Empirical Software Engineering manuscript No.
(will be inserted by the editor)

How the Apache Community Upgrades Dependencies: An Evolutionary Study[⋆]

Gabriele Bavota · Gerardo Canfora · Massimiliano Di Penta · Rocco Oliveto · Sebastiano Panichella

the date of receipt and acceptance should be inserted later

Abstract Software ecosystems consist of multiple software projects, often interrelated by means of dependency relations. When one project undergoes changes, other projects may decide to upgrade their dependency. For example, a project could use a new version of a component from another project because the latter has been enhanced or subject to some bug-fixing activities. In this paper we study the evolution of dependencies between projects in the Java subset of the Apache ecosystem, consisting of 147 projects, for a period of 14 years, resulting in 1,964 releases. Specifically, we investigate (i) how dependencies between projects evolve over time when the ecosystem grows, (ii) what are the product and process factors that can likely trigger dependency upgrades, (iii) how developers discuss the needs and risks of such upgrades,

[⋆] This paper is an extended version of the paper: "Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, Sebastiano Panichella: The Evolution of Project Inter-dependencies in a Software Ecosystem: The Case of Apache. 2013 IEEE International Conference on Software Maintenance, ICSM 2013, Eindhoven, The Netherlands, September 22-28, 2013: 280-289, IEEE"

Gabriele Bavota
Dept of Engineering, University of Sannio, Benevento (BN), Italy
E-mail: gbavota@unisannio.it

Gerardo Canfora
Dept of Engineering, University of Sannio, Benevento (BN), Italy
E-mail: gerardo.canfora@gmail.com

Massimiliano Di Penta
Dept of Engineering, University of Sannio, Benevento (BN), Italy
E-mail: dipenta@unisannio.it

Rocco Oliveto
Dept. of Bioscience and Territory, University of Molise, Pesche (IS), Italy
E-mail: rocco.oliveto@unimol.it

Sebastiano Panichella
Dept of Engineering, University of Sannio, Benevento (BN), Italy
E-mail: spanichella@unisannio.it

and (iv) what is the likely impact of upgrades on client projects. The study results—qualitatively confirmed by observations made by analyzing the developers' discussion—indicate that when a new release of a project is issued, it triggers an upgrade when the new release includes major changes (e.g., new features/services) as well as large amount of bug fixes. Instead, developers are reluctant to perform an upgrade when some APIs are removed. The impact of upgrades is generally low, unless it is related to frameworks/libraries used in crosscutting concerns. Results of this study can support the understanding of the of library/component upgrade phenomenon, and provide the basis for a new family of recommenders aimed at supporting developers in the complex (and risky) activity of managing library/component upgrade within their software projects.

Keywords Software Ecosystems · Project dependency upgrades · Mining software repositories

1 Introduction

Software development is a collaboration-based activity. The highest grade of such a collaboration can perhaps be achieved when a software company decides to make available their product line architecture and shared components to external parties. Making available their own product outside the organizational boundary generates the so-called software ecosystems [6, 26]. In other words, a software ecosystem is a group of software projects that are developed and co-evolve in the same environment. These projects share source code, depend on one another, and can be built on similar technologies. In some cases, they have a closed core that provides the basic functionality, and a set of components that provide specific functionality. For example, the Eclipse project provides the core functionality of an IDE, that can be customized into any kind of IDE or editor though a specific plug-in. In other cases, they can be completely different projects sharing a set of common components.

Software ecosystems are therefore a new dimension of collaboration, that allows companies to satisfy the need of their customers as rapidly as possible and facilitate mass customization [6]. Thus, in recent years it is possible to observe an increasing trend of software companies that are moving from product lines towards software ecosystems to better support the intra-organizational reuse of software. Such a transition recalls the need of methods and tools to effectively manage both the coordination and the evolution of software ecosystems.

A crucial activity for an effective evolution of a software ecosystem is managing the upgrades of libraries/components. When one project undergoes changes and issues a new release, this may or may not lead other projects to upgrade their dependencies. On the one hand, using up-to-date releases of libraries/components may result useful, because these releases can contain new and useful features, and/or possibly some faults may have been fixed. On the other hand, the upgrade of a component may create a series of issues.

For example, some APIs may have changed their interface, or might even be deprecated [37], which requires the adaptation of its client.

In addition, let us suppose a program uses multiple libraries, namely lib_1 and lib_2, and lib_1 depends on lib_2. It can happen that if one upgrades lib_2, then lib_1 no longer works because does not support the new release of lib_2. Last, but not least, a library/component might have changed its license making it legally incompatible with the program using it [12]. All these scenarios suggest that updating a library/component in large ecosystems is a complex and daunting task, which requires to ponder several factors.

In principle, the problem can be dealt with update management tools available in many operating systems—e.g., Windows, Linux, MacOS—however such update tools either work with entire applications or with operating system related upgrades. Also, they are not able to decide when performing the upgrade and when it might be avoided or postponed.

Following the recent trend of studies aimed at analyzing the evolution of software ecosystems [7,15,37], we present an exploratory study conducted on the Java subset of the Apache ecosystem focusing the attention on how and why dependencies (i.e., dependencies related to API usage and/or framework usage through extension) between software projects evolve. The entire Apache ecosystem is composed of 195 software projects developed by using a total of 29 programming languages. We analyzed the change history of the 147 Java software systems, in the period of time going from June 1999 to April 2013 resulting in 1,964 releases.

In our study we analyzed how the number of projects, their size, the dependencies among them, the declared software licenses, and the number of active developers changed in the ecosystem during time. After this preliminary analysis, we tried to analyze which are the factors driving a project ("client project") to upgrade (or not) a dependency with another project ("library project") when a new release of the latter is issued. Several factors could influence such a choice. We focus our attention on (i) project size; (ii) structural changes (e.g., changes involving the interfaces) and number of bug fixed in the new release of the library project; (iii) nature of the release (i.e., minor, major, or bug fixing); (iv) licensing changes; and (v) developers' overlap between the client project and the libraries being used by such a project. It is worth noting that some of these factors cannot be directly observed from source code changes only, but they require the analysis of other source of information, such as release notes, or developers' discussion. For this reason, after performing a quantitative analysis of the phenomenon of library/component upgrade, we analyzed mailing lists and issue tracking systems in order to understand to what extent developers discuss the management of dependencies and what are the factors subject of the discussion. Such an investigation required the manual analysis of 7,685 discussions and allowed us to provide some qualitative insights on the evolution of dependencies in the Apache ecosystem. Finally, we also investigated the impact of upgrades on the source code of the client project.

The obtained results indicate that when client projects upgrade their dependencies, they do not necessarily perform all available upgrades. Specifically, a client project tends to upgrade a dependency only when substantial changes in the library project are released, including bug-fixing activities. These observations are also confirmed by the analysis of communications between developers. In addition, we observed that pairs of projects having a dependency share a higher number of developers as compared to pairs of projects do not having a dependency. However, such an overlap does not influence the client's upgrade frequency. Finally, our results indicate that while the proportion of source code of client projects impacted by changes in the projects they depend on is, on average, quite limited (5%), there are specific dependencies, generally toward frameworks/libraries offering very wide services (e.g., parallel computation), that could strongly impact (up to 62%) the client project source code when a dependency is upgraded.

The study reported in this paper provides, for the first time, insights on the complex and crucial activity of library/component upgrade in large software ecosystems. Specifically, the study results provide evidence about the relevance of the problem, and that specialized methods and tools might be high beneficial for developers that have to decide whether or not upgrade a dependency based on the benefits and the side effects of such an upgrade. Thus, the study presented in this paper poses the basis for a new category of recommender systems, that could be used in software ecosystems to effectively manage the dependencies between projects.

The paper is organized as follows. Section 2 describes the study definition and planning, while results are reported in Section 3. Section 4 discusses the threats that could affect the validity of the results achieved. Section 5 presents the existing literature about the evolution of software ecosystems and evolution/adaptation of APIs. Finally, Section 6 concludes the paper and outlines directions for future work.

2 Study Definition and Planning

The *goal* of this study is to analyze how project inter-dependencies are upgraded in a software ecosystem, with the *purpose* of understanding (i) how dependencies are maintained, and (ii) the likely reasons and consequences of upgrades. The *quality focus* is software maintainability, which could be improved by understanding the phenomenon of library/component upgrade. The *perspective* is of researchers interested in understanding when and why developers upgrade dependencies in software ecosystems.

The *context* of the study consists of the entire history of the Java subset of the Apache ecosystem, that represents the vast majority of it (75% of the projects). To date, the entire Apache ecosystem is composed of 195 software projects spread over 23 different categories (e.g., big-data, FTP, mobile, library, testing, XML) and developed by using a total of 29 programming languages. We analyzed the change history of the 147 Java software systems, in the period of time going from June 1999 to April 2013 resulting in 1,964 releases. The size of the ecosystem in the analyzed period of time ranges from 32 up to 28,584 KLOCs, while the number of classes (methods) ranges from 113 to 114,000 (1,386 to 780,731). For sake of clarity, in the following we refer to the project having a dependency toward another project as the "client project" and to the project used by a "client project" as the "library project".

2.1 Research Questions

The study aims at providing answers for the following four research questions:

– $\mathbf{RQ}_1$: *How does the Apache ecosystem evolve?* This research question is preliminary to the other three, and aims at providing a context for our study. Specifically, we analyze how the number of projects, their size, the dependencies among them, the declared software licenses, and the number of active developers changed in the Apache ecosystem during time. Such information represents the foundation for the other research questions.
– $\mathbf{RQ}_2$: *What are the reasons driving a client project to upgrade a dependency toward a new available release of a project it depends on in the Apache ecosystem?* Our conjecture is that the client project does not always upgrade a library when a new release of such a library has been issued but, rather, this is done based on the benefits the upgrade would provide, and the impact such an upgrade can have on the system. Other than verifying this conjecture, we are also interested to understand what are the factors driving a client project to upgrade (or not) a library. Clearly, such upgrades can be driven by many different factors, some of which cannot be directly observed from software repositories. In our study, we choose to focus on the following factors:
  – *Project characteristics:* we analyze if certain characteristics of the clients' projects, namely the project size (in terms of # of classes and LOC) and the number of dependencies the project has towards other projects, impact their proneness to upgrade their dependencies when new releases of the libraries are available. Our conjecture is that larger systems as well as those having several dependencies, could adopt a more systematic approach in managing their dependencies as compared to small projects almost "living" on their own.
  – *Structural changes*, captured by analyzing changes in source code of the library project. The conjecture is that changes involving the library interfaces will likely trigger more upgrades than other changes.
  – *Number of bugs fixed:* we compute the number of bugs fixed between the previous library release and the current one, to determine whether a higher number of fixed bugs correlates with the likelihood of release upgrade. We focus on such factor since we expect that releases including a high number of fixed bugs triggers more upgrades with respect to releases including few or no fixes.

  - *Nature of the release,* captured by manually analyzing release notes. We are interested to investigate whether the release was due to a substantial addition of new features or if, instead, it was mainly due to bug fixes or else to minor improvements. We expect to observe a positive correlation between the number of new features present in a release and the number of upgrades it triggers. Indeed, a release including several new features is likely to be more attractive for client projects.
  - *Licensing changes*, occurring in the declared licenses, that might result in legal incompatibilities between the client and the library project. A licensing change could discourage client projects to upgrade toward a new release of the library project in order to avoid license compliance problems.
  - *Developers' overlap* between the client and the library project. The conjecture here is that projects with a non-negligible overlap of developers with the used library will perform updates more frequently than projects having a smaller (or no) overlap.
- **RQ**$_3$: *How are dependencies discussed between open source developers?* This research question aims at understanding to what extent is the management of dependencies between a client and a library discussed by developers over mailing lists and issue trackers, analyzing the factors object of the discussion and the developers involved.
- **RQ**$_4$: *If a dependency is upgraded, to what extent would this impact on the client project source code?* This research question aims at quantitatively investigating the impact on the source code of the client project when it upgrades a project it uses towards a new available release.

2.2 Data Extraction and Analysis

To answer our research questions we use a crawler and a code analyzer developed in the context of the MARKOS European project[1] The crawler is able to identify for a given project of interest the list of available releases with their release date as well as its SVN address. This information is extracted by crawling DOAP (Description Of A Project) files available on the Internet[2]. Each DOAP file provides information about a specific open source project and it is released by the project itself. Thus, the information contained in DOAP files are quite reliable and include (but are not limited to): the project name, a brief and a long project description, the versioning system address, and a list of the public releases issued by the project.

Using the information extracted by the crawler, the code analyzer checks-out files from the SVN repository and identifies the folder containing each of the project releases identified by the crawler. This is done by exploiting the SVN tag mechanism. In other words, the versioning system of Apache projects

[1] `http://www.markosproject.eu` [5] in order to download the source code of the 1,964 software releases considered in our study.

