffc, sven-oly		sffc	sven-oly	
2020-02	sffc, sven-oly	sffc, sven-oly	kirankarki	sffc	sven-oly	kirankarki
2020-07	sffc, sven-oly		kirankarki, sffc	sffc		kirankarki
2020-09	sffc, sven-oly			sffc		
2022-10			kirankarki, sffc			kirankarki

Table 5.8.3: myanmar-tools Repository Final Formatted Data

[19] https://github.com/googlei18n/myanmar-tools

Language	Developer	Contribution (%)
Java	sffc	100
Java	sven-oly	83.33
Javascript	sffc	100
Javascript	sven-oly	77.78
Ruby	kirankarki	100
Ruby	sffc	50

Table 5.8.4: myanmar-tools Repository Developer Contribution Percentage

In Table 5.8.3 a pattern similar to pattern 7 is observed with developer sffc appearing in every language with a high frequency. Other developers appear as frequently leading to the conclusion that there is strong support. What strays away from pattern 7 is the different lead developer in Ruby. kirankarki seems to be the lead developer in that language leaving sffc as support. Table 5.8.4 backs up these observations with sffc leading Java and Javascript and kirankarki leading Ruby. This behaviour is unique only to repositories with 3 ecosystems involved. Pattern 7 could not be modified to satisfy this scenario since it would come in contrast with the original purpose of the pattern and a new pattern could not be created since it is a very specific case.

Distribution of Social Pattern Across Technical Patterns:

This social pattern was detected in most of the technical patterns with 5 out of 7 technical patterns having at least one repository. Pattern 7 has the most detections with 32 repositories matched. Table 5.8.6 shows the exact number of appearances in each pattern.

Technical Pattern	Social Pattern Appearances
Pattern 1 : Base language with support of other language	10
Pattern 2 : Parallel support in both languages	5
Pattern 3 : Interchanging support in both languages	1
Pattern 5 : Attempt Success	1
Pattern 7 : Unclear Pattern	32

Table 5.8.6: Social Pattern Distribution Across Technical Patterns

5.9 Answering Research Questions

RQ1: How do key contributors spread across programming languages and ecosystems in cross ecosystem packages?

Based on the observed social patterns, developer involvement across languages and ecosystems can be described by 2 distinct distribution models.

The first distribution model is when one or multiple developers choose to work on all languages of the repository, and in result contribute to all involved ecosystems. Cases like these are mostly found in Pattern 1: Single Developer, Pattern 3: Single Lead Developer with Passive Support, Pattern 4: All Developers Fully Active, Pattern 6: Multiple Developers Leading the Project and Pattern 7: Single Lead Developer with Strong Support but it can generally be seen in all repositories. Pattern 1 is a perfect example of how one developer takes the decision to maintain all languages and ecosystems on their own. For the rest of the patterns, the model is most apparent in the lead developers that are required to maintain all languages for the pattern to be detected.

The second distribution model is when one or more developers choose to work only on a selection of languages. Cases like these are most common in Patterns 2: One Developer per Language and Pattern 5: Distinct Lead Developer(s) per Language but can be found in any pattern. Both patterns have one thing in common, one of their criteria requires the developers to work only on a subset of the languages. Pattern 2 explains the model perfectly by requiring every developer to work only on one language each. When a developer works as a lead in 1 language out of N, they follow this distribution model.

These models are the two general models observed when analyzing cross-ecosystem packages. More specific models, for example developer works in exactly half languages and ecosystems or developer focuses more on one language than the other they work on, can be found in the repositories but they are not distinct.

RQ2: Can social patterns give an explanation to the observed technical patterns?

Yes, social patterns can help explain and understand the observed technical patterns by analysing the behaviour of the contributors. The social patterns give an insight on how developers worked and behaved during the development process.

This insight can give possible reasons for a transition that changed the course of the project. An example is Technical Pattern 4: Language Migration where the project leaves one language behind and starts to use another. A possible explanation to this shift is if the repository follows the social pattern 1 then the sole developer took the decision to change the language because they wanted, or it was needed to integrate that shift.

