Thesis Dissertation

EVOLUTION OF CROSS-ECOSYSTEM PACKAGES

Constantinos Orphanos

University Of Cyprus

COMPUTER SCIENCE DEPARTMENT

May 2024

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

Evolution of cross-ecosystem packages

Constantinos Orphanos

Supervisor

Eleni Constantinou

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

May 2024

Acknowledgments

I would like to express my sincere appreciation to my supervisor, Professor Eleni Constantinou, for their guidance and introduction to the topic of my thesis.

Abstract

This thesis delves into the analysis of programming language usage patterns across cross-ecosystem packages, aiming to uncover trends and understand the dynamics in those packages.

The methodology employed involves a multi-step approach, beginning with data collection using PyDriller, followed by data cleaning and analysis, including the calculation of metrics such as Discrepancy Percentage and Language Percentage. Additionally, specialized processes were developed to identify potential language shifts within repositories, further enhancing the depth of analysis.

The results reveal seven distinct patterns observed in the data: from base languages with light support of another language, to instances of language migration and attempts at introducing new languages. Each pattern is analyzed, providing insights into the duration, commit activity, and modified files associated with each pattern.

I highlight the utility of the patterns in software development analysis, and their potential to inform decision-making processes and guide language selection strategies. Moreover, I elaborate on the usefulness of these patterns for both researchers and developers.

In conclusion, this thesis contributes to the growing body of knowledge surrounding software development practices in software ecosystems by uncovering and analyzing patterns in programming language usage across cross-ecosystem packages. The insights gained from this study have the potential to inform and empower developers, researchers, and industry practitioners.

Contents

Section 1

Introduction

In the realm of software development, the evolution and interplay of various software ecosystems represent a complex and dynamically evolving landscape. Software ecosystems are basically groups of projects that evolve and develop together in the same environment [11]. As the boundaries of technology continue to expand, the interaction between different ecosystems has become a focal point for understanding the broader implications of software development practices. Software ecosystems are very useful. One of the main reasons is that they decrease the cost involved in software development and distribution. In addition, SECOs (Software Ecosystems) support cooperation and knowledge sharing among the software developers [1].

One of the defining characteristics of software ecosystems is the dependencies among the packages of a SECO. These dependencies are not always technical linkages, but they are crucial forces that significantly influence the evolution of a SECO. Understanding the true dynamics of dependencies is very important to understanding how SECOs grow and adapt over time. To make it clearer these dependencies are basically bonds between two different packages, often because one package requires the functionality provided by the other. These “bonds” means that changes in a single package can affect other dependent packages in various ways. One of these ways is if a package introduces new features or improvements, dependent packages can leverage on these advancements leading to broader ecosystem innovation and evolution. Updates or changes in one package also may motivate modifications in dependent packages to maintain compatibility. Similarly, if a package is found to have a vulnerability or a bug, then the dependent packages are also in danger. So, we can summarize that the network of dependencies within a software ecosystem acts as a catalyst for evolution, pushing the packages affected to adapt, innovate, and improve.

Building upon the investigation of package dependencies within individual ecosystems, the subsequent and most critical concept my dissertation introduces is that of cross-

ecosystem packages. Cross-ecosystem packages are basically packages that are distributed and utilized across different programming languages and platforms. These packages are designed to operate or be shared across multiple software ecosystems, such as Maven, PyPI, RubyGems, CRAN and NPM in my case. An example of such a package is the 'Singularity' repository found on GitHub, which is found on PyPi, Maven and NPM. The key characteristic of the cross-ecosystem packages is that they can function seamlessly across different SECOs. One of the reasons that makes them so important is that they tend to be more popular, and they have many dependent packages across multiple platforms [2]. Another reason is that a bug or a vulnerability on a cross-ecosystem package may span across different ecosystems and if fixing this bug is not done for all the ecosystems involved, then the users of a specific ecosystem might still encounter problems while trying to use a certain package with a bug.

Existing research on cross ecosystem packages by two pivotal studies, comes to shed light on the presence, characteristics, and implications of these packages within and across software ecosystems. The first study conducted by Constantinou et al [2] aimed to uncover the characteristics and evolution of cross-ecosystem packages. Through their methodology and research, they identified a small fraction of packages that are distributed across multiple ecosystems. They discovered that these packages tend to favor certain ecosystems over others in terms of support with new releases, which is also a motivation for my thesis. Moreover, their analysis revealed that they have a significant impact on the dependency networks within each ecosystem, and therefore affecting many packages across different ecosystems. As previously highlighted, and confirmed by the findings in their research, these packages are more popular and have larger developer communities. All these factors indicate a higher level of importance, underscoring the need for further investigation into them. Complementing this, the study by Kula et al [12] embarks on a large-scale empirical analysis of 1.1 million libraries from five distinct software ecosystems: PyPI, CRAN, Maven, RubyGems, and NPM. Their work focuses on libraries that are released across multiple ecosystems, thereby intertwining these ecosystems. By identifying 4,146 GitHub repositories hosting such cross-ecosystem libraries and analyzing their dependency and contributor patterns, their findings highlight the deep interconnectedness between different software ecosystems. They reveal that cross-ecosystem libraries are significantly dependent upon by the ecosystems they belong to

and attract a considerable portion of contributors from within those ecosystems, emphasizing the growing interconnectedness and community reach beyond the confines of a single programming language.

Building on the groundwork laid by previous research into the characteristics and impact of cross-ecosystem packages, a significant gap remains in our understanding of how these packages evolve at the source code level across different package managers. While Constantinou et al [2] study offers valuable insights into the distribution, popularity, and dependency networks of cross-ecosystem packages, it stops short of examining the programming language usage from each package.

This thesis explores and discovers the patterns of programming language usage within cross-ecosystem packages, aiming to clarify how each ecosystem evolves and is supported throughout time. The contributions of this work are multiple: it provides an understanding of programming language patterns in cross-ecosystem packages and ultimately provides developers and researchers with insights to encourage better decisions and further research on cross-ecosystem packages. In this way, the thesis enriches existing knowledge on software ecosystems and provides a new perspective on the intricacies of cross-ecosystem software development.

Section 2

Background Knowledge

2.1 Definition of a software ecosystem

In the context of software analysis, a software ecosystem is defined by Lungu [3] as: “Collection of [interdependent] software projects that are developed and evolve together in the same environment”. Another definition by Jansen is: “A set of actors functioning as a unit and interacting with a shared market for software and services, together with the relationships among them. These relationships are frequently underpinned by a common technological platform or market and operate through the exchange of information, resources, and artifacts”.

The term "ecosystem" draws an analogy from biological ecosystems, reflecting the complex interdependencies, competition, and collaboration that occur within a shared environment. In a software ecosystem, these dynamics manifest as developers contributing to open-source projects, companies providing platforms and services, and users influencing the direction of software development through feedback and usage patterns.

2.2 Why the ecosystems evolve.

Following the definition of a software ecosystem, we will explore the reasons behind their evolution. The evolution of software ecosystems is influenced by a variety of factors

ranging from technological advancements to changes in user demand and community dynamics. This section outlines the key drivers of evolution within software ecosystems and how these drivers interact to shape the development and growth of these complex systems.

1) **Technological Innovations.**

   A key factor for the evolution of software ecosystems is technological innovation. New technologies can significantly alter the landscape of an ecosystem by introducing new capabilities, improving efficiency, or creating new opportunities for development. For example, the arrival of cloud computing transformed many software ecosystems by enabling more scalable and flexible deployment options for applications. Similarly, new findings and discoveries in artificial intelligence and machine learning are currently driving significant changes in how software is developed, managed, and utilized within the software ecosystems [13].

2) **Community Engagement and Contributions.**

   The liveliness of a software ecosystem heavily relies on its community of developers, users, and other stakeholders. Community engagement, through contributions to open-source projects, feedback, and collaboration, lead to continuous improvement and expansion of the ecosystem's components. An active and engaged community can quickly adapt to needs that may appear, fill gaps in the ecosystem, and encourage innovation by sharing knowledge and resources. The evolution of an ecosystem is often a reflection of the collaborative efforts of its community [14].

3) **Market Demands and User Needs.**

   Market demands and user need also play a vital role in the reasons behind a software ecosystem's evolution. As users' demands change, the need for the ecosystems to adapt to the new expectations arises. The ecosystems that manage to stay up to date with the market demands and user needs that very often change, are most likely to stay relevant and thrive over time [15].

4) **Regulatory and Environmental Changes.**

   Often changes in regulations or shifts in the economic and environmental contexts may drive the evolution of software ecosystems. To be more precise, regulations about protection, privacy, and cybersecurity, for example, can motivate significant adjustments in how software is developed within an ecosystem. Also, economic trends can impact funding, investment, and the overall direction of technological development, while environmental concerns might drive the adoption of more sustainable development practices [16].

5) **Cross Ecosystem Interaction**

   Additionally, the interaction between different software ecosystems can catalyze evolution by motivating the exchange of ideas and different practices.

   This cross-exchange can lead to the discovery of new tools and platforms that combine the strengths of multiple ecosystems. Cross-ecosystem collaboration can also encourage similarity between ecosystems, making it easier to integrate diverse technologies and components into cohesive solutions [17].

6) **Adaptation and Survival**

   Ultimately, the evolution of a software ecosystem is a matter of adapting to the continuous changes in the technologic landscape, to survive and stay relevant over time. Ecosystems must balance innovation with stability, ensuring that they continue to meet the needs of their community while also exploring new directions and opportunities. The most successful ecosystems are those that manage to create dynamic stability, constantly evolving without losing sight of their core values and objectives [18].

Understanding the factors why ecosystems evolve, and their interplay is crucial for getting to the bottom of how software ecosystems evolve. This knowledge not only sheds light on the current state of various ecosystems but also provides insights into potential future trends and developments.

2.3 Ecosystems used in the research.

In this section, we will take a deeper look into the software ecosystems that are pivotal to my study: Maven, PyPI, RubyGems, CRAN and NPM. While numerous ecosystems exist, these five have been selected based on their significance within the software development landscape. Understanding the key features, and their roles in software development provides the necessary knowledge to understand the dynamics of cross-ecosystem packages.

Maven:

Maven, a cornerstone in the Java ecosystem, is an open-source build automation and project management tool that is used for Java Applications. What Maven basically does is automate the source code compilation so that your source code becomes executable, assembles binary codes into packages and executes test scripts [4]. Using Maven, you can create Java deliverables like JAR, EAR, and WAR files with the help of pom.xml files. A POM file is the base of the Maven framework. It’s an XML file as I said before that accommodates data from your project and configuration details. Maven's dependency management system automates the inclusion of libraries and other project dependencies, streamlining the build process and ensuring consistency across development environments. Also, a key factor of the Maven ecosystem is the central repository which basically provides developers with access to third-party libraries and modules [4]. If we dig into the key features and benefits of using Maven, we can say that it provides a simple project setup for the developers using it and manages dependencies very well.