[2] `http://projects.apache.org/doap.html`

has a separate directory for each release (where files belonging to such a release are stored), besides keeping the project history in the SVN main trunk. In case the code analyzer does not identify any folder containing a particular release, it reports the problem. This issue occurred for 278 releases (across all projects). In order to consider also these releases in our study, we manually downloaded them from the Apache release archives, available online for each project[3].

Once downloaded all the software releases, we *extract dependencies* existing between such releases. Note that in this study we focus on dependencies existing between Java Apache projects, ignoring those toward projects external to the Apache ecosystem or not written in Java. Also in this case, the MARKOS code analyzer has been used. The identification of the inter-project dependencies is performed in different steps. Given a set of software releases, the code analyzer searches—in each folder release—for files that explicitly report inter-project dependencies. By looking into the Apache repositories, we found these files to generally be of three types: `libraries.properties`, `deps.properties`, or the Maven `pom.xml` files. Indeed, 1,231 of the 1,964 releases object of our study (63%) contain at least one of these three files in their root folder. Note that the dependency information reported in these files is generally detailed (i.e., both the name of the project as well as the used release are reported) and reliable.

When the code analyzer is not able to find any of these files (in 37% of the releases), it searches for all jar files contained in the release folder and tries to match each of those files with one of the other software releases provided. This is done by computing the Levenshtein distance [32] between the name of the jar archive and the name of each provided release. The output of the code analyzer is a list of candidate dependencies between the set of provided software releases.

In our study, we assume that the dependencies extracted by parsing the files `libraries.properties`, `deps.properties`, and `pom.xml` are correct. Instead, when the dependencies are extracted by analyzing jar files in the release folder, we manually validate and classify them as *true dependencies* or as *false positives*. This operation was done by two of the authors that analyzed a total of 3,742 dependencies, classifying 832 correct dependencies. Overall, the final number of dependencies found in the analyzed 14 years of observation and considered in our study is 3,514 (i.e., 2,682 extracted from dependencies files, plus the 832 manually verified).

To answer $\mathbf{RQ}_1$ and $\mathbf{RQ}_2$ we also identify the software licenses declared in the downloaded software releases. To this aim we use Ninka[4] [18], a lightweight license identification tool for source code that consists on a sentence-based matching algorithm that automatically identifies license from *license statements*. We run Ninka on each file contained in the 1,964 software releases

[3] An example of archive for the Ant project can be found here `http://archive.apache.org/dist/ant/source`

[4] `http://ninka.turingmachine.org/`

considered in our study, obtaining as output the license type and version declared in its licensing statement (if present).

After having performed this first data analysis (necessary for all research questions), we perform analyses specific to each research question, explained in the following.

2.2.1 RQ_1 analyses

To answer $\mathbf{RQ}_1$ we analyze the history of the Apache ecosystem, considering snapshots captured every month. In particular, starting from June 1999, we compute, with a granularity of one month (which we consider sufficient to observe the evolution of the ecosystem over several years):

1. the number of existing projects;
2. the size of the ecosystem in terms of KLOCs;
3. the dependencies existing between projects; and
4. the software licenses declared in source files.

To analyze how the number of developers changed during time ($\mathbf{RQ}_1$), we extract the list of *active developers* that worked in the ecosystem during its entire history[5]. In particular, from our starting date (i.e., June 1999), we compute the number of active developers at time intervals of six months (e.g., from June 1999 to January 2000). We consider a developer active in the time interval of interest if she performed at least one commit in one of the Apache projects existing at date. Note that, while we consider a granularity of one month for most of the measures, we check the activity of developers for a period of six months, because the lack of activity for a short period (i.e., one month) can just occur by chance. Indeed, developers engaged in open source development are often volunteers just dedicating part of their free time to development activities. Also, checking whether a developer is active in a given time period does not mean determining whether a developer has left a project or not. In other words, a developer may not be active in a given time period, but she can still be part of the project and likely contribute in the future. On the other side, computing the other measures (i.e., number of projects, KLOC of the ecosystem, number of existing dependencies, and declared software licenses) at time intervals of one month is safe (i.e., there is no possible misinterpretation behind these measures) and will provide us with a finer-grained view of the evolution of the Apache ecosystem.

2.2.2 RQ_2 analyses

Some of the data needed to answer $\mathbf{RQ}_2$ (e.g., dependencies, licenses, developers working on the various projects) is already available after having performed the general data extraction and the $\mathbf{RQ}_1$-related data extraction as described

[5] Note that we limit our analysis to developers we can detect through their activities in the versioning system, as also pointed out in Section 4.

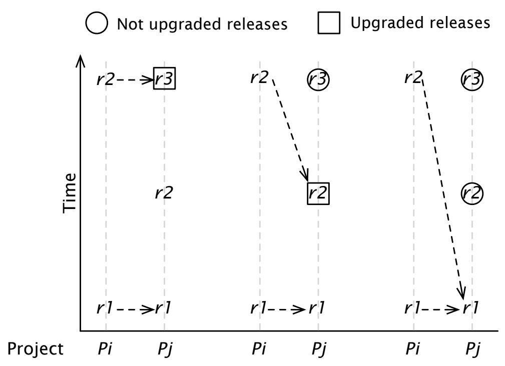


Fig. 1: Process used to divide *upgraded* and *not upgraded* releases.

above. Specifically, we describe how we measure the dependent and independent variables that concern the analyses of $\mathbf{RQ}_2$, and explain the kinds of statistical analyses we perform.

Dependent Variable: Upgrades. Given the dependencies existing between different project releases, we distinguish releases of the libraries that are upgraded by client projects (hereby referred as *upgraded releases*) and releases ignored by client projects (hereby referred as *not upgraded releases*).

To create the two sets of releases (i.e., *upgraded releases* and *not upgraded releases*) we adopt the process depicted in Fig. 1. For each pair of Apache projects, P_i and P_j, having at least one dependency between their releases, when P_i upgrades the dependency towards P_j, we determine whether P_i upgrades the dependency toward to the last existing release of P_j or to another release. In the former case, we put the upgraded P_j release in the set *upgraded releases*. Instead, when the upgrade was not toward the last available release we still put the upgraded P_j release in the set *upgraded releases*, however we also put the newer ignored releases of P_j in *not upgraded releases*.

To better understand how we compute such sets, Fig. 1 shows three different evolution scenarios of dependencies between two projects P_i and P_j. Let us assume that the release $r1$ of P_i depends on the release $r1$ of P_j. Then, a new version of project P_i is released ($r2$). In the first scenario, when $r2$ for P_i is released, its dependency is upgraded to $r3$ of P_j, the last available P_j release. In this case, $r3$ is included in the set *upgraded releases*, while no releases are added to the set *not upgraded releases*, since P_i correctly upgraded its dependency to the last available P_j release. In the second scenario (reported in the middle of Fig. 1), the release $r2$ of P_i upgrades its dependency to the release $r2$ of P_j, even if a newer release (i.e., $r3$) is available. In this case the release $r3$ of P_j has been "ignored" by P_i and thus, it is added to the set *not upgraded releases*, while release $r2$ of P_j is added to the set *upgraded releases*. In the third and last case, P_i does not upgrade at all the dependencies toward P_j, i.e., the new release of P_i continues to use the release $r1$ of P_j, despite the availability of more recent releases (i.e., $r2$ and $r3$). In this case, $r2$ and $r3$ are added to the set *not upgraded releases*, while no releases are added to the set *upgraded releases*.

Note that, if a release r_i of a project P_j belongs to the set of *upgraded releases* when analyzing dependencies between P_i and P_j, and the same release belongs to the set of *not upgraded releases* when analyzing dependencies between a project P_s and P_j, the release r_i is removed from both sets, and not considered any longer in the comparison between *upgraded releases* and *not upgraded releases*. This is done (i) to avoid overlap between the two sets (which would make the comparison unfair); and (ii) to strongly isolate only releases that are generally upgraded (and not) by client projects. As it will be clearer later on, 140 releases were classified as *upgraded* or as *not upgraded* and of these, 14 have not been assigned to one of the two sets since upgraded by some client projects and not upgraded by other clients. Thus, only 10% of upgraded/not upgraded releases were discarded from our analysis, limiting the impact on results of our choice of excluding each possible overlap between the two sets (i.e., *upgraded* vs *not upgraded*).

Independent Variable - Project characteristics. To verify if project characteristics impact the upgrade frequency of client projects, we extract the following information from each client:

1. *size*, in terms of Lines Of Code (LOC), Number of classes (classes);
2. *fan-out*, i.e., the number of dependencies it has toward other projects; and
3. *fan-in*, i.e., the number of projects using it. Note that a client project can be a library for other projects.

The update frequency (Up_f) of a client project C issuing its release r_i is computed as:

$$Up_f(C_{r_i}) = \frac{\#upgraded\_dependencies}{\#available\_upgrades}\%$$

where *#upgraded_dependencies* is the number of dependencies upgraded in r_i and *#available_upgrades* is the total number of upgrades available for C when r_i is released.

Then, we compute the Spearman rank correlation [45] between the measured system characteristics and the upgrade frequency of client projects. Cohen et al. [8] provided a set of guidelines for the interpretation of the correlation coefficient. It is assumed that there is no correlation when $0 \leq \rho < 0.1$, small correlation when $0.1 \leq \rho < 0.3$, medium correlation when $0.3 \leq \rho < 0.5$, and strong correlation when $0.5 \leq \rho \leq 1$. Similar intervals also apply for negative correlations.

Independent Variable - Structural changes. Concerning the changes performed among two subsequent releases of each project, the MARKOS code analyzer parses the source code by relying on the *srcML* toolkit [9] and extracts a set of facts concerning the files that have been added, removed, and changed in each commit. Information about the commits performed between release r_{i-1} and r_i is extracted from the versioning system. Given the set of files involved in a commit, the following kinds of changes are identified:

– *File addition and removal.* These changes can be easily identified from the versioning system log.
– *Class addition and removal.* Added and removed classes are extracted by comparing the content of each source code file involved in the commit before and after the commit was performed. Note that for this purpose it is not enough to analyze just the files added and removed in a commit, since a new class could be added or removed from a previously existing source code file (i.e., a file modified in the commit). Also, it could happen that a class is deleted from a file F_i and is added in a file F_j (i.e., the class has been moved from file F_i to file F_j), or that a class has been renamed. To detect these cases, renaming or moving of classes between files are identified through a fingerprinting-based approach. That is, a class is characterized by a set of metrics (forming a fingerprint), and such a fingerprint is used to trace classes in case of renaming or moving. We evaluated this fingerprint-based approach in the context of the MARKOS European project on a set of 200 renamed/moved classes (of which 100 renamed and 100 moved) obtaining an accuracy of 93% (i.e., 186 of the renamed/moved classes where identified by the code analyzer, which classified the remaining 14 as newly added classes).
– *Method addition and removal.* As in the previous case, added and removed methods are identified by comparing the content of each source code file before and after the commit under analysis. Moving or renaming methods is detected similarly to class renaming/moving (i.e., using a fingerprinting-based approach).
– *Method changes.* For each method contained in the files modified in the commit under analysis, our tool compares its version before and after the commit. Through this comparison, it is able to identify: (i) changes to the method visibility (e.g., a method converted from public to private); (ii) changes to the method signature (e.g., parameters added/removed, changes to the exceptions thrown by the method); and (iii) changes performed to the method body. Note that we can distinguish between changes performed to public and non public methods.

Independent Variable - Number of fixed bugs. We identify the bugs fixed in each software release by mining the bug-tracking systems of the various projects, extracting only the bugs fixed in each specific release. In particular, when identifying bugs fixed on release r_i, we search into the issue

Table 1: Tags assigned to classify the release notes.

Tag	Applied when	Constraints
minor	The new release only includes improvements of existing features	[if !*major*]
major	The new release includes new features	[if !*minor*]
bug fixing	The new release includes fixed bugs	-

tracker for issues with Type=*"Bug"* (Jira) or Severity=*"Defect"* (Bugzilla), Status=*"Resolved"* or *"Closed"*, and Resolution=*"Fixed"*, with a resolution date included in the $[t_{r_{i-1}}, t_{r_i}]$ period, where t_{r_i} is the release date of r_i.

Besides comparing descriptive statistics, we also use the Mann-Whitney test [10] to compare the distribution of changes and bug-fixing for the above described two sets of releases (information extracted through the process described in Section 2.2). We assume a significance level of $\alpha = 5\%$. We also estimate the magnitude of the difference between the number of changes for the two considered groups of releases (upgraded and not upgraded by clients) using the Cliff's Delta (or d), a non-parametric effect size measure [23] for ordinal data. We follow the guidelines of Cliff [23] to interpret the effect size values: small for $0.148 \leq d < 0.33$ (positive as well as negative values), medium for $0.33 \leq d < 0.474$ and large for $d \geq 0.474$.