A more detailed analysis on how each social pattern can explain each technical pattern is found in Section 6 that follows.

Section 6

Discussion

6.1 The Role of the Dataset in Social Pattern Discovery

The dataset chosen plays a crucial role in the social patterns that will be discovered. Each dataset has its own unique characteristics setting them apart from the one that was used during this research. It is important to understand that the social patterns found were based on the selected repositories. In different datasets, possible similarities may occur, but differences can be found depending on the dataset used for such an analysis.

The five ecosystems that were selected to be used during the analysis had distinct development patterns. If at least one of the five ecosystems was replaced with a new one, the social patterns found could be different as well. Another important variable that was based on the selected ecosystems was the thresholds used. The thresholds were chosen empirically based on the analysis of the provided repositories. Different cases could lead to different threshold selection, possibly changing the patterns' criteria.

All things considered, the detected social patterns, even though they represent only a small subset of all possible patterns that can be found in cross-ecosystem packages, provide useful insights on how developers interact and behave in cross-ecosystem packages.

6.2 Explaining Technical Patterns through Social Patterns

	Social 1	Social 2	Social 3	Social 4	Social 5	Social 6	Social 7	Social 8
Technical 1	18	1	5	1	9	2	21	10
Technical 2	49	1	3	3	1	3	9	5
Technical 3	2	-	-	-	2	-	2	1
Technical 4	6	-	1	-	-	-	4	-
Technical 5	7	-	1	-	5	-	1	1
Technical 6	2	1	-	-	2	-	-	-
Technical 7	89	-	13	5	14	11	38	32

Table 6.1: Social Pattern Matched Repositories Distribution across Technical Patterns

Table 6.1 presents the distribution of matched repositories across both social and technical patterns. Each cell indicates how many repositories were categorized under a given combination of social and technical patterns. This table allows a better understanding of the relationship between technical and social aspects and better explain that relationship in the following sections.

Technical Pattern 1: Base language with support of other language

This pattern reflects projects with one language more dominant than the other. The rest of the languages show light support compared to the base language. The social patterns found include Patterns 1-8.

This pattern is found in all observed social patterns even though each one of them has a unique structure. For each case the scenario that enables this link might be different but there are similarities that act as a common ground for the technical pattern.

Social patterns 1 (Single Developer) and 3 (Single Lead Developer with Passive Support) have as common characteristic one developer leading the project. The base language is closely tied with the lead developer and their activity in each language. The lead developer focuses on the base language while the other supporting languages receive less attention.

On the other hand, social patterns 2 (One Developer per Language) and 5 (Distinct Lead Developer(s) per Language) focus on each lead developer working on a different language. This leads to the scenario of one developer being active more frequently in comparison to the other, resulting in one language becoming the base.

Social patterns 4 (All Developers Fully Active), 6 (Multiple Developers Leading the Project), and 7 (Single Lead Developer with Strong Support) show a more collaborative approach. Multiple developers being in charge of each language, all with high activity percentages. This scenario is caused by uneven developer or commit distribution across languages. Strong and lead developers focus on one language and occasionally support others.

Technical Pattern 1 (Base language with support of other language) is mainly associated with **Social Pattern 7** (Single Lead Developer with Strong Support) (21 repos). This pattern is observed when the lead developers prioritize one primary language while, with or without the help of other strong contributors, they occasionally maintain the remaining languages.

Technical Pattern 2: Parallel support in both languages

This pattern reflects projects that have similar activity over time in both languages. There is no dominating language. The social patterns found include Patterns 1-8.

Just like Technical Pattern 1 (Base language with support of other language), Technical Pattern 2 (Parallel support in both languages) is found in all social patterns. The similarities that are shared between the social patterns are different in this case with multiple languages being supported at the same rate.

Social patterns 1 (Single Developer) and 3 (Single Lead Developer with Passive Support) have one lead developer leading the project. In this case, the lead developer works equally in both languages at the same time. This leads to an increase in workload for them and puts the languages at risk of abandonment.