PyPI: The Python Package Index, also called PyPI is a centralized repository of open-source packages written in Python which are freely accessible to everyone. Today PyPI possesses almost 500,000 projects [5]. It was found in 2002 by an Australian developer called Richar Jones. He released the first version of PyPI in 2003, and over time many other volunteer contributors joined the adventure. Each individual package that is accessible from PyPI has its own page, showing information such as its description, metadata, dependencies, and version history [5)]. As most of the packages are open source, PyPI has encouraged collaboration, sharing and innovation, whether by building programs on this existing solution or by proposing alternatives to them. We have all

undoubtedly used PyPI before by using the command “pip install”. From this simple command anyone can install and use PyPI packages. PyPI is also used by many major companies as it’s the main index for Python packages. Major PyPI users include NASA, IBM, Google, Instagram, etc. [5]. From all this information we can summarize that PyPI is an essential part of the Python ecosystem. It is not just a repository for Python packages but the central hub of the Python ecosystem. Without it the distribution and discovery of such packages would be far more difficult and complex.

RubyGems: RubyGems is a package manager for the Ruby programming language that provides the following: a standard format for distributing ruby programs and libraries, a tool designed to easily manage the installation of gems, and a server for distributing them [6]. It was created by Chad Fowler, Jim Weirich, David Alan Black, Paul Brannan, and Richard Kilmer. The development of RubyGems started in November 2003 and was released to the public on March 14, 2004. The interface of RubyGems is a command line tool called gem which can install and also manage libraries (gems). There is a public repository that helps you find gems, resolve dependencies, and install them. Every gem contains a name, version, and platform. It also consists of code, documentation, and gem specification, also called Gemspec [6]. Since gems run their own code in an app, there are security concerns that this may lead to issues due to installation of malicious gems. The creator of malicious gems may be able to compromise user’s system or server.

CRAN: CRAN stands for Comprehensive R Archive Network and is a repository for the R programming language. CRAN includes the source and compiled versions of R for Windows and Mac along with a lot of packages [7]. Those packages are updated regularly, and they depend on many other packages in a complex graph of dependencies [8]. As the primary repository for R packages, CRAN plays a crucial role in the R ecosystem, supporting statistical analysis, graphical representation, and reporting. CRAN structure ensures easy access to a wide range of statistical techniques, graphical methods, and other tools though packages submitted by the R community of developers. CRAN’s emphasis on quality and reliability, including a safe and careful package submission process, ensures that R users have access to a various set of tools for data analysis and visualization.

NPM: NPM stands for Node Package Manager and is a software ecosystem tham mainly supports Javascript. It is the package manager for Node.js, and it plays a vital role in the Node.js ecosystem. NPM allows developers to discover, install and manage libraries and tools they need for their applications by hosting reusable modules, each one serving specific functionalities [9]. It also enables version management and ensures that all dependencies work together seamlessly. Some of the basic NPM commands are ‘npm init’ and ‘npm install’ [9]. If we dive into the architecture of Node js we can see that it is designed to take advantage of JavaScript event-driven, non-blocking Input/Output model, making it very efficient and scalable for building server-side applications.

Understanding these ecosystems provides basic and foundational knowledge from which we can understand and explore the evolution of cross-ecosystem packages and their patterns of programming languages. Each ecosystem not only supports its respective language and community but also contributes to a larger, interconnected landscape of software development.

2.4 Benefits of software ecosystems.

One of the key benefits of software ecosystems is that the costs involved in software development and distribution are decreased [10]. This benefit can be broken down into several aspects. One of them is that a software ecosystem often provides a shared infrastructure and resources to all participating developers. This may include development tools, libraries, APIs, and cloud storage. By using these shared resources, developers can avoid the costs of developing their infrastructure from scratch. Another factor of the above benefit is that developers in the same ecosystem can reuse software components such as libraries, modules, and packages. By reusing existing software components, developers can reduce the time and effort required to develop new software solutions, leading to lower development costs.

2.5 Challenges of software ecosystems.

In the exploration of software ecosystems, several significant challenges emerge that impact their development, evolution, and sustainability. These challenges are critical to address as they influence the effectiveness and activity of the ecosystems.

One of the primary challenges is the establishment and management of relationships between various actors within the ecosystem. Ensuring these relationships are well-defined and effectively managed is essential for ensuring a collaborative and productive environment. Additionally, software ecosystems face architectural challenges such as maintaining platform interface stability, managing the ongoing evolution of the system, ensuring robust security, and maintaining reliability [10].

Another significant challenge is the heterogeneity of software licenses and the evolution of systems within an ecosystem. Organizations must navigate these complexities to minimize dependency risks and ensure compliance with diverse legal and operational standards. Moreover, differentiating resources within the ecosystem to maintain a competitive edge from the other competitors, and ensuring long-term viability is a considerable challenge [10].

Technical and socio-organizational barriers also pose significant challenges, particularly in coordinating and communicating requirements across geographically distributed projects. Overcoming these barriers is crucial for the seamless flow of information and collaboration that form the basis for successful distributed software development [10].

Also maintaining versioning and backward compatibility within a software ecosystem is crucial for ensuring that newer versions of software components are compatible with older systems. This challenge involves designing APIs and software components that can support both new features without disruption. Strategies like semantic versioning and rigorous testing regimes are often employed to manage this balance effectively [20].

Lastly, the lack of sufficient infrastructure and tools to foster social interaction, decision-making, and development across organizations involved in both open source and

proprietary ecosystems can hold back progress. Developing this infrastructure is vital for supporting the collaborative dynamics that drive innovation and efficiency within software ecosystems [10].

These insights into the challenges faced by software ecosystems are crucial for stakeholders involved in designing, managing, and participating in these ecosystems, providing them with a clear understanding of where focused efforts and resources are necessary to enhance ecosystem health and productivity.

2.6 Features of software ecosystems.

Software ecosystems exhibit several distinctive features that enable them to support sustained growth, collaboration, and innovation within the software industry. They are characterized by a strong architectural framework that includes interface stability, evolution management, security, and reliability, ensuring a supportive infrastructure for ongoing development and adaptation.

One of the pivotal features of many software ecosystems is the adoption of an open-source development model. This model encourages transparency, collaboration, and community-driven development, which are essential for co-innovation and rapid technological advancement. The metaphorical application of biological concepts such as mutualism, commensalism, and symbiosis further emphasize the interconnected relationships within these ecosystems, highlighting the synergy among different ecosystem participants.

In addition, software ecosystems can be used to negotiate requirements to align solutions, components, and portfolios with the needs of users and stakeholders. This alignment is important to maintain the relevance and responsiveness to the market needs. Process innovation within these ecosystems also plays a critical role, enabling continuous improvements and adaptations that enhance efficiency and adaptability to changing environments.

These features not only define the operational dynamics of software ecosystems but also underline their strategic importance in fostering a collaborative and innovative environment within the software industry. [10]

2.7 Historical development of software ecosystems.

The concept of software ecosystems has evolved significantly over the past few decades, paralleling the broader evolution of software development methodologies and the technology landscape. The roots of software ecosystems can be traced back to the early practices of modular software development in the 1960s and 1970s. As software complexity grew, the need for modular, reusable components became apparent. Parnas's seminal work on software modularity presented a foundational approach for designing software that could be easily maintained and extended, setting the stage for later developments in software ecosystems [21].

The 1980s and 1990s witnessed the rise of the open-source movement, which played a pivotal role in the development of software ecosystems. The sharing of source code and collaborative development enabled by platforms like GNU and later, Linux, showcased the power of community-driven development. This era marked a significant shift towards more open, collaborative environments that underpin many modern software ecosystems [22].

The late 1990s and early 2000s saw the rise of commercial software ecosystems with companies like Microsoft, Apple, and Oracle developing extensive platforms around their products. These ecosystems were characterized by a strategic focus on creating a network of complementary products, services, and third-party applications that all revolve around a core technology or platform [23].

With the arrival of the internet and web services in the early 2000s, APIs became a crucial component of software ecosystems, enabling different software applications to interact smoothly. The API economy further expanded the boundaries of software ecosystems by allowing diverse applications to connect, share data, and function collectively, regardless of their underlying platforms [24].

The latest phase in the evolution of software ecosystems is marked by the rise of cloud computing and microservices architectures. These technologies have further decentralized software development, allowing ecosystems to become more scalable, resilient, and faster to adapt to changes. The cloud has enabled ecosystems to extend globally, connecting an even broader range of devices and services [25].

The historical development of software ecosystems reveals a route from tightly coupled, private systems to open, interconnected networks that extend across the globe. This evolution reflects wider technological advances and changing business strategies, illustrating how software development continues to adapt to new challenges and opportunities.

2.8 Characteristics of cross-ecosystem packages.

To understand cross-ecosystem packages thoroughly, let’s re-define them. Cross-ecosystem packages are packages that are distributed and utilized across different programming languages and platforms. They are designed to operate or be shared across multiple software ecosystems. Such packages are cloudeebus, HanLP and waluigi.

These packages can take various forms. For instance, some packages primarily target one language, introducing wrappers to extend their reach into other ecosystems. Others undergo distinct development processes for each ecosystem, showing diverse development approaches.

One of the main characteristics is interoperability, which means that these packages are built to operate seamlessly across different ecosystems [15]. This characteristic is crucial for ensuring that these packages can be effectively utilized within various technological frameworks without compatibility issues. Interoperability facilitates the sharing of functionalities and data across different platforms, enhancing the utility and reach of software applications.

Another characteristic is adaptability. Cross- ecosystem packages can evolve to meet the changing requirements and conventions of multiple ecosystems. This adaptability is essential for maintaining relevance and functionality as the ecosystems themselves evolve due to technological advancements or shifts in user demand [13].

Also, understanding the evolution patterns of cross-ecosystem packages is essential. These patterns can reveal how packages adjust to new ecosystems. The GitHub repositories of these packages can provide valuable insights into their development patterns, through their commits and modified files of each commit. These insights include which languages and, therefore, which ecosystems are targeted from each package.

In conclusion, the unique characteristics of cross-ecosystem packages underscore their pivotal role in modern software development. By facilitating integration across multiple ecosystems and maintaining their relevance through their adaptability, these packages contribute to technological solutions, supporting the continued evolution of software ecosystems.