Independent Variable - Nature of the release. We manually analyze the release notes, and classify them using the tags reported in Table 1. In particular, Table 1 shows for each tag its name (e.g., *major*), explaining when it has been applied (e.g., the new release includes new features), and exclusion constraints between different tags (e.g., the tag *major* excludes the tag *minor*). Note that the tag *bug fixing* is orthogonal to *minor/major*, and can be assigned to any release note talking about fixed bugs, despite it underwent minor or major changes (this is why it does not have constraints in Table 1). This classification has been performed by two of the authors who individually analyzed and tagged the release notes. Then, they performed an open discussion to resolve any conflicts and reach a consensus on the assigned tags. Descriptive statistics of the tags assigned to the release notes are discussed in the paper. Note that such variable is not independent from the number of fixed bugs as well as from structural changes. Indeed, we found *major* releases to contain a higher number of fixed bugs and, in general, of changes, as compared to *minor* releases. For instance, the median number of bugs fixed in *major* releases is 35 as compared to the 15 fixed in *minor* releases.

Independent Variable - Licensing changes. We perform this analysis using data extracted as explained in $\mathbf{RQ}_1$. We determine if upgrades or, possibly, replacements of a library with another, occur when the library or the client change the license in a way to create an incompatibility.

Table 2: Tags assigned to classify the mailing lists discussions.

Tag	Applied when	Constraints
	GENERAL TAGS	
client	The discussion is in the client mailing list	[if !*library*]
library	The discussion is in the library mailing list	[if !*client*]
dependency	The discussion focuses on the dependency between the client and the library	-
	DEVELOPERS TAGS	
only developers client	only developers of the client project take part to the discussion	[if *dependency* && !*only developers library* && !*both developers*]
only developers library	only developers of the library project take part to the discussion	[if *dependency* && !*only developers client* && !*both developers*]
both developers	both developers of the client and of the library projects take part to the discussion	[if *dependency* && !*only developers library* && !*only developers client*]
	TOPIC TAGS	
break	the discussion is about avoiding changes that could break the dependency	[if *dependency* && !*fix* && !*upgrade* && !*use* && !*other*]
fix	the discussion is about changes needed to fix a dependency	[if *dependency* && !*break* && !*upgrade* && !*use* && !*other*]
upgrade	the discussion is focused on whether upgrading/not upgrading a dependencies toward a new available release of the library project	[if *dependency* && *client* && !*fix* && !*break* && !*use* && !*other*]
use	the discussion is about how to use the library project	[if *dependency* && *client* && !*fix* && !*upgrade* && !*break* && !*other*]
other	the discussion is about the dependency, but cannot be classified with any of the previous tags	[if *dependency* && !*break* && !*fix* && !*upgrade* && !*use*]

Independent Variable - Developers' overlap. We analyze the overlap in terms of active developers (already detected to answer $\mathbf{RQ}_1$) between all pairs of projects existing in the Apache ecosystem. Given two projects C and L, the developers' overlap (in percentage) between them is computed as:

$$overlap_{C,L} : \frac{|D_C \cap D_L|}{|D_C \cup D_L|}$$

where T_C are the developers of project C and T_L are the developers of project L. With this analysis we want to understand if (i) pairs of projects *having* a dependency share more/less developers than pairs of projects *do not having* a dependency and (ii) client projects having a high overlap of developers with the libraries they use have a higher upgrade frequency (still by using the Spearman correlation).

2.2.3 RQ_3 analyses

Concerning $\mathbf{RQ}_3$, we downloaded [6] the Apache mailing lists and the discussions on the Apache issue trackers for the projects object of our study showing at least a dependency. This resulted in the download of 84 mailing lists and nine issue trackers. Note that the number of issue trackers downloaded is considerably lower than the number of mailing lists, due to the fact that most of the Apache projects use Jira[7] as issue tracker, which automatically forwards discussions to the projects' mailing list. Therefore, in such cases it was

[6] `http://mail-archives.apache.org/mod_mbox/`

[7] `https://issues.apache.org/jira/secure/Dashboard.jspa`

sufficient to limit our analyses to mailing list only. Instead, this is not the case for projects using Bugzilla[8]. Overall, we downloaded 664,490 discussions containing a total of 1,924,002 messages.

Then, for each pair of projects C, L exhibiting a dependency, we filter—from the project C mailing lists and issue tracker—all discussions containing (in the mail object/issue title or in the mail body/issue description) the name of project L. Similarly, we filter—from the project L mailing lists and issue tracker—all discussions containing (in the mail object/issue title or in the mail body/issue description) the name of project C. This resulted in 7,685 discussions that have been manually analyzed by two of the authors, and classified using the tags shown in Table 2.

The manual analysis has been performed as follows. Firstly, *general tags* are assigned to the discussion, classifying it as belonging to the *client* or to the *library* mailing list/issue tracker and, most importantly, verifying if the discussion is focused on the management of the dependency between the client and the library project (tag *dependency* in Table 2). Then, if the tag *dependency* has been assigned to the discussion, developers and topic tags are also associated to it. Developer tags aim at classifying the developers taking part to the discussion, while topic tags categorize the aim of the discussion (see Table 2). Developer tags have been automatically assigned by matching the email addresses used in the mailing lists/issue trackers with those used in the versioning systems of the project under study. We report descriptive statistics of the tags assigned to the analyzed discussions, and then discuss the most interesting cases. Note that results of the manual analysis performed in the context of this research question can also help in corroborating quantitative findings from $\mathbf{RQ}_2$.

2.2.4 RQ_4 analyses

To answer $\mathbf{RQ}_4$, we identify—using again the MARKOS code analyzer—the source code potentially impacted when an upgrade of a dependency is performed by a client project. The impacted source code is overestimated considering as candidate impact set all the classes of the client project importing at least one class of the upgraded project. We report descriptive statistics of the impacted client code in terms of percentage of impacted classes, and percentage of impacted LOCs.

2.3 Replication package

The study described in this section can be replicated using the replication package available online[9]. The replication package provides information to download all analyzed projects, and includes data sets used to answer the study research questions. In particular, we provide:

[8] `http://www.bugzilla.org/`

[9] `http://distat.unimol.it/reports/emse-apache/`

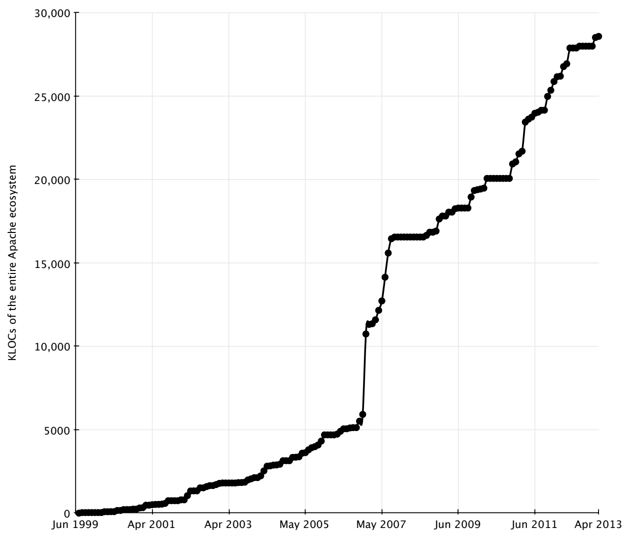


Fig. 2: Evolution of the size in the Apache ecosystem.

- raw data of the evolution during time of number of projects, dependencies between them, size, and number of developers of the Apache ecosystem;
- the history of dependencies between project releases,
- the changes captured through the MARKOS code analyzer for upgraded and not upgraded releases, and
- raw data of the manual tagging performed on the developers' discussions.

3 Analysis of the Results

This section discusses the study results, in order to answer the four research questions formulated in Section 2.1.

3.1 RQ1: How does the Apache ecosystem evolve?

Fig. 2, 3, and 4 report the evolution over time of the Java Apache ecosystem, in terms of size measured in KLOCs (see Fig. 2), number of projects (black line in Fig. 3), number of dependencies existing between them (gray line in Fig. 3), and number of active developers[10] (Fig. 4).

The results of model fitting suggest the exponential growth (adjusted $R^2 = 0.56$) of the Apache ecosystem during the analyzed 14 years (Fig. 2). From the single Java project existing in 1999 (i.e., Apache ECS[11]) the Apache

[10] Remember that we consider a developer "active" if she performed at least one commit in the previous six months.

[11] `http://projects.apache.org/projects/ecs.html`

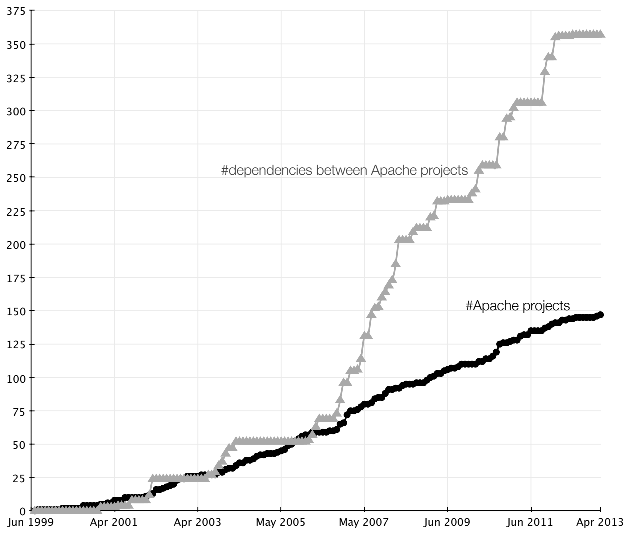


Fig. 3: Evolution of the projects and dependencies in the Apache ecosystem.

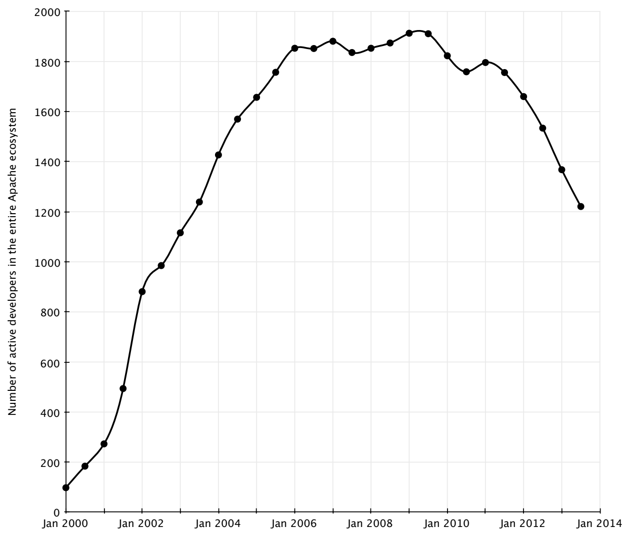


Fig. 4: Evolution of active developers in the Apache ecosystem.

ecosystem grows up to the 147 Java projects existing today. Such a growth is linear (adjusted $R^2 = 0.98$). With the number of projects also the size—see Fig. 2—of the entire ecosystem grows, by reaching almost 30 Million LOCs in April 2013. A very strong peak in the size of the ecosystem can be observed

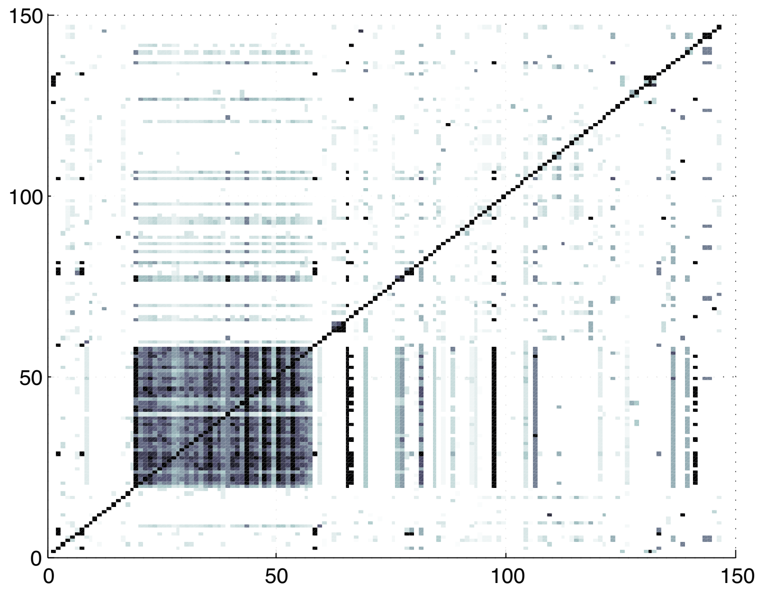


Fig. 5: Developers' overlap in the Apache ecosystem in 2013. The x- and y- axis represent the 147 projects object of our study reported in alphabetic order.

between the end of 2006 and the begin of 2007, when the Apache ecosystem doubled its size. In this period, several new, large project have been added to the ecosystem. Examples include Apache UIMA[12] with its two millions of LOCs and Apache Derby[13] with almost one million LOCs.