Social patterns 2 (One Developer per Language) and 5 (Distinct Lead Developer(s) per Language) on the other hand, have 2 lead developers working in parallel using the same effort in committing. Their activity is the same and so is their commit period.

Social patterns 4 (All Developers Fully Active), 6 (Multiple Developers Leading the Project), and 7 (Single Lead Developer with Strong Support) follow 2 different scenarios. The first scenario is that all strong and lead developers work on both languages at the same time. The second scenario is that the developers are divided into teams that are synchronized.

Technical Pattern 2 (Parallel support in both languages) is mainly associated with **Social Pattern 1** (Single Developer) (49 repos). This pattern is observed when one lead developer, with or without contribution from other strong contributors, supports both languages of the system at the same time.

Technical Pattern 3: Interchanging support in both languages

This pattern reflects projects that do not have a dominating language, but each language has higher commit frequency on different time periods than the other. There is a "switch" in the attention given to each language. The social patterns found include Patterns 1, 5, 7 and 8.

Social pattern 1 (Single Developer) has one developer maintaining both languages. The interchanging behavior reflects on how a developer divides their time and effort between languages while trying to keep a balanced support on both.

Social pattern 5 (Distinct Lead Developer(s) per Language) has different developers working in different languages. The developers in charge of each language work asynchronously resulting in the observed interchanging behavior. This can also mean that different parts of the project evolve independently and in different periods.

However, in social pattern 7 (Single Lead Developer with Strong Support) there is a lead developer that works in both languages and has strong support from others that work in at least one of them. In this case, the languages are being maintained by multiple developers with high activities. The interchange comes from different groups of developers working on different parts of the project. It is also caused by the same group deciding to interchange their focus on different parts of the project each time.

Technical Pattern 3 (Interchanging support in both languages) is mainly associated with **Social Pattern 1** (Single Developer), **5** (Distinct Lead Developer(s) per Language), and **7** (Single Lead Developer with Strong Support) (each with 2 repos). This pattern is observed when one or more developers divide their attention to a language based on active time periods or different groups of developers work on separate languages on different time periods.

Technical Pattern 4: Language Migration

This pattern reflects projects that have a successful transition from one main language to another. There is a period where they overlap. The social patterns found include Patterns 1, 3 and 7.

This pattern has a smaller set of social patterns found but all of them have a lead developer being their common characteristic. This characteristic is very important in the explanation of the pattern.

Social patterns 1 (Single Developer) and 3 (Single Lead Developer with Passive Support) are the most similar with both of them having clear leadership over the project, unlike social pattern 7 that shares the lead with other strong contributors. In the first 2 social patterns the decision to change from one language to another can be made by one individual. The lead developer is the one in charge of that migration since their effort is the key.

In comparison, in social pattern 7 (Single Lead Developer with Strong Support) the migration is led by the lead developer, but it is supported by the strong contributors. The language migration is a team effort and not a solo work. If the lead developer tries to migrate from one language to another without the other strong developers' support, the migration fails.

Technical Pattern 4 (Language Migration) is mainly associated with **Social Pattern 1** (Single Developer) (6 repos). This pattern is observed when a lead developer performs the language migration alone or with the support of other contributors.

Technical Pattern 5: Attempt Success

This pattern reflects projects that have a successful integration of a new programming language into the project's development workflow. The social patterns found include Patterns 1, 3, 5, 7 and 8.

Social patterns 1 (Single Developer) and 3 (Single Lead Developer with Passive Support) have a clear lead developer, that is the reason for this successful integration. The lead developer takes the initiative and the responsibility to successfully bring in the project a new language.

However, in social pattern 5 (Distinct Lead Developer(s) per Language) there is a clear distinction between the lead developers. This means that in this case a new lead developer, different from the already established ones, starts to contribute to the repository in a new language. This lead developer is responsible for the integration of this new language.

For social pattern 7 (Single Lead Developer with Strong Support), we have a similar scenario found in technical pattern 4 (language migration). The scenario has the lead developer initializing the integration and at least one other contributor actively supporting it.