Section 3

Related work

3.1 Research on software ecosystems.

Existing research delves into the characteristics, benefits, and challenges of software ecosystems. Joshua et al. [10] explored how ecosystems foster collaboration, enable integration, and streamline software distribution, while also highlighting challenges such as dependency management and version control. Lungu [11] examined reverse engineering software ecosystems, highlighting how understanding an ecosystem's structure and interrelations can lead to better management and development strategies. It emphasizes the importance of mapping dependencies to encourage sustainable ecosystems.

Jansen et al. [13] dig into managing business networks as a survival strategy for software ecosystems, examining the dynamics between ecosystems and how collaboration, competition and interdependencies shape their evolution. Fitzergald and Aderfalk [14] investigate the role of open-source software in software ecosystems, discussing how open-source projects contribute to ecosystem growth. Bosch [15] discusses the transition from software product lines to software ecosystems, outlining how ecosystems evolve beyond individual products, embracing networks of related projects that collectively drive innovation and development.

Jansen et al. [16] offer a comprehensive analysis of software ecosystems, exploring how to manage business networks in the software industry. It delves into how ecosystems function as networks of projects, organizations, and contributors. Manikas and Hansen [17] provide a systematic literature review of software ecosystems, summarizing key findings and trends. They emphasize the importance of community engagement, technological innovation, and regulatory factors in shaping ecosystem evolution. Iansiti and Levien [18] discuss strategy in the context of software ecosystems, emphasizing how

regulatory, economic, and technological factors influence ecosystem development and growth.

Jacobson et al. [24] discuss the role of APIs in software ecosystems, emphasizing how they facilitate integration and collaboration, contributing to ecosystem growth and diversification. Newman [25] discusses the design of microservices and their role in software ecosystems, emphasizing how they promote modularity and flexibility, enabling ecosystems to adapt to changing technological landscapes.

3.2 Research on cross-ecosystem packages.

Furthermore, studies have explored the dynamics of cross-ecosystem packages. Constantinou et al. [2] conducted a comprehensive study to uncover the characteristics and evolution of cross-ecosystem packages. They found that only a small fraction of packages is distributed across multiple ecosystems, but these packages have a significant impact on dependency networks within each ecosystem, influencing packages across different ecosystems. Additionally, their study highlighted how these packages tend to favor certain ecosystems over others, based on support and new releases. Their findings also revealed that these packages are more popular and have larger developer communities, underscoring their importance and the need for further investigation.

Complementing this, Kula et al. [12] embarked on a large-scale empirical analysis of 1.1 million libraries from five distinct software ecosystems: PyPI, CRAN, Maven, RubyGems, and NPM. This study focused on libraries released across multiple ecosystems, identifying 4,146 GitHub repositories hosting such libraries. The findings emphasized the interconnectedness of different software ecosystems and how cross-ecosystem libraries are deeply intertwined with the dependency networks of their ecosystems. The study also showed that these libraries attract contributors from within their ecosystems, reflecting the growing community reach beyond a single programming language.

This body of research from multiple papers and contributors provides a comprehensive overview of software ecosystems and cross-ecosystem packages, highlighting their

characteristics, challenges, and opportunities. It lays the foundations for understanding how ecosystems function, evolve and interact and offers insights into their development and sustainability.

Section 4

Methodology

This section describes the methodology employed to extract and analyze data concerning cross-ecosystem packages across the five big software ecosystems that I previously mentioned (Maven, NPM, PyPi, Rubygems, CRAN). The primary goal was to identify patterns in the programming development, maintenance, and evolution of these packages. To this end, a multi-method approach was employed, combining Python and its powerful data manipulation library, Pandas, to handle and analyze the data. Additionally, to retrieve that data, PyDriller was utilized. PyDriller allowed me to extract all the necessary data from these packages, providing me useful insights into their structural and functional dynamics. In the next small sections I will analyze how I retrieve those insights and how I used them to come up with the patterns I was looking for.

4.1 Data Collection

Before I started the data collection process, I first had to discover the cross-ecosystem packages. To accomplish this, I used Kula et al [12] study and obtained detailed csv files for the 5 main ecosystems of my research. Those comma-separated values files contained all the repositories that take part in each ecosystem. So, to discover which of them appear in more than one ecosystem I first loaded them using python's library Pandas in a separate DataFrame for each one of them. Then, all the individual DataFrames were concatenated into a single big DataFrame. This step was crucial as it merged data from different ecosystems into a unified structure, setting the stage for cross-ecosystem analysis. Pandas

‘concat’ function was used for this purpose, ensuring that indices were reset, and no data alignment issued occurred. The unified DataFrame was then grouped by the ‘Repository URL’ to aggregate data based on unique repository addresses. This aggregation was essential to identify and count the unique platforms associated with each repository. Two main aggregations were performed: counting unique platforms and concatenating platform names into a single string for each repository. Repositories appearing in more than one ecosystem were filtered out using a condition that selected repositories that the count of unique platforms was greater than one. This step pinpointed the repositories that are shared across multiple ecosystems, highlighting their cross-ecosystem nature. Finally, the filtered data, which now only contained the repositories that appeared in more than one ecosystem, was exported to a csv file.

After producing this csv with all the repositories appearing in multiple ecosystems, I was ready to start collecting data about them. The data collection process for this thesis was structured around extracting detailed commit and file modification data from those multiple software repositories. This was achieved using Python, with the help of the PyDriller library, a tool specifically designed for mining data from Git repositories. Pydriller allows to extract a lot of details about the commits of the repository (such as its hash, message, author, committer, author date, committer date, modified files, project name, insertions, lines, files) and also information for the modified files of each commit (such as its old path, new path, filename, added lines, deleted lines, source code, methods, lines of code). The procedure was carefully planned to automate extraction, ensure accuracy, and handle potential errors efficiently and effectively.

Initially, we imported the list of cross-ecosystem packages that we initially extracted using Pandas. Each repository URL from that csv file was processed to clone the repository locally, allowing direct interaction with the repository’s data. Using PyDriller’s ‘Repository’ module, the script traversed trough each commit in the cloned repositories. For every commit relevant data such as: Commit Hash, Commit Message, Commit Author Name, Commit Author Email, Commit Committer Name, Commit Committer Email, Commit Authored Date, Commit Committer Date, Branches, Merge, Commit Parents, Project Name, Project Path, Number of Deleted Lines, Number of Inserted Lines, Lines, Number of files changed were captured. Moreover, for each

commit, details about the modified files were extracted such as: Commit Hash, Old Path, New Path, File Name, Type of the Change, Diff, Diff Parsed, Added lines, Deleted Lines, Source code, Source code before, Methods, Methods before, Changed methods, Lines of code, Complexity, Token Count. The key connecting the modified files to the appropriate commit was the commit hash.

The extracted data for commits and modified files were then written into CSV format file using Python's CSV library, ensuring that the data could be easily accessed and analyzed in subsequent stages. These CSV files were zipped to optimize storage and maintain the integrity of the data during the transfer or backup processes. Each zip file was named after the name of the repository.

Throughout the data collection process, various exceptions were carefully handled to ensure robustness. Errors such as 'GitCommandError', 'ValueError', 'NotADirectoryError', 'MemoryError', 'calledProcessError' were captured, and for the repository causing them I left a comment on it, to know that it is problematic for that reason. If there were no errors while extracting the data for a certain repository, then I marked the comment column of that package as 'Done'. This error handling process was crucial to maintain the continuity of the data extraction process across all repositories, ensuring that any issues were documented and could be addressed without disrupting the over data collection workflow. After this process data for 3304 of the total 3987 repositories, was extracted without any errors.

The methodology adopted for data collection not only facilitated a comprehensive extraction of the required data but also ensured that the process was scalable, repeatable, and efficient, laying a solid foundation for the detailed analysis that would follow in the study.

4.2 Adding extension and Language information.

Following the initial step of data collection, the next phase of the methodology involved enhancing the data that I extracted with some additional useful information for further use. Specifically, each modified file's programming language and file extension were

determined and added to the dataset. This enrichment of the data was crucial for further analysis, as it allowed a more detailed insight into the nature of changes across different software ecosystems. The programming language of the file is instrumental in enabling this analysis, as ecosystems are programming language based.

The first step of this data enrichment process was the creation of a language mapping based on the file extensions, which was derived from the classifications provided by the CLOC tool [26]. CLOC offers extensive coverage of various file types categorized per programming language, enhancing the reliability and comprehensiveness of the mapping. A dictionary was constructed to map each file extension to its corresponding programming language. This 'dictionary' included common programming languages such as C, Java, Python and less common ones like Elixir and Objective-C, ensuring broad coverage.

The mapping was implemented in Python, utilizing a dictionary structure where keys represented file extensions and values represented the corresponding programming languages. This dictionary was then written to a CSV file to serve as a persistent and easily accessible reference for language identification. The CSV file creation involved using Python's 'csv' module to write the extension-language pairs into a file, ensuring that this vital information was correctly recorded and retrievable for future processing steps.

The main data enrichment process was conducted by another Python script that processed each entry in the dataset of modified files. First, the language mapping CSV file was loaded into a DataFrame, and a dictionary was created from it to facilitate quick lookup operations. This setup ensured that the mapping could be efficiently accessed during the data processing.

A function was designed to extract the file extension from each file name. This function also handled edge cases such as files without extensions and specific files like 'LICENSE' which was treated as 'Unknown'. Such files are important, but not dedicated to a specific programming language, therefore they must be excluded from the pattern finding process. Using this function, for each file in the data (modified files) the script calculated the file

extension and the programming language. These values were then added as new columns to the DataFrame, providing a richer dataset that included both the type of file and the programming context. The enriched DataFrame was then saved into a new CSV file within the ZIP file. This process was repeated for the full set of packages.

4.3 Commit Time Analysis and Threshold Determination.

The subsequent phase of the methodology involved analyzing the timing of commits for each programming language within the repositories and establishing thresholds to classify commit intervals as either ‘Gap’ or ‘No Gap’. This step of the methodology is important for understanding the activity patterns.

First, each repository’s ZIP file, which contains the modified_files_languages and commits csv files was processed to merge these two datasets. The merging of these two csv files was achieved by first extracting them from their CSV files within the ZIP file, then the modified files data was filtered to exclude entries that belong to languages who were non-programming, such as README or Git configuration files. After that, the remaining modified files data was merged with the commits data on the ‘Commit Hash’ field. As previously mentioned, the ‘key’ connecting those two files is ‘Commit Hash’ as it associates each commit with the programming languages of the files affected in that commit.