Fig. 4 shows that the number of active developers grows exponentially (adjusted $R^2 = 0.82$) until January 2006 together with the increase of the number of projects (Fig. 3) in the Apache ecosystem. In particular, in 2006 there were 58 Java projects carried out by almost 1,800 developers. Then, while the number of projects continued its linearly growth (see Fig. 3), the number active developers stopped its growth, and remained almost stable for four years at a level of 1,800 people. Then, from 2011 beyond we observed a decrease of the number of active developers. At the time of writing (November 2013) such a number is of around 1,200 people.

Given the continuous increase of the number of projects in the ecosystem, this result might appear counterintuitive. We found two possible interpretations for that. The first one is related to the developers' overlap existing between the Apache projects. As previously mentioned, the developers base in

[12] `http://uima.apache.org/`

[13] `http://db.apache.org/derby/`

2006 was very large (1,800 people), therefore when new projects were added to the ecosystem, it is likely that they were mostly carried out by developers already active on other (previously existing) projects. A relevant example is represented by the Apache commons projects, that are evolved and maintained by a very cohesive community of developers. The number of distinct developers working in 2013 on the Apache ecosystem is 147, against 2,674 developers counted by project (i.e., a developer is counted multiple times if she works on more than one project). This means that there is a strong overlap of developers between these projects. Thus, when new projects are added to the ecosystem, this does not necessarily imply that new developers also join the ecosystem.

To get a better view of the developers' overlap existing between the Apache projects, Fig. 5 shows, for each pair of the analyzed 147 Java projects, the percentage of developers overlap in 2013: black means 100% of overlap between the two projects, white means 0% of overlap. The diagonal is colored in black by default, since each project will have 100% of developers' overlap with itself. As we can see, several projects share developers, also in high percentage. The black rectangle that can be observed in the right-up part of Fig. 5 corresponds to the Apache commons projects.

While the overlap figure explains the stable number of developers between 2006 and 2011, it is still unclear why from 2001 to 2013 the developers base decreases, despite the increase of projects in the ecosystem. As it can be noticed from Fig. 3, on the one hand the number of projects does not have a substantial increase between 2011 and 2013 (12 projects added). On the other hand, in such a period the overall ecosystem LOC (Fig. 2) increased of about 18%. That is, changes occurred in the ecosystem mainly concerned addition/improvement of features in existing projects (as well as bug fixes). However, this kind of activity concerned a relatively limited number of developers, whereas many developers worked on the early development activity of each project. Also, until 2011, several projects have reached a very stable state, in which there is not a lot of activity. Therefore, we see a decrease in the number of active developers (which does not necessarily mean that developers abandon the project). For example, the last release of Apache Chainsaw[14] is dated March 2006, while the last release of Apache Commons Betwixt[15] is dated March 2008. Referring to the previous examples, the number of developers in Apache Chainsaw has decreased from eight in 2004 to one in 2013, while the number of developers in Apache Commons Betwixt from 19 to five in the same time period. Besides that, we noticed that also very active projects—i.e., still issuing releases during the last year—had a decrease of active developers due the reached mature state. For instance, in 2005 Apache Cocoon[16] had 64 active developers; nowadays the number of active developers is reduced to seven. Finally, it is worth noting that Goeminne *et al.*

[14] `http://logging.apache.org/chainsaw/`

[15] `http://commons.apache.org/proper/commons-betwixt/`

[16] `http://cocoon.apache.org/`

Table 3: Top Ten Most Used Libraries in the Apache Ecosystem in 2013

Projects List	Nr. Times used Library
1. Apachelog4j	43
2. ApacheAnt	38
3. Apache Commons Compress	25
4. Apache Geronimo	18
5. Apache Commons Httpd Client	17
6. Apache Commons Jelly	16
7. Apache Commons Exec	16
8. Apache Commons VFS	16
9. Apache ORO	14
10. Apache Derby	14

[20] observed in the GNOME ecosystem a variation trend for the number of active developers very similar to the one we found in the Apache ecosystem. Specifically, after an initial increase of active developers from 1997 to 2003, they found the developers base to be almost stable until the 2008. Then, they observed a decrease of the active developers from 2008 until 2013.

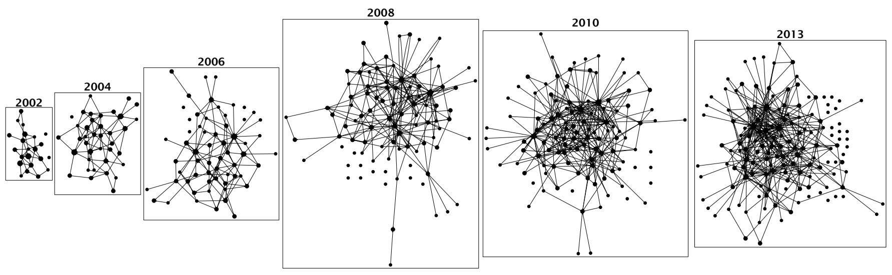
Fig. 3 shows that the number of dependencies between projects continuously increases during evolution. Similarly to the size, but differently from the number of projects, dependencies follow an exponential trend (adjusted $R^2 = 0.56$). In fact, until 2003 (when about 25 projects were in the ecosystem) there were few dependencies between the projects. After 2003, dependencies sensibly grow in the following years. This is mainly due to the fact that several projects added after 2003 are projects implementing reusable components—like those belonging to the Apache Commons[17]—that are used as libraries by several Apache projects. For example, the number of client projects for Apache Commons Compress[18] grows up to 20 (April 2013). Table 3 reports the ten most used libraries in the Apache ecosystem in 2013. As expected, all of them belong to the Apache Commons project.

To provide a better view on how the Apache software projects and the dependencies between them evolved during time, Fig. 6 shows snapshots of the Apache ecosystem from 2002 to 2013. We ignored the years before 2002 since, as reported in Fig. 3, the number of projects (and dependencies) is quite low. In the graphs of Fig. 6, each node represents a project, while an edge connecting two nodes represents a dependency between two projects. Also, the bottom part of Fig. 6 reports information useful to describe the characteristics of the depicted graphs. In particular, we report for each graph:

– *Descriptive statistics of the degree of the nodes present in it.* The degree for a node (project) P_i is the sum of its in-degree (i.e., the number of projects using P_i as library) and out-degree ((i.e., the number of projects P_i uses as library). Higher degree values indicate a higher number of projects' dependencies in the Apache ecosystem.

[17] `http://commons.apache.org/`

[18] `http://commons.apache.org/proper/commons-compress/`



		2002	2004	2006	2008	2010	2013
Degree	Mean	3.0	3.8	4.0	4.3	4.6	4.8
	Median	2.5	3.5	3.0	4.0	3.0	3.0
	St. Dev	1.6	1.8	2.6	3.3	4.0	4.3
Mean Shortest Path		3.0	4.5	4.8	5.5	6.3	6.6
#Hubs		8	11	15	20	34	44
Graph Diameter		3.0	4.0	5.0	6.0	6.0	6.0
Kolmogorov-Smirnov test		0.98	0.83	1.00	0.99	0.67	1.00

Fig. 6: Snapshots of Projects and their Dependencies in the Apache Ecosystem History.

- *The mean shortest path.* The average length of the shortest path existing between all pairs of nodes (projects) in the graph. The addition of new projects to the ecosystem would result in the increase of such number, except if the added projects have a very high degree (i.e., they are easily reachable from several other nodes). A low average value for the shortest path is a characteristic of small-world networks[19] [41].
- *The number of hubs.* The number of hubs present in the ecosystem. A hub is a project having several dependencies (i.e., a high degree). A high number of hubs is also a characteristic of small-world networks [41].
- *The graph diameter.* It represents the greatest distance between any pair of nodes in the graph.
- *The result (p-value) of the Kolmogorov-Smirnov test [24] performed on the graph degree distribution.* We verify if the degree distribution can be fit in a power-law distribution, which is considered an indication that the network is a small-world [41]. Having a power-law node degree distribution means to have only a few nodes with a very high degree and a large number of nodes with low degree. A p-value lower than 0.05 indicates that the degree distribution is not a power-law distribution, while high p-values indicate a good fitting to the power-law distribution.

By looking at Fig. 6 it is clear as the net of dependencies in the ecosystem grows during evolution. The degree of the nodes also grows during time, going from an average of 3.0 in 2002 up to an average of 4.8 in 2013. This indicates an increase of the dependencies existing in the ecosystem. As a consequence

[19] In a small-world network most of the nodes are not neighbors of one another but most nodes can be reached from every other by passing a small number of edges [41].

of the increasing number of projects (going from 24 in 2002 up to 147 in 2013) the average length of the shortest path goes from 3.0 up to 6.6 while the percentage of *hub projects* present in the graph almost remain constant—33% in 2002 (8 over 24 projects) *vs* 30% in 2013 (44 over 147 projects). Finally, in all snapshots the degree distribution of the corresponding graph fits a power-law distribution (the minimum p-value is 0.67 in 2010) indicating that the Apache projects and their dependencies result in a small-world network [41].

As explained in Section 2.2.2, we also analyzed the evolution of the software licenses declared by the Apache projects during time. From 1999 until 2003 we found *Apache Software License (ASL) v1.1* as the only license present in all source code files of all the existing projects. Starting from 2004 all projects started to migrate towards *ASL v2.0* and, by the end of 2004, 86% of the source code files in the Apache ecosystem already completed such a migration, leaving the remaining 14% to *v1.1*. This migration was complete in 2008. In addition to these two licenses, we just found one Apache project (i.e., Apache Tapestry[20]) containing in the majority of its source code files a different license, namely *BSD 3*. However, this does not create any legal issues for potential client projects interested in using Apache Tapestry as a library. In fact, the *ASL* is largely inspired to the *BSD* license and, contrarily to the *GPL* one, source code files having a *BSD* license can be used by source code files having an *ASL*. Given that the changes in terms of licenses observed during the Apache ecosystem history cannot generate legal issues, in our $\mathbf{RQ}_2$ we will not analyze licenses as a possible factor motivating the upgrade of a dependency for client projects.

Summary of $\mathbf{RQ}_1$. We can summarize results of $\mathbf{RQ}_1$ stating that the Apache ecosystem size and dependencies exponentially increase over time. Instead, the number of active developers increased until a certain point (2006), then it remained stable until 2011, since new projects were basically maintained by existing developers, and finally decreased because some projects became stable and required less activity.

3.2 RQ2: What are the reasons driving a client project to upgrade a dependency toward a new available release of a project it depends on in the Apache ecosystem?

Concerning ($\mathbf{RQ}_2$), we first checked if *not upgraded* releases exist in the Apache ecosystem history. Among the 1,964 releases considered in our study, 950 have been involved in at least one dependency (as client or as library) during their history. Of these 950, 140 releases belong to the 48 projects that have been used as “library project”, i.e., have at least one client project using them. Thus, these are the 140 releases that we classified as *upgraded* or as *not upgraded* by client projects, following the process described in Section 2.2.2. It is worthwhile to note that 14 of these releases have not been assigned to one of the two

[20] `http://tapestry.apache.org/`

Table 4: Spearman correlation between client characteristics and upgrade frequency.

System characteristic	ρ	p-value
LOC	0.08 (none)	<0.01
classes	0.09 (none)	<0.01
fan-in	0.03 (none)	<0.01
fan-out	0.13 (small)	<0.01

sets, due to the fact that they are upgraded by some client projects and not upgraded by other clients. Af for the other releases, 87 belong to the *not upgraded* set, while 39 have been assigned to the *upgraded* set. This means that 69% of new releases of Apache software projects are "ignored" by client projects that depend on such projects.

In the following, we report and discuss results for each one of the independent variables of $\mathbf{RQ}_2$ described in Section 2.2.2.

Project Characteristics. We checked whether specific project characteristics—e.g., project size, number of dependencies, or fan-in and fan out—impact the upgrade frequency of client projects, measured as explained in Section 2. Table 4 reports the results of the Spearman correlation between LOC, classes, fan-in, and fan-out of client projects and their upgrade frequency. The table clearly shows that none of the considered properties has a strong correlation with the library upgrade frequency.