Technical Pattern 5 (Attempt Success) is mainly associated with **Social Pattern 1** (Single Developer) (7 repos). This pattern is observed when an existing or new lead developer implements and supports a new language actively for the remaining of the project alone or with support from other contributors.

Technical Pattern 6: Attempt Failure

This pattern reflects projects that fail to integrate a new programming language into the project's development workflow resulting to its abandonment. The social patterns found include Patterns 1, 2 and 5.

This pattern is found in a very small subset of the social patterns indicating its specific requirements. The social patterns have common attributes that can explain this pattern.

Social pattern 1 (Single Developer) represents a developer that tried to integrate a new language in their project but failed and abandoned it before the end of the project. This failure could be a result of multiple factors as the developer did not have the time or the skills to integrate into the project the new language.

In contrast, social patterns 2 (One Developer per Language) and 5 (Distinct Lead Developer(s) per Language) have new lead developers trying to integrate a new language but failing in the process and abandoning the language in social pattern 5 (Distinct Lead Developer(s) per Language) or the project all together in both. The new developer might have failed to sync themselves with the rest of the contributors or possibly did not have the time to continue maintaining the language and dropped the project.

Technical Pattern 6 (Attempt Failure) is mainly associated with **Social Pattern 1** (Single Developer) and **5** (Distinct Lead Developer(s) per Language) (2 repos each). This pattern is observed when an existing or new lead developer tries to support a new language but abandons it before the end of the project.

Technical Pattern 7: Unclear Pattern

This pattern reflects projects with unclear language patterns. They show lack of notable trends or consistent usage dynamics, making it impossible to sort them into a particular category. The social patterns found include Patterns 1 and 3-8.

This pattern represents all repositories left without a pattern during the repository categorization process. It is expected for the pattern to have multiple different social patterns. It is not possible to explain through the observed social patterns exactly why these repositories were left with no technical pattern.

Possible reasons can be either technical or social. New requirements emerging constantly forcing the developers to not follow a consistent pattern or lack of communication and incoordination between the involved developers can lead to an unpredicted development process.

The reason that social pattern 2 (One Developer per Language) does not include this pattern is solely because of the small number of repositories found following this social pattern.

Technical Pattern 7 (Unclear Pattern) is mainly associated with **Social Pattern 1** (Single Developer) (89 repos). This pattern is observed when no distinct collaboration styles are observed with no strong lead developer patterns.

6.3 Distribution of Repositories by Social Pattern

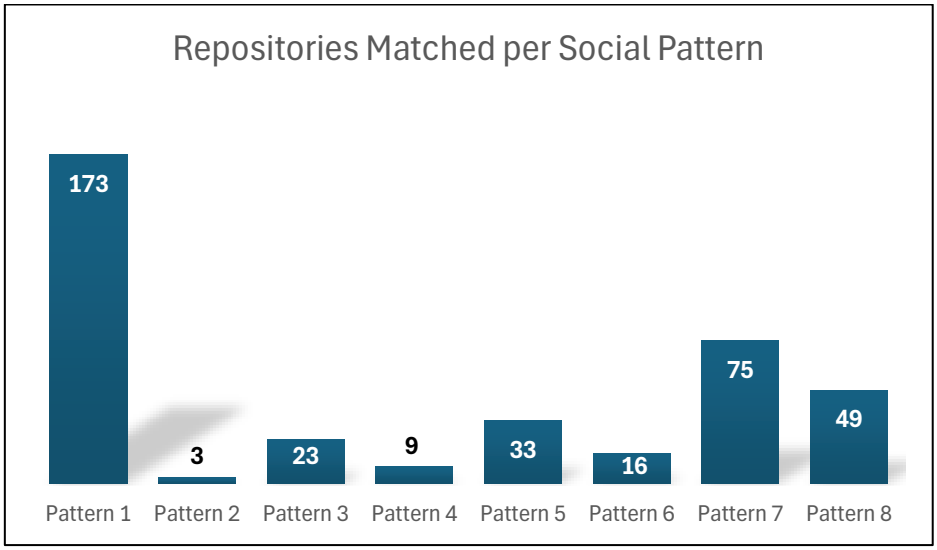

Figure 6.2: Number of Repositories Detected per Social Pattern

The results of the repository categorization process show a clear imbalance on how collaboration patterns are distributed across the analysed repositories. As seen in Figure 6.2, each pattern has a different number of repositories matched.