For each programming language in the data, the entries were sorted by the ‘Commit Authored Date’ and then the difference between two consecutive commits of that language was calculated. This step was crucial for determining the frequency and regularity of commits per language. Statistical analysis was due next, where the average of the time differences and the standard deviation of those time differences were calculated. These two statistics provided insights into the typical commit intervals and the variability of each language context.

An important part of this phase was setting the threshold for each Language to classify the commit intervals. For each language, the threshold was defined as the sum of the average time difference and the standard deviation of the time differences, the two

statistics that were calculated before. This threshold helped identify unusually long intervals between commits, classified as ‘Gap’, and shorter intervals classified as ‘No Gap’. The way that those intervals were classified was simple. Intervals exceeding the threshold were labeled as ‘Gap’, while shorter intervals were labeled as ‘No Gaps’. This classification was added to the dataset, enhancing the data with significant insights into the commit intervals.

Two CSV files were generated during this process, the ‘thresholds’ and ‘commit_time_difference’. The ‘thresholds’ CSV file contained detailed statistics for each language, including average time differences, standard deviation, the threshold, which is the sum of the previous two, and the-time difference between the first commit containing that language and the first commit of that repository, as well as the difference between the last commit containing that language and the last commit of the repository. The ‘commit_time_difference’ documented each commit’s timing details, including the programming language, the date and time of the actual and the next commit, the calculated time difference between the previous two, the time difference in days, and the classification of that interval (Gap or No Gap).

4.4 Language Consistency verification process.

An essential step in my methodology consisted of verifying the consistency of programming languages used in each repository against the expected languages based on the repository’s associated platforms. This verification ensured that the subsequent analysis and visualization would be meaningful and based on accurate repository data.

Initially, a predefined mapping was established to associate each software platform with its typical used programming language. To be more specific, repositories on Maven were expected to use Java, repositories on PyPi were expected to use python, repositories on CRAN were expected to use R, repositories on NPM were expected to use JavaScript and repositories on Rubygems were expected to use Ruby. This mapping was crucial to make sure which languages should be present in the repositories associated with each platform.

The process began with the extraction of existing language data from each repository. For each repository listed as ‘Done’ in the preliminary report (shared_repos_report_final.csv), indicating that it had previously been processed without errors, the associated ZIP file was accessed to extract language data. This was done by reading the ‘thesholds.csv’ file from the ZIP archive, which as we mentioned before, contained information about the languages detected in the repository during the earlier analyses.

For each repository, the languages extracted from the thresolds.csv file were then cross-verified against the expected languages derived from the platform-language mapping we mentioned before. This verification involved:

- Extracting the platforms associated with the repository from the preliminary report (shared_repos_report_final.csv).
- Finding the expected languages for these platforms based on the mapping.
- Counting the number of expected languages that matched the languages actually found in the repository (through the thresholds.csv)

If fewer than two of the repository’s expected languages were found in the repository’s actual data, then it was flagged in the comment section as: ‘Repository languages do not match’. This flagging was important because it indicated an ‘issue’ for those repositories, which was that they were not actually supported by 1 or more platforms so there was no point in further analyzing it. These cases may occur because of a wrapper. The package is probably supporting both languages, but don’t commit in both languages.

Finally, the updated data, including the comments about language inconsistency was written back to a final CSV file. This file served as a comprehensive record of each repository’s status regarding language consistency, informing the next steps in the methodology and ensuring that only repositories who are supported by more than 1 platform are further analyzed.

4.5 Visualizing Commit Activity over time.

The next step in the methodology involved visualizing the timeline of each repository's commit activity across its main programming languages. The main criterion for a repository to undergo this process was to pass the language consistency check in the previous step and its 'Comment' column to still be marked as 'Done'. This visualization was created by marking each month with colors corresponding to the activity status ('Gap' or 'No Gap') for each language. This part of the entire process aimed to provide a clear, visual representation of the temporal patterns of commits, facilitating easy identification of active and inactive periods across different programming languages within each repository.

Firstly, the process starts by identifying the expected languages for each repository based on its platform. A mapping of platforms to their main languages (e.g., NPM to Javascript, PyPi to Python) is used to determine these expected languages. This mapping ensures that the analysis focuses on relevant languages avoiding other less significant languages.

For each repository listed in a centralized CSV file, the script extracts the repository name and locates the corresponding ZIP file containing the commit data. If the Zip file exists, it proceeds to process the data. If it does not exist, the repository is skipped, and a log message is generated.

The commit data is read from a 'commit_time_differences.csv' file within the repository's ZIP file. The authored dated of the commits are parsed, and languages are standardized by stripping any leading or trailing whitespace. The script then filters this data to include only commits in the languages determined to be relevant from the platform mapping.

Also, a full range of commit dates is calculated from the earliest to the latest commit. Based on this range, a list of months is generated. For each language present after filtering, the script initializes a DataFrame where each row represents a month, and each column represents a language. All cells are initially set to 'Gap' indicating no activity. For each language, the script identifies the months in which commits were made and

marks these in the DataFrame as ‘No Gap’. This marking is based on the actual commit authored dates, reflecting active development periods of that language within the repository.

An excel workbook is created to visually represent this timeline. Cells are colored green for ‘No Gap’ and red for ‘Gap’. This coloring provides a visual representation of repository’s activity over its whole timeline, making it easy for me to spot trends and patterns amongst the repository’s main languages.

Once the workbook is formatted, it is saved into an in-memory buffer and then written back into the repository’s ZIP file as a new Excel file named as: {repository’s name} _commit_activity_final.csv. This file serves as a final, visual summary of the repositor’s activity over time, providing valuable insights into the development dynamics based on the filtered programming languages.

4.6 Discrepancy Percentage calculation.

The discrepancy percentage is a crucial metric in the study of cross-ecosystem packages to come away with evolution patterns. It quantifies the difference in commit activity between the two languages associated in a repository, expressed as a percentage over the whole series of months of the repository. To make it clearer, I provide some examples below:

Figure 1:

Month	Python	Javascript
2013-10	No Gap	Gap
2013-11	No Gap	No Gap
2013-12	No Gap	No Gap
2014-01	No Gap	No Gap

Figure 1 is libcredit’s[1] package timeline of commits. As we can see the only month that the 2 main languages of this repository (Python and Javascript) have difference in their

[1] https://github.com/commonsmachinery/libcredit

commit activity (the one is Gap, and the other is No Gap) is October of 2013. Based on the calculation of the discrepancy percentage this is 25%.

Figure 2

Month	Java	Ruby
2013-07	No Gap	No Gap
2013-08	Gap	No Gap

Figure 2 is Message-Bus's [2]package timeline. Here the only difference is August of 2013 where it is Gap in Java (there are no commits modifying java files in that month) but there is activity about Ruby. For this example, the discrepancy percentage is 50%.

This metric, as can be seen from the examples above, provides insights into the development patterns and synchronization between different languages within cross-ecosystem packages, highlighting how well these packages integrate and evolve across ecosystems.

To calculate the discrepancy percentage for repositories associated with exactly two platforms, information about repositories was gathered from a centralized CSV file (called final_repos.csv), which contains details about each repository, including its URL, the platforms it belongs to, and the count of platforms. Inside this CSV file, there are only the repositories that passed the check about Language Consistency that I mentioned before. Only repositories associated with exactly two platforms were selected for further analysis, and the calculation of the discrepancy percentage. That way we ensured the focus remained on packages spanning across two different ecosystems, making the discrepancy percentage particularly meaningful.

For each repository, the commit activity data was extracted from the commit_activity_final excel file contained within the repository's ZIP archive. This Excel file provides a timeline of the repository's activity, recording the status of each language's commit activity (Gap or No Gap) for each month, as we also mentioned before. The excel file was then processed to count the number of months with discrepancies between the

[2] https://github.com/groupon/Message-Bus

two languages' activities. A discrepancy was recorded for a month if one language showed a 'Gap' (indicating there were no commits including this language, during that month) while the other showed 'No Gap' (indicating there were commits including this language, during that month). The total number of months was also counted.

The discrepancy percentage was then calculated as the ratio of discrepancy months to the total number of months, reflecting how often the two languages' activities are out of sync, indicating differences in their development patterns. The discrepancy percentages, along with the repositories' information, were saved to a new CSV file (called final_repos_months.csv), serving as a comprehensive record of each repository's discrepancy percentage (and other metrics in the future), providing valuable data for further analysis or reporting.

The discrepancy percentage metric offers crucial insights into the synchronization between different programming languages within cross-ecosystem packages. By quantifying how often these languages' activities diverge, we highlight potential areas for improvement in package integration and development strategies. Understanding these patterns is essential for identifying different patterns, to then classify each repository based on its commit activity.

4.7 Language Percentage calculation.

The language percentage is a metric that measures how much of the total timeline a given programming language shows active development (No Gaps). To make it clearer, we are going to revisit the previous examples:

Figure 3:

Month	Python	Javascript
2013-10	No Gap	Gap
2013-11	No Gap	No Gap
2013-12	No Gap	No Gap
2014-01	No Gap	No Gap

For the libcredit package as we can see from the timeline of the two languages (Python and Javascript), Python has been active for the whole timeline of the package. Therefore, its percentage is 100%. On the other hand, Javascript has only been active for 3 of the 4 months so for Javascript the percentage is 75%.

Figure 4:

Month	Java	Ruby
2013-07	No Gap	No Gap
2013-08	Gap	No Gap

In the Message_Bus case we can clearly see that Java misses out in August of 2013 so Java's percentage is 50% and Ruby has activity throughout the whole timeline, which makes it 100% percentage.

This metric offers insights into the consistency and continuity of development across different languages within cross-ecosystem packages, highlighting their integration and evolution over time.

To calculate the language percentage of each language of a package, commit activity data for each repository is extracted from the Excel file (commit_activity_data) stored inside each package's ZIP file. This file, as we also mentioned before, records the activityfor each month for the languages of the package. This Excel data is read into a DataFrame and its values are converted into numerical values: 'No Gap' is mapped to 1, and 'Gap' is mapped to 0. This conversion simplifies the calculation process, making it easy to sum up the active months for each language.

For each main language (the five languages associated with the big five ecosystems I am working with), the number of months with active commits on that language is counted. This count is then divided by the total number of months of the package to calculate the percentage of time the language was actively used in development. This percentage is recorded for each language, offering a direct comparison between them. Additionally, the difference between the percentages of the two languages is computed, highlighting the disparity or balance in their usage over the repository's timeline. This difference provides

further insights into how well the two languages are integrated and whether one language dominates the development cycle.