We also checked whether client projects having a fan-out higher than one (i.e., depending on more than one library) tend to upgrade their dependencies all together or by selecting the libraries to upgrade/not upgrade. In particular, given L_C the set of libraries used by projects C, t_{C_i} the time when the ith C release has been issued and L_{t_i} the subset of L_C for which a new release is available at time t_{C_i}, we verify which percentage of L_{t_i} is generally upgraded by clients. Note that, since we are interested in understanding if client projects upgrade their dependencies all together or not, we just focused our attention on cases where at least one library has been upgraded. We found that, on average, clients upgrade together 60% of the new available releases of dependencies they depend on, ignoring a further 40% available for upgrades. This highlights a selection made by the client projects on which libraries upgrade and which libraries ignore, the main topic of our investigation in this research question.

Structural Changes. Fig. 7 reports the boxplots for different type of changes for releases that are ignored by client projects (i.e., *not upgraded releases*), and releases used by client projects to upgrade their dependencies (i.e., *upgraded releases*). Moreover, Table 5 reports results of the Mann-Whitney test (p-values, significant ones highlighted in bold face) and the Cliff's d effect size when comparing the distributions for the different types of changes performed on *upgraded* and *not upgraded* releases. On average, in *upgraded releases* there are 25 times more added classes than in *not upgraded releases* (125 *vs* 5)—see

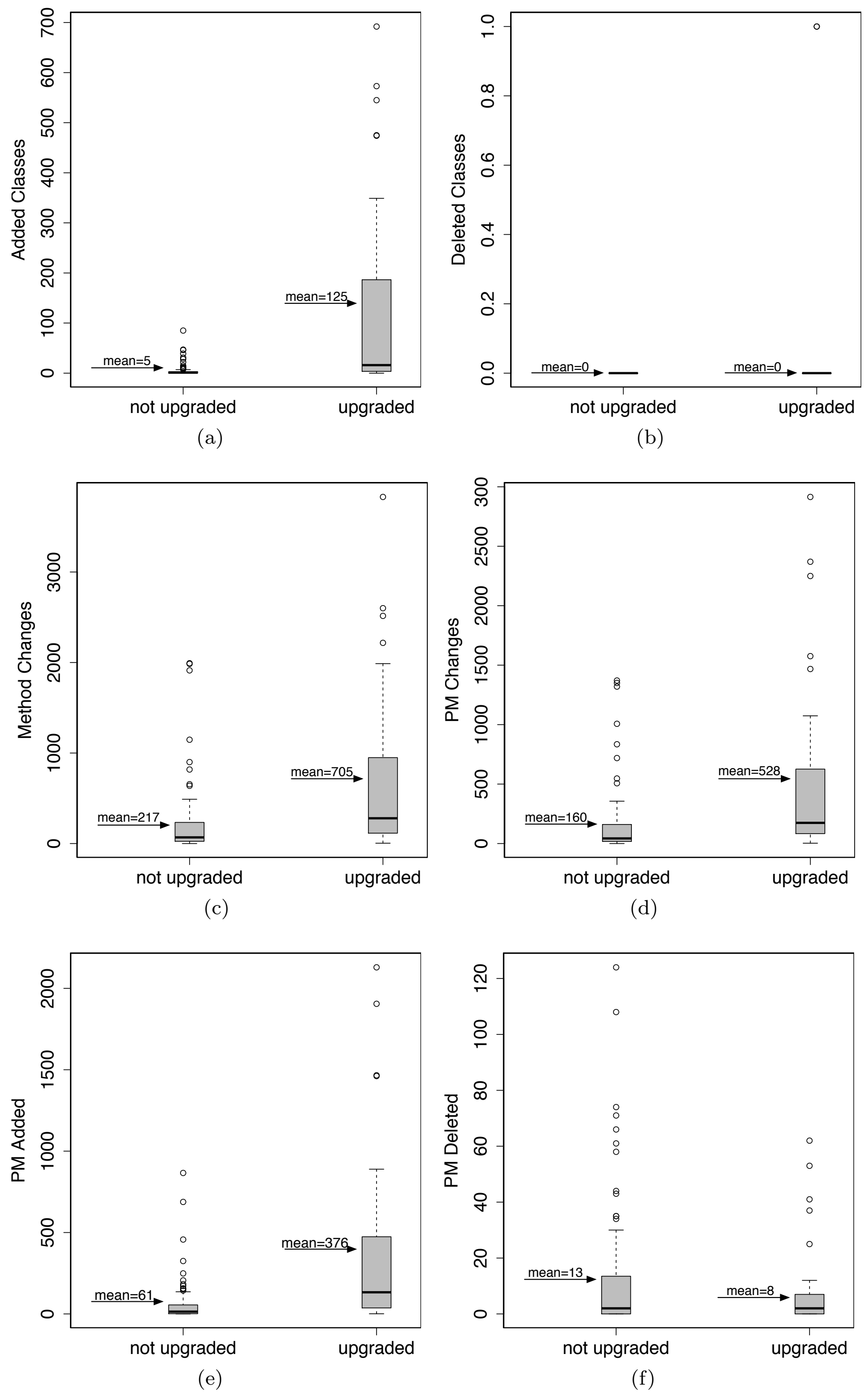


Fig. 7: Changes in upgraded and not upgraded releases.

Table 5: Changes and fixed bugs in upgraded and not upgraded releases: Mann-Whitney test (adj. p-value) and Cliff's (*d*).

Tested	p-value	Cliff's *d*
Added Classes	< **0.0001**	0.62 (Large)
Deleted Classes	0.51	0.05
Method Changes	<**0.0001**	0.48 (Large)
PM Changes	<**0.0001**	0.46 (Medium)
PM Added	<**0.0001**	0.57 (Large)
PM Deleted	0.48	-0.01
Fixed Bugs	<**0.0001**	-0.35 (Medium)

Fig. 7(a). As shown in Table 5, this difference is statistically significant (p-value <0.0001) with a large effect size (0.62). Fig. 7 shows that, in general, there are no deleted classes (with respect to the previous release) in both kinds of releases—see Fig. 7(b)–and thus, no statistically significant difference.

As for the changes applied to existing methods (i.e., methods already present in a previous release of the project), we observed almost three times more changes for the *upgraded releases* when analyzing all methods in the system (705 *vs* 217)–see Fig. 7(c)—as well as when just focusing on public methods (that are those used by the client projects), 527 *vs* 160—see Fig. 7(d). For both kinds of changes, results in Table 5 highlight statistically significant differences between *upgraded* and *not upgraded* releases, with a large effect size (0.48) when considering all methods, and a medium effect size (0.46) when just focusing on public methods. Also, the number of added public methods is larger in the *upgraded releases* (six times larger) than in *not upgraded* releases—see Fig. 7(e)—with statistical significance and a large effect size (0.57).

All these results quantitatively highlight that *upgraded releases* contain changes affecting the interfaces and substantial changes, if compared to the *not upgraded releases.* This is particularly evident when focusing on added classes (29 times more) that are likely related to new features provided by the new project release, and on added public methods (six times more than for *not upgraded releases*), that represent new services available to the client projects. Also, the higher number of overall method changes (three times more than *not upgraded releases*) highlights substantial changes in the *upgraded releases* if compared to the *not upgraded releases.* The only change for which we did not observe a higher proportion in the *upgraded releases* are the deleted methods (-63%)–see Fig. 7(f). Note that deleted public methods mean removed services for the client projects. Thus, it is reasonable to think that client projects using the removed services tend to not upgrade the dependency towards the new release until they fix the client code in order to properly works with the new release. This could explain the lower number of deleted methods for *upgraded releases*, compared to *not upgraded releases.* However, this difference is not statistically significant (see Table 5).

Number of fixed bugs. Concerning the number of bugs fixed in *upgraded* and *not upgraded* releases, Fig. 8 reports their distribution. On average, the

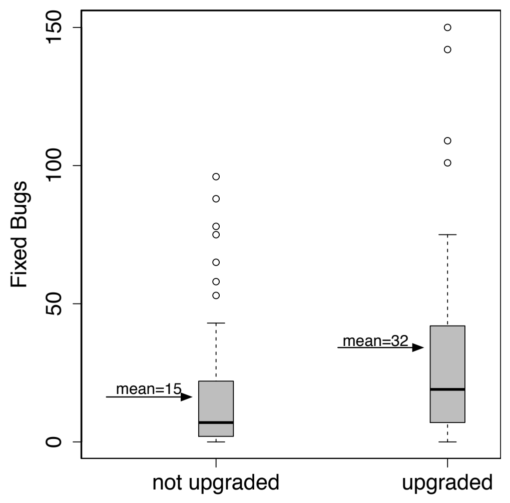


Fig. 8: Fixed bugs in upgraded and not upgraded releases.

Table 6: Analysis of release notes for upgraded and not upgraded sets of releases.

Release type	Minor	Major	Bug fixing
not upgraded	79%	21%	82%
upgraded	59%	41%	92%

number of bugs fixed in the *upgraded releases* is more than two times greater than for *not upgraded releases* (32 *vs* 15). Also, this difference is statistically significant with a medium effect size (-0.35)—see Table 5.

Nature of the release. As explained in Section 2.2.2, to provide further evidence to the results reported above, we inspected the release notes of both *upgraded releases* and *not upgraded releases* to understand what are the changes generally declared by developers when releasing both kinds of releases.

First, we found release notes of *upgraded releases* much longer than those of *not upgraded releases.* For instance, Apache log4j release notes for the six *not upgraded releases* considered in our study are composed, on average, of 676 words each, against the 2,339 words of the five *upgraded releases.* The same difference can be observed between the release notes of the four Apache Ant *not upgraded releases* having an average length of 1,417 words and those of the eight *upgraded releases* with an average of 10,476 words. This suggests that release notes for *upgraded releases* have a longer content, which often means (as confirmed by a manual analysis) describing much more novelties, improvements, and bug fixes. For example, Apache log4j releases from 1.2.5 to 1.2.8 (4 releases), plus 1.2.11 and 1.2.12 belong to the *not upgraded releases* set. Their six release notes describe, in total, 29 bug fixes and one perfective maintenance activity, the latter being a different option to initialize the system. Instead, release notes for the five *upgraded releases* (i.e., 1.2.9, 1.2.13, 1.2.14, 1.2.16, and 1.2.17) include 123 fixed bugs, two perfective maintenance activities, and one new feature.

Among the six LOG4J *not upgraded releases*, five have been tagged as *minor* (83%) while one (1.2.12) as *major* (17%). Also, all of them have been tagged as *bug fixing*. Concerning the five *upgraded releases*, two (i.e., 1.2.9 and 1.2.13) have been tagged as *minor* (40%), while three as *major*. Also in this case, all 5 releases have been also tagged as *bug fixing*.

The classification of the inspected release notes is reported in Table 6. As we can see, 79% of *not upgraded releases* have been tagged as *minor*, against 59% of the *upgraded releases*, while 41% of *upgraded releases* have been tagged as *major*, against 21% of the *not upgraded releases*. Concerning the bug-fixing activities declared in release notes, overall 82% of the release notes for *not upgraded releases* have been tagged as *bug fixing*, against 92% of the *upgraded releases*. Note that finding some kind of reference to a performed bug-fix in release notes is quite the norm. Thus, it is expected that the difference in terms of *bug fixing* between upgraded and not upgraded releases is not that high. Despite this, the number of *upgraded releases* tagged as *bug fixing* is about 10% greater than the number of *upgraded releases*.

Overall, the inspection of the release notes confirms that *client projects tend to upgrade their dependencies when substantial changes in the projects they depend on are released, including bug-fixing activities.*

Licensing changes. Since we found in **RQ**$_1$ that nearly all licensing changes concerned two different versions of the *ASF*, this did not create any licensing issue, and for such a reason it is no longer necessary to study the impact of such a factor.

Developers' overlap. Concerning the overlap of developers between projects, we first investigated whether pairs of projects having a dependency share a greater number of developers than projects do not having a dependency. Fig. 9 reports the distribution of developers' overlap between projects having and not having a dependency, showing that the former generally share a higher number of developers as compared to the latter. This difference is also statistically significant: the Mann-Whitney test returned a p-value <0.01, with a medium effect size (Cliff's d=0.47).

After that, we checked whether among the pairs of projects having a dependency, those sharing a higher number of developers with the library they use also exhibit a higher upgrade frequency by the client (i.e., the client tends to upgrade more frequently to new releases of the library projects). In particular, we computed the Spearman correlation between the client upgrade frequency and the average overlap of developers it has with its libraries. However, we only observed a small correlation (ρ =0.13, p-value<0.01).