Social pattern 1 (Single Developer) has the highest distribution with 173 repositories. This pattern represents highly centralized and solo-driven projects, and it is very common in open-source repositories. Avelino et al. in his research on the survival and the abandonment of open-source projects [23], calculated how many truck factor developers each project had. The calculation returned that more than half of the repositories that were analysed had only 1 truck factor.

The second most common pattern is pattern 7 (Single Lead Developer with Strong Support). It suggests that while one developer has most of the activity and leads the project, multiple other developers also contribute significantly to it. It presents a more realistic scenario found in open-source repositories on how contributors collaborate with each other to maintain and evolve the project.

On the other hand, social patterns 2 (One Developer per Language) and 4 (All Developers Fully Active) have the least repositories. This indicates that perfectly balanced projects are not very common in open-source projects. Open-source projects include multiple developers, usually developers with no connection between them, that each collaborates on their own rate. This leads to more collaborative patterns with wide range of activity percentages.

Section 7

Conclusions and Future Work

7.1 Conclusions

This research investigates the social aspects of cross-ecosystem packages and more specifically the way that developers behave and collaborate with each other across multiple languages and ecosystems. Through the analysis of 381 repositories that expand across five major ecosystems, 8 distinct social patterns were uncovered. These patterns range from solo-driven projects to multi-membered team collaborations, providing valuable information on the human side of evolving cross-ecosystem packages.

The results show that these projects do not follow a common structure, with certain ecosystems receiving more attention than the others. This asymmetry is possibly linked to the distribution of the developers across the project and their activity percentages. Developers with higher percentages have more influence in the project evolution compared to minor contributors.

Finally, social patterns have a strong correlation with Orphanos's [5] observed technical patterns. The social patterns can successfully explain why certain technical behaviors were observed and what was the possible reason for their appearance.

7.2 Future Work

In the future, while this research offers valuable insights into the social dynamics of cross-ecosystems, there are multiple opportunities to expand the topic further based on these findings. Firstly, expanding the selection of ecosystems to include more software ecosystems could

provide a better understanding of the social dynamics and return richer results that represent more ecosystems.

Additionally, more social patterns can be detected. Multiple repositories did not have a pattern that represented them, and further research could be a way to eliminate pattern 8: Pattern Unclear. Also, patterns can be expanded to investigate how developer roles change over time. A possible pattern could analyse how key contributors abandon the project and new developers continue the work keeping the project alive. These patterns could explain certain technical patterns with more accuracy because they contain the element of time.

Bibliography

[1] M. Lungu, "Towards reverse engineering software ecosystems," 2008 IEEE International Conference on Software Maintenance, pp. 428-431, 2008.

[2] E. Constantinou, A. Decan, and T. Mens, “Breaking the borders: an investigation of cross-ecosystem software packages”, 17th Belgium-Netherlands Software Evolution Workshop, 2018.

[3] K. Kannee, G. Raula, S. Kula, K. Wattanakriengkrai, and Matsumoto, “Intertwining Communities: Exploring Libraries that Cross Software Ecosystems”, IEEE/ACM 20th International Conference on Mining Software Repositories, pp. 518-522, 2023.

[4] R. Paramitha, Y. Feng, F. Massacci, and C. E. Budde, "Cross-ecosystem categorization: A manual-curation protocol for the categorization of Java Maven libraries along Python PyPI Topics", 2024

[5] C. Orphanos, “Evolution of Cross-Ecosystem Packages.”, Undergraduate Thesis, University of Cyprus, 2024.

[6] P. Malcher, O. Barbosa, D. Viana, and R. Santos, “Software Ecosystems: A Tertiary Study and a Thematic Model,” 2022.