The language percentages and difference between them are stored in the same CSV file which consolidates the information about the Discrepancy percentage as well. This file serves a comprehensive record of those two metrics that will be crucial to extract patterns in the language usage.

The language percentage metric offers a crucial understanding of the development patterns and consistency of cross-ecosystem packages. By calculating how much of the total timeline each language shows active development, it highlights the balance or disparity between different languages, reflecting their integration into the project's ecosystem.

4.8 Identifying possible Significant Language Shifts.

In addition to the primary analyses, the methodology I used to identify the patterns incorporates a specialized process to identify possible significant language shifts within repositories. This involves detecting periods where a programming language has sustained activity for six or more consecutive months, followed by a period of inactivity (gaps), suggesting potential shifts in language usage. I chose the six months threshold to set a language as significant language for the repository based on the thought that it strikes a balance between sensitivity and specificity in identifying meaningful shifts. While shorter thresholds might lead to false positives by capturing temporary fluctuations, longer thresholds could overlook shorter-term shifts or periods of less intense development. Additionally, six months of inactivity in an active project is less likely to be a random occurrence and is more indicative of a shift in language usage, making it a reasonable sign for detecting significant language shifts.

The whole process starts by reading the commit activity data from Excel, where each months' activity for each language is recorded, as previously explained. The data extracted from the commit activity excel file is then processed to replace the textual representations of 'Gap' or 'No Gap' with 0 and 1 (0 for Gap and 1 for No Gap). That

way the identification of shifts is simplified. For each language in a repository, the script tracks the continuity of active months and detects when a shift to inactivity occurs after a sustained period of activity. If a language shows active development for at least six consecutive months followed by a single gap, it is then flagged as a possible significant language shift.

This detection mechanism is applied to every repository. The script analyzes the data of the package for shifts and records the corresponding dates of any possible shifts. Repositories with detected possible shifts are extracted in a csv file, providing a base for further manual verification to determine if these shifts correspond to actual language migrations.

While manually looking each of those possible language shifts extracted in a separate csv file, I was also monitoring the metrics of each repository (Discrepancy percentage and Language Percentage). These two metrics offer additional insights into the consistency and synchronization of development efforts between the two languages of the repository. This dual approach of both monitoring the calculated metrics of the package but also manual verification I performed on the commit activity timeline, ensured an in-depth analysis of significant language shifts.

This sectiondetailed the methodology used to analyze cross-ecosystem packages across Maven, NPM, PyPi, Rubygems and CRAN, aiming to identify patterns in their development, maintenance, and evolution regarding their programming language use. Using Python, Pandas and PyDriller mainly, the process involved several critical steps. From data collection using the PyDriller, to data cleaning from the Language Consistency verification, to calculating metrics such as Discrepancy Percentage and Language Percentage to analyze all the data I collected. This comprehensive approach ensured reliable data handling and analysis, setting the stage for revealing the evolution patterns in the subsequent results chapter.

Chapter 5

Results

This chapter presents the findings from the extensive analysis conducted on the cross-ecosystem packages across the five major software ecosystems: Maven, PyPi, NPM, Rubygems and CRAN. Through the metrics I calculated and manual inspection of each repository's commit activity, I have successfully identified patterns in programming language usage within these packages. The metrics, including the Discrepancy Percentage as well as the Language Percentage for each language, alongside the manual review of package's timelines, have provided a reliable framework for understanding how programming languages are adopted, used, and shifted over time in these cross ecosystem packages. The next sections will dive into these patterns that I found, providing useful information and examples for each one of them.

5.1 Pattern 1: Base language with light support of other language.

In my analysis of cross-ecosystem packages, one pattern that emerged is the "Base Language with light support of other language" pattern. This pattern is characterized by the presence of a dominant base language, while lightly supporting another language. This pattern is particularly notable in repositories where the metric of the Discrepancy percentage is high and there is also large Language percentage difference between the two languages. Below we can see some examples:

Figure 5

Month	Javascript	Ruby
2013-02	No Gap	Gap
2013-03	No Gap	Gap
2013-04	No Gap	No Gap
2013-05	No Gap	Gap
2013-06	No Gap	Gap
2013-07	No Gap	Gap
2013-08	No Gap	Gap
2013-09	No Gap	Gap
2013-10	No Gap	Gap

In Figure 5 we have ember-auth's [3] timeline of activity. In the second column we see the continuous support of Javascript (100% Language Percentage) since all time periods are No Gap. On the other hand, we have activity in Ruby in only one of the total nine months of the total timeline (that makes it only 11.1% Language Percentage). High Discrepancy Percentage (88.9%) and at the same time high Language Percentage Difference, classifies this package as Pattern 1.

Figure 6

Month	Python	Javascript
2017-04	No Gap	Gap
2017-05	No Gap	Gap
2017-06	No Gap	Gap
2017-07	Gap	Gap
2017-08	No Gap	Gap
2017-09	Gap	No Gap

In the second example provided we have the timeline of the resources-api-v1[4] package. As we can see in this example, Python is the most significant language with a Discrepancy Percentage of 83.3%, and the Language Percentage Difference equals 50%.

Characteristics of Pattern 1:

This pattern is identified through a combination of high discrepancy percentage and a considerable Language percentage difference between the 2 languages used in that package. Specifically, repositories falling into this pattern have a discrepancy percentage

[3] https://github.com/heartsentwined/ember-auth
[4] https://github.com/schul-cloud/resources-api-v1

greater than 50% and a language percentage difference greater than 40%. These 2 metrics combined suggest that one language is consistently used more actively than the other, which only plays a minor or supporting role.

I choose the threshold of 50% for the discrepancy percentage and 40% for the Language Percentage difference by calculating the average results for each metric and by manually inspecting example packages of different numbers on these 2 metrics. I used the same methodology for every other threshold I used in defining the patterns.

5.2 Pattern 2: Parallel support in both languages.

In my exploration of cross-ecosystem packages, another distinct pattern identified is the "Parallel Support in both languages" pattern. This pattern is characterized by minimal discrepancy percentages, indicating that both languages used within the repository in the same way (used or not used actively) with similar levels of commitment over time. Below we can see some examples:

Figure 7:

Month	Python	Javascript
2014-02	No Gap	No Gap
2014-03	No Gap	No Gap
2014-04	Gap	Gap
2014-05	Gap	Gap
2014-06	No Gap	No Gap
2014-07	Gap	Gap
2014-08	Gap	Gap
2014-09	No Gap	No Gap

Figure 7 displays tatl's[5] package timeline. As we can see for the first 2 months, there is commit activity for both languages. This is followed by 2 months of inactivity, for both languages again. This is a typical example of this pattern. Notably, tatl's Discrepancy percentage is 0%, reflecting identical commit activity for both languages throughout the entire timeline, thereby perfectly aligning with the characteristics of this pattern.

[5] https://github.com/tln/tatl

Figure 8:

Month	Python	Javascript
2019-01	No Gap	No Gap
2019-02	No Gap	No Gap
2019-03	No Gap	No Gap
2019-04	No Gap	No Gap
2019-05	No Gap	No Gap
2019-06	Gap	Gap
2019-07	No Gap	No Gap
2019-08	No Gap	No Gap
2019-09	Gap	Gap
2019-10	Gap	No Gap
2019-11	No Gap	No Gap
2019-12	Gap	Gap
2020-01	Gap	Gap
2020-02	Gap	Gap
2020-03	Gap	Gap
2020-04	Gap	Gap
2020-05	Gap	Gap
2020-06	No Gap	No Gap
2020-07	Gap	Gap
2020-08	Gap	Gap
2020-09	Gap	Gap
2020-10	Gap	Gap
2020-11	Gap	Gap
2020-12	Gap	Gap
2021-01	Gap	Gap

Figure 8 is a big part of the dash-extendable-graph [6]commit activity timeline. Observing the results, it is evident the commit activity between the languages only differs in one month of this big part of the timeline. Given that the total timeline for commits in this package spans 43 months, such a minimal discrepancy, nearly close to 0%, categorizes this case under Pattern 2.

Characteristics of pattern 2:

Pattern 2 is marked by a very low discrepancy percentage- specifically, less than 5%. This statistic suggests that the activity between the two languages in these repositories is almost synchronous, with both languages showing similar levels of active development across the timeline. In repositories classifying in this pattern, the absence of significant gaps in development activity between the two languages implies a balanced use where both languages are essential to the project's operations.

The code used to classify repositories into this pattern checks for a discrepancy percentage that falls within this low range, ensuring that the languages are supported and evolved in almost parallel.

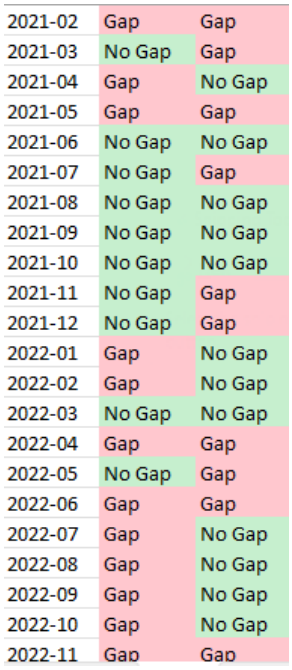
[6] https://github.com/bcliang/dash-extendable-graph

In summary, Pattern 2 reveals a scenario where cross-ecosystem packages benefit from balanced and harmonious language usage, maintaining consistency and synergy across different programming languages within the same package.

5.3 Pattern 3: Interchanging support in both languages.

In the analysis of cross-ecosystem packages, Pattern 3 emerges as "Interchanging Support in Both Languages.". This pattern is characterized by a lack of a clear dominant language but notable differences in the commit activity between the two languages and therefore notable numbers in the Discrepancy Percentage metric. Unlike pattern 1 where one language consistently leads, Pattern 3 indicates a more dynamic interchange where there is not a clear significant language. Below we can see some examples of this pattern:

Figure 9:



2021-02	Gap	Gap
2021-03	No Gap	Gap
2021-04	Gap	No Gap
2021-05	Gap	Gap
2021-06	No Gap	No Gap
2021-07	No Gap	Gap
2021-08	No Gap	No Gap
2021-09	No Gap	No Gap
2021-10	No Gap	No Gap
2021-11	No Gap	Gap
2021-12	No Gap	Gap
2022-01	Gap	No Gap
2022-02	Gap	No Gap
2022-03	No Gap	No Gap
2022-04	Gap	Gap
2022-05	No Gap	Gap
2022-06	Gap	Gap
2022-07	Gap	No Gap
2022-08	Gap	No Gap
2022-09	Gap	No Gap
2022-10	Gap	No Gap
2022-11	Gap	Gap

Figure 9 is a big part of the swim's[7] timeline. By observing the results from the commit activity excel file, we can clearly see that there is no significant language (Language percentages helps us confirm it: 35% Language 1, 66% Language 2) and there is a significant number of months that the commit activity differs between the languages (Discrepancy Percentage is 55%).