Summary of RQ$_2$:

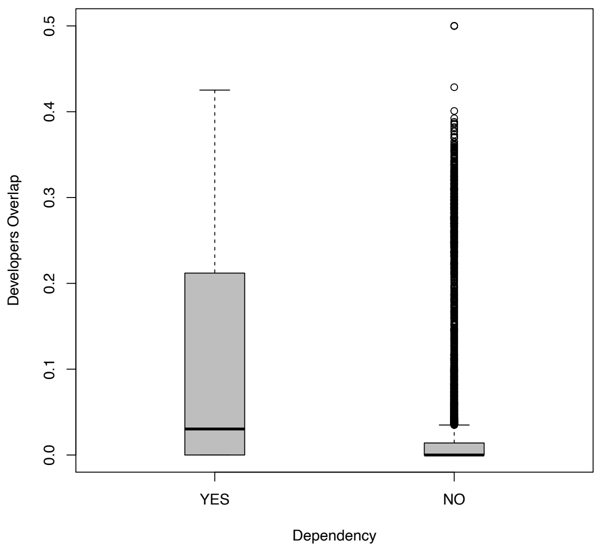


Fig. 9: Developers' overlap (in percentage) in projects having and not having a dependency.

1. *Client projects' characteristics do not influence their upgrade frequency.* The projects' size (in terms of LOC and number of classes) and the number of dependencies of the client projects (fan-in and fan-out) do not correlated with the upgrade frequency.
2. *Client projects tend to upgrade their dependencies when substantial changes in the projects they depend on are released, including bug-fixing activities.* This result has been confirmed by both the quantitative and qualitative analysis we performed.
3. *Pairs of projects having a dependency share a greater number of developers than pairs of projects not having a dependency.* This result is a first indication that client and library projects co-operate in the management of the dependency, that will be object of our $\mathbf{RQ}_3$.
4. *When client projects upgrade their dependencies, they do not perform all available upgrades.* On average, client projects upgrade 60% of the new available releases together, confirming the selection on the basis of the criteria of point 2.

Table 7: Tags manually assigned to the 871 discussions talking about dependencies between projects.

Tag	Number of discussions	Percentage
GENERAL TAGS		
client	759	87%
library	112	13%
DEVELOPERS TAGS		
only developers client	725	83%
only developers library	107	12%
both developers	39	5%
TOPIC TAGS		
break	24	3%
fix	283	33%
upgrade	187	22%
use	53	6%
other	324	36%

3.3 RQ3: How are dependencies discussed between open source developers?

Among the 7,685 discussions manually analyzed, 871 received the *dependency* tag, indicating that the discussion was actually about the management of a dependency between a client and a library project. Table 7 reports, for each of the tags considered in our study, the number (and percentage) of discussions to which it has been assigned.

Starting from the "general tags", it is clear that most of the discussions on dependencies' management is carried out on the client projects' side. In fact, 759 out of the 871 discussions (87%) were extracted from the clients mailing lists and issue trackers. This is an expected result, since it is reasonable to think that between client and library projects the former ones are those more interested in the correct working of dependencies. This is also confirmed by the "developers tags" showing as 83% of discussions related to dependencies only involve developers from the client project. However, even if in a smaller proportion (12%), also developers of the library projects discuss about dependencies management, sometimes together with the developers of the client projects (5%).

Concerning the topic object of the discussions ("topic tags" in Table 7), we found 33% of them focused on fixing problems caused by a dependency (tagged as *fix*). Specifically, we observed discussions concerning different kinds of problems. The most common problems observed are those related to bugs present in the used library, consequently causing a bug in the client project. For instance, such a kind of problem was discussed in the Apache Stanbol[21] mailing list. Apache Stanbol is a software providing reusable components for semantic content management (e.g., automatic tag extraction from webpages, text completion in search fields, etc.), and it uses as library Apache Tika[22], a toolkit able to detect and extract metadata and structured text from various

[21] `http://stanbol.apache.org/`

[22] `http://tika.apache.org/`

document formats. Developers of the client project discuss[23] about problems related to the extraction of metadata from JPEG images. This feature is provided to STANBOL by the TIKA library.

Another discussion tagged as *fix* occurred in the APACHE MINA[24] mailing list. MINA, a network application framework to develop high performance and scalability networks, is used as library by the APACHE SSHD[25] project, supporting SSH protocols for client-server communications. In this case[26], developers are discussing about a problem found MINA 2.0.2, and causing a bug in SSHD. The solution has been the simple upgrade to MINA 2.0.4, fixing the reported issue. This example qualitatively supports one of our **RQ**$_2$ findings: *client projects tend to upgrade their dependencies when bug-fixing activities have been performed.*

22% of the analyzed discussions was tagged as *upgrade*, indicating that the discussion was focused on whether upgrading or not a dependency towards a new available release of a library project. A very interesting example is the one we found in the APACHE TORQUE[27] mailing list. TORQUE is an object-relational mapper for java using several Apache projects as library (i.e., COMMONS BEANUTILS, COMMONS COLLECTIONS, COMMONS CONFIGURATION, COMMONS LANG, XERCES-J, XML COMMONS, VELOCITY, and ANT). Developer T.F. wrote in the discussion[28]:

> *I have gone through the libraries Torque depends on and seen if there is a newer version available. Those are the updates I would suggest:*
>
> *commons-beanutils from 1.6.1 to 1.7.0*
> *commons-collections from 3.0 to 3.1*
> *commons-configuration from 1.0 to 1.1*
> *commons-lang from 2.0 to 2.1*
> *xercesImpl from 2.4.0 to 2.6.2*
> *xml-apis from 1.0.b2 to 2.0.2*
> *ant from 1.5.1 to 1.6.5*
>
> *[...]*
>
> *Note that velocity is not updated to 1.4. I have heard rumors that version 1.4 has a memory leak (but I have also heard rumors that the current velocity version chokes in very large files, so not sure whether the memory leak is not already there in 1.3.1). [...]*

This discussion suggests that (i) sometimes the choice to ignore a new available release of a library the client depends on (VELOCITY 1.4 in this case)

[23] `http://tinyurl.com/p3nxkyc`
[24] `http://mina.apache.org/`
[25] `http://mina.apache.org/sshd-project/`
[26] `http://tinyurl.com/nfrqfrf`
[27] `http://db.apache.org/torque/torque-4.0/index.html`
[28] `http://tinyurl.com/qjyw6u2`

is based on the fear to introduce errors in the client project, and (ii) even when some effort is spent to upgrade dependencies like in this case, not all dependencies are upgraded together, as also highlighted by the quantitative analysis we performed in $\mathbf{RQ}_2$.

Several discussions tagged as *upgrade* also confirmed the fact that potential brakes in the client push away the client from upgrading their dependencies. For instance, in the APACHE ROLLER[29] mailing list we found a discussion [30] on whether upgrading or not a dependency towards APACHE LOG4J. In particular, one of the ROLLER developers asked if it is the case to upgrade the release of LOG4J used in ROLLER:

> *log4j is up to 1.2.12. We're still using/distributing 1.2.4 in the trunk (bound for 2.0). I think we should upgrade to 1.2.12 in the trunk.*

The answer came from another ROLLER developer:

> *I recently tried to upgrade to 1.2.12 and found that there were some incompatibilities with my config file. I forget what they were - but it basically wasn't a simple upgrade. For that reason, I'm currently using 1.2.11.*

Thus, even if the ROLLER developers were going to issue their new release 2.0 and were conscious of using an old LOG4J release, the choice was to not risk to perform a tricky upgrade.

The *use* tag has been assigned to 6% of the analyzed discussions, dealing on how to use the library in the client, while only 3% of them were tagged with the *break* tag, indicating discussions aimed at avoiding changes that could break the dependency. These discussions generally happen in the library projects' communication channels. For instance, in the APACHE GERONIMO mailing list we found a discussion[31] where a developer was alerting about the possible issues that could be caused by the removal of a dependency in the project: *This change removed Sun SAAJ implementation dependency. That dependency is currently needed and should not be removed (I'm pretty sure it will break CXF).* Note that APACHE CXF[32] is a client project using GERONIMO and, among the developers of these two projects, we found a dense network of communication mostly carried out by developers in overlap between the two projects. We depicted this network in Fig. 10, where CFX's developers are represented with blue nodes, GERONIMO's developers with orange nodes, and developers in overlap between the two projects are colored in yellow. An edge exists between two developers if they exchanged at least two messages (i.e., the communication between them is not occasional). Blue edges are messages exchanged in the CFX's communication channels, while the red ones are messages exchanged in the GERONIMO's communication channels. From Fig. 10 it is interesting to notice that (i) most of the developers in overlap are hubs exchanging messages

[29] `http://roller.apache.org/`
[30] `http://tinyurl.com/qz254l8`
[31] `http://tinyurl.com/opmlu8z`
[32] `http://cxf.apache.org/`

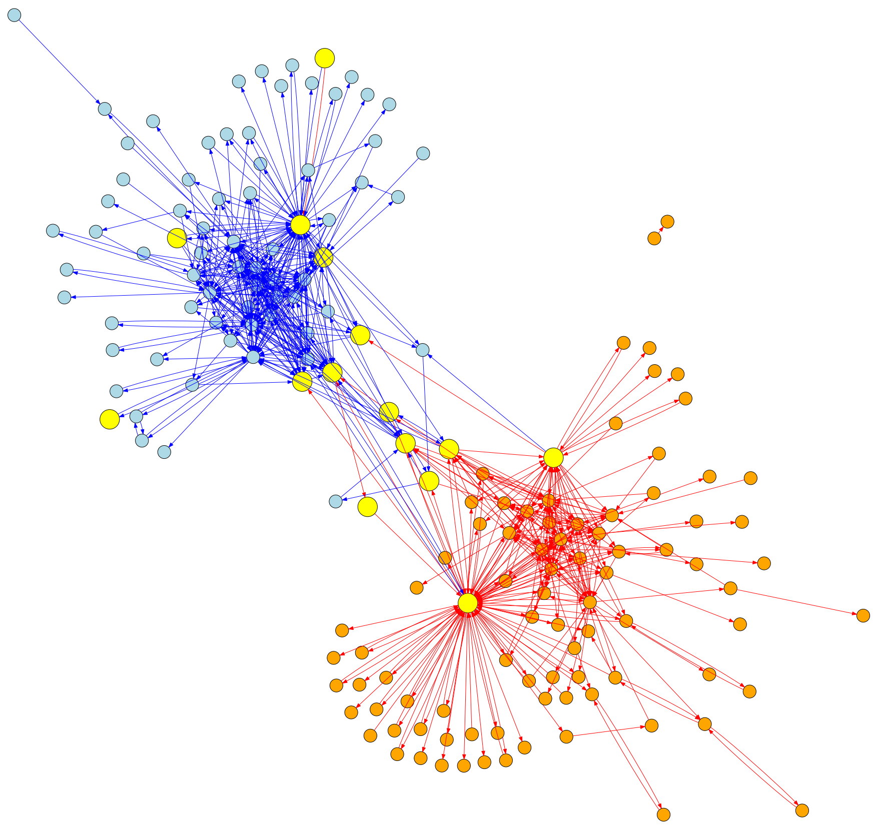


Fig. 10: Communication network between Geronimo and CFX developers. CFX's developers are shown in blue, Geronimo's developers in orange, while yellow circles are developers overlapping between the two projects.

with several other developers, (ii) most of the communications between the two projects passes through the developers in overlap (i.e., the yellow nodes).

Finally, 36% of discussions were tagged as *other*, because they were related to topics that cannot be placed in the previous discussed tags. Examples are discussions related to missing references in the release bundles, to the possibility of whether or not providing support to old releases of some clients, or to legal issues. An example was a discussion started by an Apache Aperture developer:

Hello Tika!
Hello Aperture!

We (the Aperture project) have recently updated the pdfbox to the current trunk version. It seems that they've introduced a new dependency

Table 8: Impacted source code components in client projects.

	#Classes (%)	#KLOC (%)
Mean	58 (5%)	65 (6%)
Median	6 (1%)	12 (1%)
St. Dev.	122 (9%)	14 (12%)
Min	1 (0%)	0.039 (0%)
Max	518 (41%)	77 (62%)

on the Java Advanced Imaging API (JAI). The problem is that JAI imposes certain constraints on redistribution. [...]

Summary of $\mathbf{RQ}_3$. The manual analysis performed in the context of $\mathbf{RQ}_3$ showed that developers actively discuss about dependency management. Generally, this discussion is carried out by developers of the client projects mainly discussing about problems due to the (*fix*) of dependencies, and of the possibility to whether or not upgrading a dependency. Results of $\mathbf{RQ}_3$ qualitatively confirmed some of the findings of our $\mathbf{RQ}_2$, and in particular: (i) when client projects upgrade their dependencies, they do not always perform all available upgrades, and (ii) client projects tend to upgrade their dependencies when bug-fixing activities have been performed.

3.4 RQ4: If a dependency is upgraded, to what extent would this impact on the client project source code?

Table 8 reports descriptive statistics of the impacted source code of client projects upgrading one of their dependencies. The values are reported in terms of impacted number (percentage) of classes and number (percentage) of KLOCs of the client project.