[7] K. Manikas and K. M. Hansen, “Software ecosystems – A systematic literature review”, Vol. 86, No. 5, pp. 1294, 2012.

[8] D. Lettner, F. Angerer, H. Prähofer, and P. Grünbacher, “A case study on software ecosystem characteristics in industrial automation software,” ACM, pp. 40, 2014.

[9] J. V. Joshua, D. O. Alao, S. O. Okolie, and O. Awodele, "Software Ecosystem: Features, Benefits and Challenges," International Journal of Advanced Computer Science and Applications, Vol. 4, No. 8, pp. 242–247, 2013

[10] “Stack Overflow Annual Developer Survey 2024,” survey.stackoverflow.co. https://survey.stackoverflow.co/2024/technology#most-popular-technologies (accessed May 17, 2025).

[11] “Stack Overflow Annual Survey,” survey.stackoverflow.co. https://survey.stackoverflow.co/ (accessed May 17, 2025).

[12] “Stack Overflow Developer Survey 2024: Most Popular Programming Languages,” survey.stackoverflow.co. https://survey.stackoverflow.co/2024/technology#most-popular-technologies-language (accessed May 17, 2025).

[13] “Libraries.io,” libraries.io. https://libraries.io/ (accessed May 17, 2025).

[14] “Apache Maven,” maven.apache.org. https://maven.apache.org/index.html (accessed May 17, 2025).

[15] “PyPi,” pypi.org. https://pypi.org/ (accessed May 17, 2025).

[16] “NPM,” npmjs.com. https://www.npmjs.com/ (accessed May 17, 2025).

[17] “Rubygems,” rubygems.org. https://rubygems.org/ (accessed May 17, 2025).

[18] “CRAN,” cran.r-project.org. https://cran.r-project.org/ (accessed May 17, 2025).

[19] D. G. Messerschmitt and C. Szyperski, *Software Ecosystem: Understanding an Indispensable Technology and Industry*. MA: MIT Press, 2003.

[20] “PyDriller Library,” pydriller.readthedocs.io. https://pydriller.readthedocs.io/en/latest/index.html (accessed May 17, 2025).

[21] C. Bird, A. Gourley, P. Devanbu, M. Gertz, and A. Swaminathan, “Mining email social networks,” in Proceedings of the 2006 international workshop on Mining software repositories, USA: Association for Computing Machinery, pp. 137–143, 2006.

[22] E. Kouters, B. Vasilescu, A. Serebrenik, and M. G. J. van den Brand, “Who’s who in Gnome: Using LSA to merge software repository identities,” in 2012 28th IEEE International Conference on Software Maintenance, pp. 592–595, 2012

[23] M. Goeminne and T. Mens, “A comparison of identity merge algorithms for software repositories” Science of Computer Programming, Vol. 78, No. 8, pp. 971–986, 2013.

[24] “BoDeGiC Repository,” github.com. https://github.com/mehdigolzadeh/BoDeGiC (accessed May 17, 2025).

[25] M. Golzadeh, A. Decan, and T. Mens, “Evaluating a bot detection model on git commit messages,”, 2021.

[26] G. Avelino, E. Constantinou, M. T. Valente and A. Serebrenik, "On the abandonment and survival of open source projects: An empirical investigation," 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, pp. 1-12, 2019.

[27] G. Avelino, L. Passos, A. Hora, and M. T. Valente, “A novel approach for estimating Truck Factors,” in Proceedings of the 24th IEEE International Conference on Program Comprehension, pp. 1–10, 2016.

[28] “Truck Factor Repository,” github.com. https://github.com/HelgeCPH/truckfactor (accessed May 17, 2025).

[29] “truckfactor PyPi Package,” pypi.org. https://pypi.org/project/truckfactor/ (accessed May 17, 2025).

[30] G. Robles and J. M. González-Barahona, “Contributor Turnover in Libre Software Projects,” in Open Source Systems, E. Damiani, B. Fitzgerald, W. Scacchi, M. Scotto, and G. Succi, Eds., Springer, pp. 273–286, 2006.