[7] https://github.com/swimos/swim

Figure 10:

2018-06	No Gap	No Gap
2018-07	Gap	No Gap
2018-08	No Gap	No Gap
2018-09	Gap	Gap
2018-10	Gap	Gap
2018-11	Gap	No Gap
2018-12	Gap	No Gap
2019-01	Gap	Gap
2019-02	Gap	Gap
2019-03	Gap	Gap
2019-04	Gap	No Gap
2019-05	No Gap	No Gap
2019-06	No Gap	Gap
2019-07	No Gap	No Gap
2019-08	Gap	Gap
2019-09	Gap	No Gap
2019-10	Gap	No Gap
2019-11	Gap	Gap
2019-12	Gap	Gap
2020-01	No Gap	No Gap
2020-02	Gap	Gap
2020-03	Gap	Gap
2020-04	Gap	Gap
2020-05	No Gap	Gap
2020-06	No Gap	Gap

Another example of this pattern is the stellarstation-api [8]package visualized in figure 10. In this case also, there is no significant language from the above visualization and some discrepancy in the commit activity of the two languages.

Characteristics of Pattern 3

Pattern 3 is identified in repositories where the discrepancy percentage is notably high (over 45%), yet the percentage difference between the two languages remains relatively low (below 40%). This indicates that while both languages are used at the same level, their periods of activity differ over time, without one language consistently being more "active" than the other.

In summary "Pattern 3: Interchanging support in both languages" reflects a flexible approach to software development within cross-ecosystem packages, where the interchanging support of languages facilitates comprehensive and adaptable project development. This pattern offers a unique perspective on how diverse technological environments and programming languages coexist over the life of a project, adapting to its changing needs and opportunities.

[8] https://github.com/infostellarinc/stellarstation-api

5.4 Pattern 4: Language migration.

Pattern 4, identified as “Language Migration”, represents a significant shift in the dominant programming language within a repository. This pattern is characterized by a transition from one of the main languages to another, as evidenced by continuous activity in a new language that replaces the previously dominant one. There might be some slight but insignificant overlap during migration. Below we can see some examples:

Figure 11:

2019-06	No Gap	Gap
2019-07	No Gap	Gap
2019-08	No Gap	Gap
2019-09	No Gap	Gap
2019-10	No Gap	Gap
2019-11	No Gap	Gap
2019-12	No Gap	Gap
2020-01	Gap	No Gap
2020-02	No Gap	No Gap
2020-03	Gap	No Gap
2020-04	Gap	No Gap
2020-05	Gap	No Gap
2020-06	Gap	No Gap

Figure 11 is the part of the HanLP[9] package where the switch happens. As we can clearly see from the visualization of the timeline, the significant language switches from Language 1 (Java) to Language 2 (Python).

[9] https://github.com/hankcs/HanLP

Figure 12:

No Gap	Gap
No Gap	Gap
No Gap	Gap
No Gap	Gap
No Gap	Gap
No Gap	Gap
No Gap	No Gap
No Gap	No Gap
No Gap	No Gap
No Gap	No Gap
Gap	No Gap
No Gap	No Gap
No Gap	No Gap
Gap	No Gap
Gap	No Gap
Gap	No Gap

Figure 12 is the part of the Recongizers-Text [10] package where the language migration happens. This example illustrates a smoother transition from Language 1 (Javascript) to Language 2 (Python). Here we can also see that there is also some overlap between the languages, but it seems that the significant language is changing.

Characteristics of Pattern 4

Language migration is detected when a repository shows a clear switch in the primary language used for development, following a period where another language had maintained sustained activity. This transition is identified through an analytical process that tracks each language's active months, following a period of reduced or no activity (Gaps) in its usage. This shift suggests a strategic decision to adopt a new primary language, possibly due to technological, team, or project direction changes.

Detection process

The code I used for this pattern did not automatically classify packages to this pattern. The code analyzed commit activity excel files from all the repositories to identify possible shifts in programming language. This is done by tracking the continuity of active months for each language and noting when a shift to a new language has possibly occurred. More

[10] https://github.com/Microsoft/Recognizers-Text

information about this process can be found in the Methodology sub-section: Identifying Possible Language Shifts.

In conclusion, Pattern 4 offers interesting insights into the strategic shifts in programming languages within projects. By identifying and analyzing these migrations, developers can gain valuable information into the adaptability in software development, ensuring that the projects they are working on remain relevant and efficient in the process of changing technological landscapes.

5.5 Pattern 5: Attempt Success.

Pattern 5, called "Attempt Success", describes a scenario within cross-ecosystem packages where a new programming language successfully gets integrated into the project's development workflow. This pattern is identified by the entry of a second language into the project after the initial 25% of the project timeline, and for the following time the 2 languages show similar levels of activity. This pattern reflects successful adoption and sustained use of a new language alongside the primary language. Below we can see some examples:

Figure 13:

2018-01	No Gap	Gap
2018-02	No Gap	Gap
2018-03	No Gap	Gap
2018-04	Gap	Gap
2018-05	No Gap	Gap
2018-06	Gap	Gap
2018-07	No Gap	Gap
2018-08	No Gap	Gap
2018-09	No Gap	Gap
2018-10	No Gap	No Gap
2018-11	No Gap	No Gap
2018-12	No Gap	No Gap
2019-01	No Gap	No Gap
2019-02	No Gap	No Gap
2019-03	No Gap	No Gap
2019-04	Gap	No Gap
2019-05	No Gap	No Gap

In Figure 13 there is a part of the 0x-monorepo [11]package. During October of 2018 the second language of the repository is introduced. After its introduction the second language shows an active commitment to the repository's workflow by showing consistent 'No Gaps'.

Figure 14:

2010-01	No Gap	Gap
2010-02	No Gap	No Gap
2010-03	No Gap	Gap
2010-04	No Gap	Gap
2010-05	No Gap	Gap
2010-06	No Gap	Gap
2010-07	No Gap	No Gap
2010-08	No Gap	No Gap
2010-09	No Gap	No Gap
2010-10	No Gap	Gap
2010-11	No Gap	Gap
2010-12	No Gap	Gap
2011-01	No Gap	No Gap
2011-02	No Gap	No Gap
2011-03	No Gap	No Gap
2011-04	No Gap	No Gap

Figure 14 is the point of introduction of the second language in the thrift[12] package. As we can see after its introduction, the second language has a complimentary role, but still is actively engaged in the development of the package.

Characteristics of Attempt Success

This pattern is particularly interesting when a new language not only enters the development cycle but also achieves a balance in usage with the existing primary language. The criteria for identifying this pattern include:

1) **Late Entry:** The second language appears after the first 25% of the total package timeline.
2) **Consistent Engagement:** Following its introduction, the secondary language maintains a continuous presence in the timeline, marked by the absence of significant gaps.

[11] https://github.com/0xProject/0x-monorepo
[12] https://github.com/apache/thrift

3) **Balanced Activity:** Post-introduction, both the primary and secondary and secondary languages exhibit a balanced level of activity. This balance is quantified by ensuring that the difference in activity levels between the two languages is less than 50%, showing that new language is not supplementary but an integral part of the development.

Detection Process

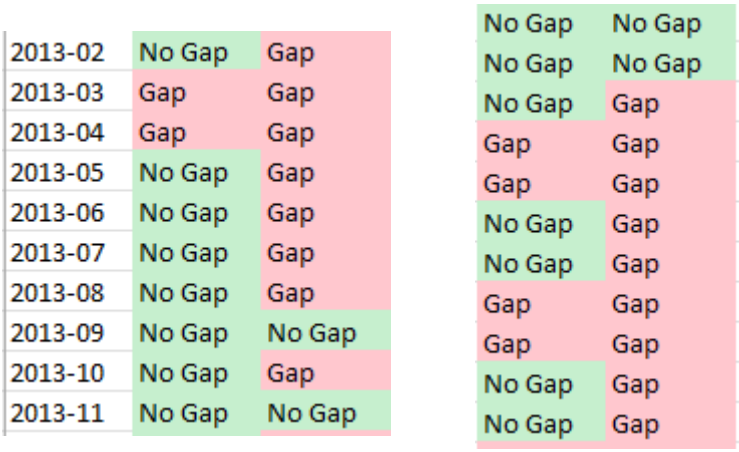
The detection of this pattern involves a detailed examination of the commit activity data extracted from the package's repositories. A script analyzed the activity status (Gaps or No Gaps) for each language over time, focusing on the presence of gaps before and consistent activity after the introduction threshold for the other language. The script ensures that the secondary language, once introduced, does not show big difference in activity compared to the primary language, confirming its successful integration and parallel development.

In summary, Pattern 5 reveals a dynamic aspect of software development within cross-ecosystem environments, where new technologies are not only tested but also successfully integrated into the ongoing project workflow. This pattern provides insights into the evolving nature of software projects and the successful management of multiple programming languages, contributing to a richer, more flexible development environment.

5.6 Pattern 6: Attempt Failure.

Pattern 6, referred to as "Attempt Failure", characterizes a situation where a new programming language is introduced into a project after the 25% mark, but fails to sustain its presence until the end of the project timeline. This pattern identifies an unsuccessful attempt to incorporate a new language into the project's development workflow, resulting in its abandonment before project completion. Below we can see some examples of this pattern:

Figure 15:

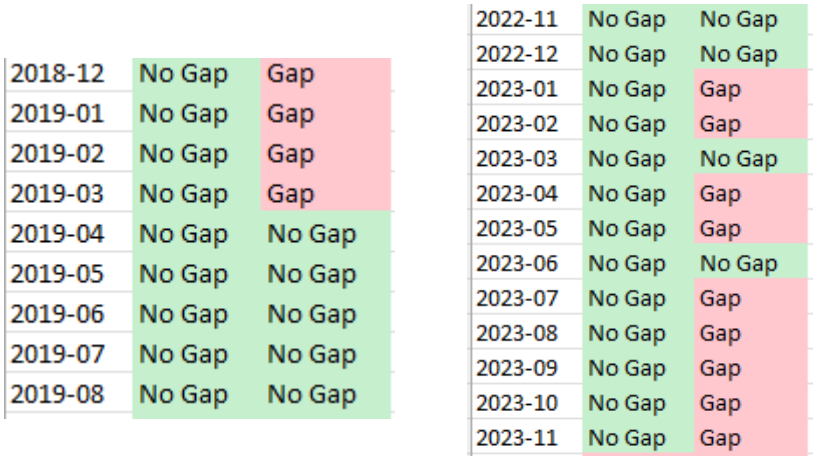


2013-02	No Gap	Gap
2013-03	Gap	Gap
2013-04	Gap	Gap
2013-05	No Gap	Gap
2013-06	No Gap	Gap
2013-07	No Gap	Gap
2013-08	No Gap	Gap
2013-09	No Gap	No Gap
2013-10	No Gap	Gap
2013-11	No Gap	No Gap

No Gap	No Gap
No Gap	No Gap
No Gap	Gap
Gap	Gap
Gap	Gap
No Gap	Gap
No Gap	Gap
Gap	Gap
Gap	Gap
No Gap	Gap
No Gap	Gap

In Figure 15 we have two important phases of the pv's[13] package timeline. In the first screenshot we have the time when the second language gets introduced in the package. In the second screenshot we can clearly see that the introduced language is clearly abandoned before the end of the timeline.