On average, the impacted source code of the client project is quite limited, about 5% of the total number of classes and 6% of the KLOCs. This is quite expected, since most the dependencies a client project has are just due to few classes exploiting the services provided by this dependency. For instance, all the dependencies towards the Apache Commons projects are generally due to few methods in the client code exploiting the offered services, like the `compressors` and `archivers` services provided by the Apache Commons Compress project to manipulate archive files, or the collection of I/O utilities available in the Apache Commons IO project. Since these services support the implementation of specific tasks, it is expected that they just impact on classes having such tasks among their responsibilities.

Instead, there are some projects offering very wide services exploited by a great part of the client project source code. This consideration can be derived by looking at the row "Max" of Table 8, reporting the maximum value of impacted client source code we measured in our study. This value is referred to a dependency that the project Apache Accumulo[33] (client project) has

[33] `http://accumulo.apache.org/`

towards the project Apache Hadoop. Accumulo is a database system, while Hadoop is a framework supporting distributed processing of large data sets across clusters of computers using simple programming models. Accumulo exhibits dependencies towards Hadoop in 518 of its 1,263 classes (41%), for a total of 77 KLOCs impacted (62% of the total size). Other projects exhibiting an high impact on the client code are Apache Tomcat[34], impacting on average 23% of the client projects KLOCs, and Apache MINA with an average of 10%. Again, both projects offer very generic services that could be reasonably exploited by several classes in the client projects. In fact, Apache Tomcat is an implementation of the Java Servlet and JavaServer Pages (JSP) technologies, while Apache MINA is an application framework helping users in developing high performance and high scalability network applications.

Summary of $\mathbf{RQ}_4$. Results highlight that the proportion of source code of client projects impacted by changes in the projects they depend on is quite limited, around 5%. However, there are specific dependencies, generally towards frameworks/libraries offering very wide services, that could strongly impact the client project source code when a dependency is upgraded.

4 Threats to Validity

This section discusses the threats that can affect the validity of the achieved results.

Threats to *construct validity* concern the relation between the theory and the observation. They can be mainly due to imprecisions in the measurements we performed. This is a summary of the main sources of imprecision:

- The mapping between dependencies declared within a project and other projects was performed using a set of heuristics, as explained in Section 2.2. To cope with the imprecision of such heuristics, results were manually verified. Still, human errors are possible in this manual checking.
- The analysis of licensing relies on the precision of Ninka, which is deemed to be greater than 90% [18].
- In the analysis of the evolution of active developers in $\mathbf{RQ}_1$ and $\mathbf{RQ}_2$, we cannot exclude that the projects also involved other contributors whose activity is not evident from the versioning system commits.
- The fine-grained extraction of changes performed by using the MARKOS code analyzer is not guaranteed to be 100% precise, especially due to the use of a fingerprint-based approach to identify renaming/moving of code components distinguishing them from the deletion of a component accompanied by the addition of a new one.
- The analysis of the nature of changes performed in $\mathbf{RQ}_2$ involved a manual classification of releases. This could have lead to some subjectivity in the classification. To avoid that, two of the authors performed the classification independently, and then discussed cases where their choices were inconsistent.

[34] `http://tomcat.apache.org/`

- In $\mathbf{RQ}_2$, we identify whether a change is a bug fix or not using information contained in the versioning system. Although the Apache (Bugzilla) issue tracker has an explicit "Defect" value for the severity field, and Jira always foresees a specific distinction between bug fixes and other changes, imprecisions on such a classification are still possible [2].
- The analysis of developers' communication performed to address $\mathbf{RQ}_3$ has been conducted by considering, as communication means, project mailing lists and issue reports. In many projects—and especially in worldwide-distributed open source projects like the ones we analyzed—it is a consolidated practice to communicate through mailing lists and issue trackers. However, we are aware that there could still be some hidden communication [3] we might have missed in our analyses. A different matter concerns the manual tagging of such a communication which, due to the large number of emails/issues to be analyzed (7,695), was split between the two inspectors. Although we are aware that mistakes could have occurred, both inspectors agreed on guidelines to perform a classification, and they discussed together unclear cases.
- The analysis of change impact done in $\mathbf{RQ}_4$ includes all client classes importing an API class that underwent a change. To determine whether a changed method was used or not, a fine-grained analysis would have been necessary. However, this was not our intent. Instead, we were interested to determine the potential set of clients for the changed API class, i.e., a set of classes that might need some verification/testing activities.

Threats to *internal validity* concern factors internal to the study that could influence our results. Such kind of threats typically do not affect exploratory studies like the one in this paper. The only case worthwhile of being discussed is about $\mathbf{RQ}_2$ (reasons for upgrades) and to some extent $\mathbf{RQ}_4$ (why some changes in libraries have more impact than others). In the first case, although we have found some correlation between certain kinds of changes and upgrades decisions, we cannot claim there is a cause-effect relation. Nevertheless, we manually inspected release notes to support our findings. In addition to that, the large manual analysis of developers' communication conducted in the context of $\mathbf{RQ}_3$ provided a strong qualitative support to the quantitative findings of $\mathbf{RQ}_2$. Last, but not least, in $\mathbf{RQ}_2$ we also checked whether simple factors related to intrinsic projects' characteristics could have been the main reasons for our findings (rather than the other factors we observed), however we did not find any empirical evidence of that.

Threats to *conclusion validity* concern the relationship between the treatment and the outcome. The analyses performed in this paper mainly have an observational nature, although we used, where appropriate ($\mathbf{RQ}_2$), statistical procedures and effect size measures to support our claims.

Threats to *external validity* concern the generalizability of our findings. Such a generalizability is clearly limited to the ecosystem being analyzed, i.e., Apache, and specifically Java projects of the Apache ecosystem. Also, in terms of assessing dependency upgrades, such assessment is confined to *within-*

ecosystem dependencies, as we are not interested to analyze dependencies to projects that are not part of the ecosystem. Future studies need to be done to investigate upgrades with respect to external dependencies too, and to repeat the study on other ecosystems.

5 Related work

In this section, we discuss the related literature, focusing our attention on (i) work studying software ecosystems and (ii) work observing the impact on software evolution and stability of changes/deprecations of APIs.

5.1 Analysis of Software Ecosystems

In the last decade several software ecosystems have been studied from different perspectives. Table 9 reports these studies, classifying them by (i) the ecosystem being studied, (ii) the source of information exploited, (iii) the objectives of the study, and (iv) the main findings.

One of the first software ecosystems subject of several empirical studies has been the Debian Linux distribution [22,19,17]. Specifically, Godfrey *et al.* [19] analyzed the size of the Linux operating system Kernel, observing a super-linear rate growth for several years. Gonzalez-Barahona *et al.* [22] found that the Debian Linux distribution has been doubling in size every two years while the average size of its packages remained stable over time. Also, they observed as the number of dependencies between packages increased exponentially. German *et al.* [17] proposed a methodology and visualization tool aimed at supporting the study of inter-dependencies in complex software systems. The tool has been used to analyze the dependencies between projects in the Debian Linux distribution. Capturing dependencies between projects in an ecosystem is far from trivial [35] and it is the reason why several authors focused their attention on methods for the extraction of dependencies in large software ecosystems [33,17]. Similarly to what done in other studies, in the context of our work we also exploited specific heuristics (see Section 2.2) to identify the dependencies between projects.

Another software ecosystem that has been studied by several authors is the Eclipse IDE. Wermelinger *et al.* [42,43] analyzed the evolution of the Eclipse's architecture and found that the success as application framework for the Eclipse SDK mainly depends from the fact that it *"follows several practices that support sustainable architectural evolution"* [43]. In particular, the Eclipse developers manage APIs carefully, avoiding to break existing APIs when issuing new releases. Mens *et al.* [34] found that the Eclipse core plugins adhere to the laws of continuing change and growth, but not to the law of increasing complexity. Businge *et al.* [7] analyzed the dependencies and the survival of 467 Eclipse third-party plugins. They found that plugins depending only on stable and supported Eclipse APIs have a very high source compatibility success rate. This means that third-party plugins that depend from stable

Source	**Objectives**	**Findings**	**Ref**
	APACHE ECOSYSTEM		
Versioning System and Mailing lists	Defining a set of invariant metrics to detect "stagnant projects".	Stagnant projects can be identified by measuring the ratio of the e-mail exchanged in mailing lists and the number of commits.	[14]
Versioning System, Release Notes, Issue Trackers, and Mailing lists	Focus on projects dependency management.	Clients tend to upgrade their dependencies when libraries are subject to bug fixes, while changes to interfaces make the upgrade less appealing. Most of the times the impact of upgrades is well-confined.	**Our Work**
	LINUX ECOSYSTEM		
Versioning System	Analysis of the evolution of the Linux Kernel.	The size of the Linux Kernel has been growing at a super-linear rate for several years.	[19]
Versioning System	Analysis of the evolution of the Debian Linux distribution	The overall size has been doubling every 2 years, while the average size of packages remained stable. Instead, the number of dependencies increased exponentially.	[22]
Versioning System	Analysis of dependencies in the Debian Linux distribution.	A methodology and visualization for studying inter-dependencies of a complex software system.	[17]
	SQUEAK ECOSYSTEM		
Versioning System	Recover dependencies between the software projects of the Squeak ecosystem.	Accurate detection of dependencies for Smalltalk.	[33]
Versioning System	Analysis of API changes in a software ecosystem.	API changes caused by deprecation can have a very large impact on the ecosystem in terms of the number of changes needed to fix broken dependencies.	[37]
	ECLIPSE ECOSYSTEM		
Versioning System	Analysis of the Eclipse architecture.	The development follows a systematic process and there is a stable architectural core that remains since the first release.	[42, 43]
Versioning System	Analysis of the evolution of Eclipse core plugins.	Eclipse plugins adhere to the laws of continuing change and growth, but not to the law of increasing complexity.	[34]
Versioning System	Analysis of the evolution of third-party plugins.	Third-party plugins that depend from stable and supported Eclipse APIs have a higher compatibility success rate than plugins depending on discouraged and unsupported APIs.	[7]
Mailing lists and issue trackers	Analysis of developers' productivity.	Adding new features to Eclipse slows down the bug fixing process.	[28]
	GNOME ECOSYSTEM		
Versioning System	Analysis of the active developers.	The number of active developers increased until 2003, remained stable until the 2008, and then decreased.	[20]
Versioning System	Analysis of the ecosystem evolution.	A list of practices that could benefit both open and commercial software development organizations.	[16]
Mailing lists and issue trackers	Analysis of the activities on and contributors of the software ecosystem.	GNOME contains both paid contributors and volunteers. Coding is the most preeminent activity in the ecosystem.	[21, 30, 40]
Versioning System	Analysis of cloning and copying operations between GNOME Projects.	Larger clones exist between the sub-projects of GNOME and more than 60% of the clone pairs can be automatically separated into original and copy.	[17]
Versioning System, Mailing lists and Issue Trackers	Analysis of social processes in the ecosystem.	Participants in such ecosystems may be able to use a significant amount of transferrable knowledge when moving between projects in the ecosystem and, thereby, skip steps in the "onion model"	[27]
	GNU R ECOSYSTEM		
Versioning System and Mailing lists	Analysis of the differences between code characteristics of core and user-contributed packages.	User-contributed packages has been growing steadily since the R conception at a significantly faster rate than core packages.	[15]
	GURUX ECOSYSTEM		
Versioning System and Mailing lists	Analysis of the evolution.	Supporting processes, guidelines and best practices for building open source communities.	[29]
	FIREFOX, UNITY, AND	**GOOGLE CHROME ECOSYSTEMS**	
Versioning System	Analysis of how the software licenses are reported in software ecosystems.	Software component licenses and the architectural composition of a system help to better define the software ecosystem "niche" in which a given system lies (i.e., the license is wrong).	[38]

Table 9: Studies that analyzed software ecosystems.

and supported Eclipse APIs have a higher source compatibility success rate than plugins depending on discouraged and unsupported Eclipse non-APIs. In addition, Businge *et al.* found that the majority of plugins hosted on SourceForge[35] do not evolve beyond the first year of release while, Singh *et al.* [39] observed how committers networks in SourceForge are small-world networks.

GNOME is another very well investigated software ecosystem [20,16,30, 31,40,21]. German *et al.* [16] distilled a list of practices that could benefit both open and commercial software development organizations by studying the GNOME ecosystem. For example, a careful coordination of the development activities between the sub-projects belonging to the ecosystem can be the one of the keys for the success of the different projects. Mens *et al.* [21,40] studied the GNOME mailing lists and issue trackers observing that GNOME contains both paid contributors and volunteers. Also, coding seems to be the most preeminent activity in the ecosystem, followed by activities such as translation and development documentation. In addition, members of the GNOME community tend to specialize themselves in a limited number of activity types [21]. Goeminne *et al.* [20] observed in the GNOME ecosystem a variation trend for the number of active developers similar to the one we identified in the Apache Ecosystem. Specifically, after an initial increase of active developers in the 1997-2003 time window, they found the developers base to be almost stable until 2008. Then, they observed a decrease in the number of active developers up to date.