Figure 16:



2018-12	No Gap	Gap
2019-01	No Gap	Gap
2019-02	No Gap	Gap
2019-03	No Gap	Gap
2019-04	No Gap	No Gap
2019-05	No Gap	No Gap
2019-06	No Gap	No Gap
2019-07	No Gap	No Gap
2019-08	No Gap	No Gap

2022-11	No Gap	No Gap
2022-12	No Gap	No Gap
2023-01	No Gap	Gap
2023-02	No Gap	Gap
2023-03	No Gap	No Gap
2023-04	No Gap	Gap
2023-05	No Gap	Gap
2023-06	No Gap	No Gap
2023-07	No Gap	Gap
2023-08	No Gap	Gap
2023-09	No Gap	Gap
2023-10	No Gap	Gap
2023-11	No Gap	Gap

Similarly for this example shown in Figure 12, which is visualizing a key part of the deck.gl's [14]package timeline, we can observe the introduction of the second language in April of 2019, and its complete abandonment in July of 2023.

Characteristics of Attempt Failure

The key attributes of this pattern include:

1) **Introduction of the new language:** The project witnesses the entry of a second language at some point after the 25% during its timeline.

[13] https://github.com/biasmv/pv
[14] https://github.com/uber/deck.gl

2) **Subsequent Abandonment:** Despite the initial adoption, the second language fails to maintain its presence or relevance throughout the project's lifecycle.

Detection and Verification Process

The detection process for Attempt Failure mirrors that of Attempt Success, utilizing the same code to identify the packages that classify as Attempt Success. The difference between the two processes is that the code I used for Pattern 5 does not check if the language got abandoned in the end, so I manually inspect each package's timeline to classify a certain package as Pattern 6.

In summary, Pattern 6 highlights the dynamic nature of language adoption and the subsequent uncertainties involved in incorporating new technologies into software projects.

5.7 Pattern 7: Unclear Pattern.

Pattern 7, termed "Unclear Pattern", classifies repositories where the discrepancy percentage between programming languages falls within a range that does not distinctly align with any predefined or new pattern. This pattern suggests a lack of notable trends or consistent language usage dynamics, making it challenging or impossible to categorize the repository into a specific pattern category. Below we can see some examples:

Figure 17:

Month	Java	Javascript
2017-12	No Gap	No Gap
2018-01	No Gap	Gap
2018-02	Gap	Gap
2018-03	Gap	Gap
2018-04	Gap	Gap

Figure 17 is poseidon's [15] package whole timeline. As we can observe from the visualization of its timeline there is only different activity for 1 of the total 5 months of the package, indicating there is not a clear pattern or a notable trend.

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

Figure 18:

Month	Ruby	Javascript
2016-08	No Gap	No Gap
2016-09	Gap	Gap
2016-10	Gap	Gap
2016-11	Gap	Gap
2016-12	Gap	Gap
2017-01	Gap	Gap
2017-02	Gap	Gap
2017-03	Gap	Gap
2017-04	No Gap	No Gap
2017-05	No Gap	Gap

In Figure 18 we can see the whole happybara's[16] timeline. By observing the timeline, we can see that there is only difference in the commit activity for May of 2017, resulting for 10% discrepancy percentage and therefore classifying as Pattern 7.

Characteristics of Unclear Pattern:

The main features defining this pattern include:

1) **Discrepancy Percentage Range:** Repositories classified under Pattern 7 exhibit a discrepancy percentage falling within a specific range, typically between 5% and 25%. This range indicates a moderate level of variance between the usage of different programming languages within the repository.
2) **Absence of Clear Trends:** Despite the small presence of variance in language usage, repositories classified under Pattern 7 lack clear, identifiable patterns or trends in their language usage behavior.
3) **Uncertainty in Classification:** Pattern 7 reflects the uncertainty surrounding the underlying language usage dynamics of the repository. While other patterns showcase distinct characteristics and behaviors, repositories categorized under Pattern 7 lack straightforward classification due to their mixed or inconclusive language usage patterns.

[16] https://github.com/amireh/happybara

Classification Process

The classification of repositories into Pattern 7 involves the application of specific criteria to identify instances where the discrepancy percentage falls within the defined range (5% to 25%) and where no other pattern classification has been assigned.

In summary, Pattern 7 represents a big category of repositories with unclear language usage dynamics that lack straightforward characteristics in order to classify them into a predefined or new pattern.

5.8 Research Questions.

1) **Question: What is the total number of repositories that passed the Language Consistency verification?**

   Answer: Although I collected data for 3987 repositories using the above detailed methodology, only 523 passed the Language verification process (more details about it in its own sub-section in the Methodology chapter) and were able for a further analysis.

2) **Question: What is the distribution of repositories across different patterns of language usage within cross-ecosystem packages?**

   Answer: The analysis that I did reveals a varied distribution of repositories across the patterns that we discussed in the previous sub-sections. Among the identified patterns, Pattern 7, characterized by unclear or inconclusive language usage trends, emerges as the one with the most packages with 214 repositories. Following, Pattern 2, exhibiting parallel support in both languages, shows a notable presence with 78 repositories. Additionally, 73 repositories are classified under Pattern1. Pattern 5, representing Attempt Success, is observed in 17 repositories, while Pattern 4 in 13 repositories. Furthermore, Pattern 3, showcasing interchanging support in both languages is found in only 7 repositories while Pattern 6 is identified in 5 repositories. Notably, 116 repositories remain unassigned to any specific pattern category. In total, the

analysis assigns patterns to 407 repositories, providing valuable insights into the diverse language usage within cross-ecosystem packages.

Chapter 6

Discussion

6.1 Pattern discovery relies on the dataset.

The patterns identified in this thesis regarding language usage within repositories are derived empirically from the dataset I worked with. It's crucial to acknowledge that the number and nature of these patterns may differ with different datasets. Several factors contribute to this variability.

Firstly, throughout the study I collected data for the cross-ecosystem packages of five ecosystems: Maven, NPM, PyPi, Rubygems and CRAN. If another composition of ecosystems is used for the research, then the results may differ, because of different repositories exhibiting distinct development patterns. Another big factor is the manual inspection and the thresholds that I chose. Different thresholds may produce different patterns.

Overall, while the patterns identified empirically in my thesis provide valuable insights into language usage across cross-ecosystem packages, they represent only a subset of possible patterns within the broader software development landscape.

6.2 Pattern interpretation

- **Pattern 1: Base language with light support of the other language.**

This pattern likely emerges when a repository primarily uses one programming language for its core functionality while occasionally incorporating features or modules written in another language for supplementary purposes. The significant discrepancy percentage between the two languages indicates that one language dominates the development efforts. The sporadic activity in the second language suggests that its role is providing specialized functionality or addressing specific requirements that are suited to that language's strengths. Reasons for this pattern could include exploiting language-specific optimizations, or integrating external systems that are more compatible with the secondary language.

- **Pattern 2: Parallel support in both languages.**

In this pattern, both languages exhibit similar levels of commit activity throughout the repository's timeline, indicating a balanced and parallel development approach. This pattern underscores the capability of development teams to manage multiple languages efficiently, ensuring that no part of the project falls behind due to unequal attention or resource distribution. The presence of this pattern within a repository can often indicate a well-integrated project structure, where cross-functional teams work together to advance the project's goals without bias toward one technology stack over another.

- **Pattern 3: Interchanging support in both languages.**

The interchanging support pattern suggests a fluidity in language usage within the repository, where neither language emerges as the dominant one. Repositories exhibiting this pattern may demonstrate a project structure that accommodates and perhaps benefits from the strengths of different programming languages.

- **Pattern 4: Language Migration.**

Language Migration is a critical pattern to recognize as it can significantly impact project development and maintenance. It reflects deeper changes within a project's lifecycle, such as shifts in technology preference, adaptation to market trends, or responses to

community feedback. Identifying this pattern may provide useful information into strategic decision-making processes regarding technology use within organizations.

- **Pattern 5: Attempt Success.**

The attempt success pattern showcases an effort to introduce a new language into the repository, with the intention of integrating it into the project's development workflow successfully. This pattern may occur when developers recognize the benefits of introducing and using a new language to address specific challenges, leverage unique features, or enhance the project's capabilities. Reasons for this pattern could include the need for language-specific tools or libraries, the desire to exploit the performance or efficiency gains offered by the new language, or the intention to attract contributors with expertise in the targeted language.

- **Pattern 6: Attempt failure.**

Contrary to the attempt success pattern, attempt failure indicates an unsuccessful attempt to introduce a new language into the repository, as evidenced by the language's abandonment before the end of the timeline. This pattern may result from factors such as technical challenges or limitations encountered during the integration process, insufficient community support or expertise in the new language or differences between project requirements and the capabilities of the chosen language.

6.3 Comparing pattern characteristics

The observed differences in average timeline duration, number of commits, and number of modified files among patterns may offer valuable insights into the nature, dynamics, and characteristics of software development projects:

Average Timeline Duration:

Pattern 1: 77 months

Pattern 2: 38 months

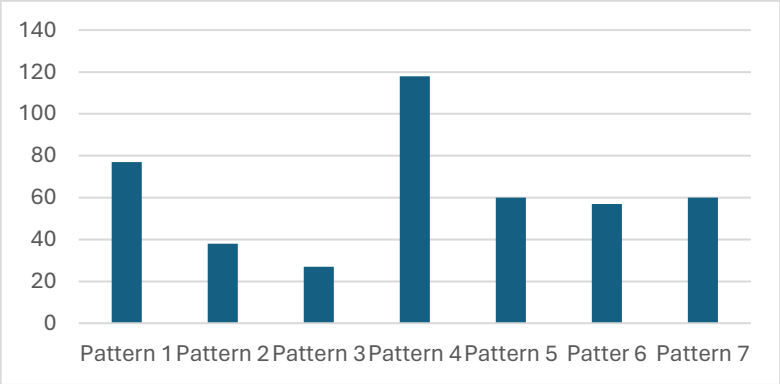
Pattern 3: 27 months

Pattern 4: 118 months

Pattern 5: 60 months

Pattern 6: 57 months

Pattern 7: 60 months



- Longer average timeline durations, as seen in repositories of Pattern 4 , may suggest that these projects have long-term development efforts. This long-term timeline of development might be the reason for the language migration of the package.
- Short average timeline durations, as observed in Pattern 3 repositories, could indicate projects with specific objectives or shorter development cycles.

Average Number of Commits:

Pattern 1: 2622 commits.

Pattern 2: 172 commits.

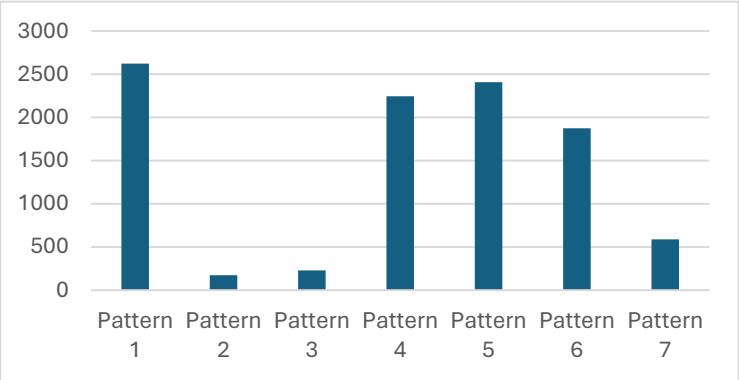
Pattern 3: 230 commits.

Pattern 4: 2244 commits.

Pattern 5: 2406 commits.

Pattern 6: 1874 commits.

Pattern 7: 589 commits.



- Higher average numbers of commits, as seen in Pattern1, 4 and 5 repositories, may happen because these projects are characterized by active collaboration, frequent updates and therefore commits, and continuous integration practices. These projects might prioritize agility and responsiveness to changes.
- Lower average numbers of commits, as observed in Pattern 2 and 3, might suggest projects with more conservative approaches. These projects may prioritize code stability and reliability over frequent changes and adjustments.

Average Number of Modified Files:

Pattern 1: 12953 modified files.

Pattern 2: 2242 modified files.

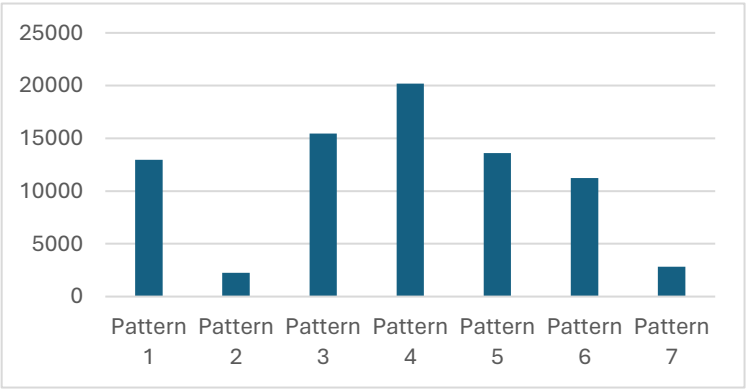
Pattern 3: 15464 modified files.

Pattern 4: 20195 modified files.

Pattern 5: 13599 modified files.

Pattern 6: 11247 modified files.

Pattern 7: 2844 modified files.



- Higher average numbers of modified files, as seen in Pattern 3 and 4, could indicate projects undergoing significant refactoring, architectural changes, or extensive feature expansions. These projects may experience ongoing evolution requiring modifying multiple modules and components.
- Lower average of modified files, as observed in Pattern 2 and 7, might suggest projects with narrower scope, simple architectures, or limited development activities.

6.4 Usefulness of the results.

The patterns analyzed in this thesis are important for several reasons. Firstly, they offer deep insights into how software projects evolve over time, highlighting trends in language usage, commit activity, and repository characteristics. These insights guide strategic decision-making by helping stakeholders identify successful projects and avoid unsuccessful ones. Moreover, the patterns enable comparative analysis, allowing developers to evaluate their techniques against others and identify areas for improvement. Additionally, the patterns serve as predictive indicators of future development trends, enabling proactive adaptation. Lastly, the patterns that I discovered or patterns in general may foster collaboration and communication among stakeholders, providing a common language for discussing project characteristics and aligning goals and expectations. Overall, identifying these patterns enhances project management, drives improvements in software engineering practices, and fosters innovation in the field.

After analyzing why these patterns are useful, the examination that emerges after that is to whom are these patterns useful. The answer is that these patterns hold utility for both developers and researchers in the software development domain. For developers, understanding these patterns helps them make informed decisions regarding project management. By recognizing the development trends and patterns, developers can better prepare for challenges and optimize their workflows. Additionally, for researchers these patterns can serve as a foundation for further empirical studies and theoretical investigations. Furthermore, researchers can leverage these patterns to develop predictive models, validate hypotheses, and contribute to the development of best practices and guidelines in the field.

6.5 Limitations.

While the identified patterns provide valuable insights into programming language usage across repositories, it's important to acknowledge the absence of direct validation from developers regarding the intentions behind language choices. Without direct confirmation from developers, the reasoning behind the language usages that lead to these patterns remains uncertain, and there’s a possibility that other factors, that I did not mention in the Discussion section, influenced the language usage.

Chapter 7

Conclusions and Future Work.

In this thesis, I managed a comprehensive analysis of cross-ecosystem software packages, focusing on identifying patterns in programming language usage. Through a multi-step methodology involving data collection, enrichment, and analysis, I uncovered seven distinct patterns, shedding light on the dynamics of language usage within these packages. These patterns ranged from base languages with light support to instances of language migration, providing valuable insights into the evolution and maintenance of cross-ecosystem packages.

By leveraging various metrics and manual verification processes, I not only identified these patterns but also provided context and possible explanations for their occurrence. This deeper understanding of language usage patterns in cross-ecosystem packages contributes to the broader discussion on software development practices.

In the future, there are several opportunities for extra research based on the findings of this thesis. Firstly, extending the analysis to include additional software ecosystems beyond the ones studied in this thesis could provide a more comprehensive understanding of language usage patterns across different domains. Exploring emerging ecosystems could uncover unique patterns and trends that were not captured in the current study. Also, engaging with developers to gain deeper insights into their decision-making processes regarding programming language selection and to validate the identified patterns against their actual intentions, may be a future research topic. This approach would offer a more comprehensive understanding of the dynamics driving language usage in software development projects. In addition, an interesting topic for future work would be how different characteristics of the package such as its age or the developers involved, affect the pattern of the pattern.

Βιβλιογραφία

(1) https://thesai.org/Downloads/Volume4No8/Paper_33-Software_Ecosystem_Features,_Benefits_and_Challenges.pdf

(2) Constantinou, E., Decan, A., & Mens, T. (2018). Breaking the borders: an investigation of cross-ecosystem software packages. arXiv preprint arXiv:1812.04868.

(3) M. Lungu, "Towards reverse engineering software ecosystems," *IEEE International Conference on Software Maintenance*, pp. 428-431, 2008, doi: 10.1109/ICSM.2008.4658096.

(4) https://www.browserstack.com/guide/what-is-maven-in-java

(5) https://datascientest.com/en/pypi-the-complete-guide-to-the-python-third-party-repository

(6) https://guides.rubygems.org/what-is-a-gem/

(7) https://www.pcmag.com/encyclopedia/term/cran

(8) https://onlinelibrary.wiley.com/doi/10.1002/smr.2270

(9) https://medium.com/@shikha.ritu17/understanding-node-js-ecosystem-and-tooling-fe7466d686c9

(10) Joshua, J.V., Alao, D.O., Okolie, S.O., & Awodele, O. (2013). Software Ecosystem: Features, Benefits and Challenges. *International Journal of Advanced Computer Science and Applications*

(11) Lungu, Mircea (2009). *Reverse Engineering Software Ecosystems* (Ph.D.). University of Lugano.

(12) Kannee, K., Kula, R. G., Wattanakriengkrai, S., & Matsumoto, K. (2023). Intertwining Communities: Exploring Libraries that Cross Software Ecosystems. In Proceedings of the 20th International Conference on Mining Software Repositories (MSR 2023). ACM.

(13) Jansen, S., Brinkkemper, S., & Finkelstein, A. (2009). Business Network Management as a Survival Strategy: A Tale of Two Software Ecosystems. In *First International Workshop on Software Ecosystems*.

(14) Fitzgerald, B., & Agerfalk, P. J. (2008). The mysteries of open source software: Black and white and red all over? In *Transactions on Software Engineering*.

(15) Bosch, J. (2009). From Software Product Lines to Software Ecosystems. In *Proceedings of the 13th International Software Product Line Conference*.

(16) Jansen, S., Cusumano, M. A., & Brinkkemper, S. (2013). Software Ecosystems: Analyzing and Managing Business Networks in the Software Industry. Edward Elgar Publishing.

(17) Manikas, K., & Hansen, K. M. (2013). Software Ecosystems – A Systematic Literature Review. In *Journal of Systems and Software*.

(18) Iansiti, M., & Levien, R. (2004). Strategy as Ecology. *Harvard Business Review*.

(19) Joshua, J.V., Alao, D.O., Okolie, S.O., & Awodele, O. (2013). Software Ecosystem: Features, Benefits and Challenges. *International Journal of Advanced Computer Science and Applications*

(20) Raemaekers, S., van Deursen, A., & Visser, J. (2012). Semantic versioning versus breaking changes: A study of the maven repository. *2012 28th IEEE International Conference on Software Maintenance (ICSM)*.

(21) Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM, 15*(12), 1053-1058.

(22) Raymond, E. S. (1999). *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly Media.

(23) Iansiti, M., & Levien, R. (2004). The Keystone Advantage: What the New Dynamics of Business Ecosystems Mean for Strategy, Innovation, and Sustainability. *Harvard Business School Press*.

(24) Jacobson, D., Woods, D., & Brail, G. (2011). *APIs: A Strategy Guide*. O'Reilly Media.

(25) Newman, S. (2015). *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media.

(26) https://cloc.sourceforge.net/