German *et al* [15] studied the evolution of the statistical computing project GNU R, with the aim of analyzing the differences between code characteristics of core and user-contributed packages. They found that the ecosystem of user-contributed packages has been growing steadily since the R conception at a significantly faster rate than core packages, yet each individual package remains stable in size.

Scacchi *et al* [38] examined how the software licenses are reported in software ecosystems and in particular, observe how software component licenses and the architectural composition of a system help to better define the software ecosystem "niche" in which a given system lies (i.e., the license is wrong).

Other than the analysis of ecosystems evolution, social/community aspects of ecosystems (for example bug reports and/or mailing list traffic between developers teams) have also been analyzed [27,30,29,28]. Kidane *et al.* [28] found that adding new features to Eclipse slows down the bug fixing process.

Yu *et al.* [44] studied the mailing lists of the Linux kernel, to analyze different ecosystem collaboration patterns between companies. Jergensen *et al.* [27] instead, analyzed multiple systems which have "common underlying components, technology, and social norms". They observed how participants in such ecosystems may be able to use a significant amount of transferrable

[35] `http://sourceforge.net/`

knowledge when moving between projects in the ecosystem and, thereby, skip steps in the "onion model"[36]

Annosi *et al.* [1] proposed a framework to support developers in the upgrade of third-party components. The decision is driven by various factors, partially related to the kind of change occurred in the component (as mined from release notes or issue trackers), and partially on expert judgements collected within the company. The work presented in this paper is complementary to the work of Annosi *et al.*, because it helps to identify what are the factors and events that trigger component upgrades in a large software ecosystem.

Gala *et al.* [14] also analyze the Apache ecosystem proposing a set of "invariant metrics" in the domain of software projects. They found that metrics measuring the proportion (or ratio) of the e-mails exchanged in mailing lists and the total number of commits performed by developers can be useful to identify stagnant projects and projects in danger of stagnation. In our study, we also observed that project's stagnation is one of the factors that reduces the active developers' base.

In a previous work [4] (extended by the present paper), we performed a first analysis of the Apache ecosystem, highlighting the exponential growth of the ecosystem size, investigating some of the factors that could affect dependency upgrades, and analyzing the impact of upgrades on clients' code. With respect to our previous work [4], this paper deepens the analysis of the ecosystem evolution (e.g., analyzing the evolution of the number of developers). Also, it considers several additional factors that could have influenced dependency upgrades, not considered in the previous paper (e.g., project characteristics, nature of releases, developers' overlap between client projects and library). Finally, it reports a large qualitative analysis of how dependency upgrades are discussed over developers' communication.

5.2 Analysis of API Changes

From a theoretical point of view, the API of a component in a software system should never change. Successful software systems try to avoid change in the APIs that can impact the stability of software components [43]. However, what happens in practice is that when a new version of a software component is released, it is very likely that its interface changes. This requires projects that use the component to be changed before the new release of the component can be used. How and why API change during software evolution has been studied by several authors. Dig *et al.* [13] studied the changes between two major releases of four frameworks (one proprietary and three open-source) and one

[36] The onion model is a socialization process where newcomers join a project by first contributing through mailing list discussions and bug trackers and they advance to more important roles contributing where they can improve the code and making design decisions.

library written in Java. They found that on average 90% of the API breaking changes[37] are represented by refactoring operations.

Hou *et al.* [25] analyzed the evolution of AWT/Swing at the package and class level. They found that—during 11 years of the JDK release history (i.e., since JDK 1.0 to Java SE 6)—the number of changed elements was relatively small compared to the size of the whole API, and the majority of them happened in release 1.1. Thus, the main conclusion of their study is that the initial design of the APIs contributes to the smooth evolution of the AWT/Swing API. Raemaekers *et al.* [36] studied changes in APIs to measure the stability of the Apache Commons library. Their findings suggested that a relatively small number of new methods were added in each snapshot to the COMMONS LOGGING library, and there is more work going on in new methods of COMMON CODEC than in old ones.

Recently, Robbes *et al.* [37] observed how much the API of a framework (or library) changes. They studied API deprecations that led to ripple effects across an entire ecosystem. The results showed that a number of API changes caused by deprecation can have a very large impact on the ecosystem and consequently on projects or developers that are impacted by the change, or the measure of the overall number of changes.

Changes in APIs and frameworks require the adaptation of clients, that can, sometimes, be automated. To this aim, Degenais and Robillard [11] proposed SemDiff, a tool to recommend client adaptation required when the used framework evolve. The authors evaluated SemDiff on the evolution of the Eclipse-JDT framework and three of its clients.

We share with the aforementioned papers the need for studying how the evolution of projects used as libraries in software ecosystems impacts on the evolution of client projects. However, instead of proposing how client projects should be adapted, we aimed at analyzing to what extent are dependencies upgraded—i.e., towards a new release of the target project—and what are the drivers of such upgrades. Our study provides some insights on the design of recommendation systems for supporting developers in the activity of library/component upgrade.

6 Conclusion and Future Work

This paper investigated on the evolution of project inter-dependencies in the Java subset of the Apache ecosystem, comprising a total of 1,964 releases of 147 projects, for 14 years. First, we investigated how the ecosystem has grown over time in size, number of projects, dependencies between projects, and number of developers. After that, we analyzed several factors that could have influenced the upgrade (or not) of a dependency between a project and a library it uses. Also, we qualitatively investigated, by looking into mailing list and issue report discussions, how developers discussed the opportunity to

[37] API breaking changes would cause an application built with an older version of the component to fail under a newer version.

perform an upgrade and its possible impacts/risks. Finally, we assessed the potential impact of each upgrade in terms of classes using the dependency.

The study results indicated that:

- The ecosystem size exponentially grows over time, and consequentially the dependencies between projects grow too. The number of active developers involved in the project grows until a certain year (2006). Then, it remains stable because most of the new projects (many of which part of the Apache Commons) involve developers already active in the ecosystem). Finally, over the last few years (since 2011) we observe a decrease in the number of active developers. This can be explained because very few projects have been added to the ecosystem during such a period, while the size growth is mainly due to the evolution of existing projects. The latter could, however, concern only some specific features and therefore be performed by a subset of developers only.
- For what concerns the factors that could have influenced dependency upgrades, we found that this does not really depend on project-level characteristics such as project size or fan-in and fan-out. Instead, we found that the kinds of changes performed between a release and the subsequent one likely correlate with the upgrade. That is, on the one hand client projects are more willing to upgrade a library when its new release involve a substantial number of bug fixes. On the other hand, changes to API interfaces tend to discourage upgrades because it can require non-negligible changes to the client source code. These findings have been reflected from the discussions developers had over mailing lists and issue trackers.
- In general, the studied projects have been designed so to have a limited impact (about 5%) from changes in libraries form which they depend. However, some libraries (above all frameworks) have a large impact on their client projects and their upgrades need to be carefully pondered.

This work has mainly an observational nature, i.e., it aimed at empirically investigating a phenomenon—dependency upgrades in a software ecosystem—-from both quantitative and qualitative point-of-view. Nevertheless, there are different possible uses one can make of the results of this paper. First, the paper highlights that the dependency phenomenon has an exponential growth and should therefore carefully be considered by developers contributing to the ecosystem. Second, it provides an overview of possible factors that could influence dependency upgrades, with indications of the role played by such factors in the Apache ecosystem, and of how the main reasons for upgrading or not were discussed by developers.

Future work could start from the observations made in this paper to build recommenders aimed at supporting developers in the complex dependency upgrade decision making activity. Clearly, such recommenders should combine a careful analysis of the aforementioned factors with an analysis of release notes to understand the pros and cons of upgrades. Also, our findings on developers' discussions can represent the first step toward the building of an ontology of concepts that can be used to analyze and categorize discussions. Finally, it is also worthwhile to replicate this study on other ecosystems, possibly also considering factors that were not investigated in this work.

Acknowledgements

Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, and Sebastiano Panichella are partially funded by the EU FP7-ICT-2011-8 project Markos, contract no. 317743. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

References

1. Annosi, M., Di Penta, M., Tortora, G.: Managing and assessing the risk of component upgrades. In: Product Line Approaches in Software Engineering (PLEASE), 2012 3rd International Workshop on, pp. 9–12 (2012)
2. Antoniol, G., Ayari, K., Di Penta, M., Khomh, F., Guéhéneuc, Y.G.: Is it a bug or an enhancement?: a text-based approach to classify change requests. In: Proceedings of the 2008 conference of the Centre for Advanced Studies on Collaborative Research, October 27-30, 2008, Richmond Hill, Ontario, Canada, p. 23. IBM (2008)
3. Aranda, J., Venolia, G.: The secret life of bugs: Going past the errors and omissions in software repositories. In: 31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings, pp. 298–308 (2009)
4. Bavota, G., Canfora, G., Di Penta, M., Oliveto, R., Panichella, S.: The evolution of project inter-dependencies in a software ecosystem: the case of Apache. In: 29th IEEE International Conference on Software Maintenance (ICSM 20013), September 22-28, 2013, Eindhoven, The Netherlands. IEEE (2013)
5. Bavota, G., Ciemniewska, A., Chulani, I., De Nigro, A., Di Penta, M., Galletti, D., Galoppini, R., Gordon, T.F., Kedziora, P., Lener, I., Torelli, F., Pratola, R., Pukacki, J., Rebahi, Y., Villalonga, S.G.: The market for open source: An intelligent virtual open source marketplace. In: Joint 18th European Conference on Software Maintenance and Reengineering / 21st Working Conference on Reverse Engineering, CSMR18/WCRE21, February 3-6, 2014, Antwerp, Belgium, Proceedings, p. To appear (2014)
6. Bosh, J.: From software product lines to software ecosystems. In: Proceedings of the 13th International Conference on Software Product Lines (SPLC), pp. 111–119 (2009)
7. Businge, J., Serebrenik, A., van den Brand, M.: Survival of Eclipse third-party plug-ins. In: 28th IEEE International Conference on Software Maintenance (ICSM 2012), Trento, Italy, Sep 23-28, 2012, pp. 368–377. IEEE Computer Society (2012)
8. Cohen, J.: Statistical power analysis for the behavioral sciences, 2nd edn. Lawrence Earlbaum Associates (1988)
9. Collard, M.L., Kagdi, H.H., Maletic, J.I.: An xml-based lightweight c++ fact extractor. In: 11th International Workshop on Program Comprehension (IWPC 2003), May 10-11, 2003, Portland, Oregon, USA, pp. 134–143. IEEE Computer Society (2003)
10. Conover, W.J.: Practical Nonparametric Statistics, 3rd edition edn. Wiley (1998)
11. Dagenais, B., Robillard, M.P.: Recommending adaptive changes for framework evolution. In: 30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, pp. 481–490. ACM (2008)
12. Di Penta, M., Germán, D.M., Guéhéneuc, Y.G., Antoniol, G.: An exploratory study of the evolution of software licensing. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010, pp. 145–154. ACM (2010)
13. Dig, D., Johnson, R.: How do APIs evolve? a story of refactoring. Journal of Software Maintenance and Evolution: Research and Practice **18**, 83–107 (2006)

14. Gala-Perez, S., Robles, G., Gonzalez-Barahona, J.M., Herraiz, I.: Intensive metrics for the study of the Evolution of Open Source Projects. In: 10th IEEE Working Conference on Mining Software Repositories. San Francisco, California, USA (2013)
15. German, D., Adams, B., Hassan, A.E.: Programming Language Ecosystems: the Evolution of R. In: Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR), pp. 243–252. Genova, Italy (2013)
16. Germán, D.M.: The GNOME project: a case study of open source, global software development. Software Process: Improvement and Practice **8**(4), 201–215 (2003)
17. German, D.M., Gonzalez-Barahona, J.M., Robles, G.: A model to understand the building and running inter-dependencies of software. In: Proceedings of the 14th Working Conference on Reverse Engineering, WCRE '07, pp. 140–149. IEEE Computer Society, Washington, DC, USA (2007)
18. German, D.M., Manabe, Y., Inoue, K.: A sentence-matching method for automatic license identification of source code files. In: Proceedings of the IEEE/ACM international conference on Automated software engineering, ASE '10. ACM, New York, NY, USA (2010)
19. Godfrey, M.W., Tu, Q.: Evolution in open source software: A case study. In: Proceedings of the International Conference on Software Maintenance (ICSM'00), pp. 131–140. IEEE Computer Society, Washington, DC, USA (2000)
20. Goeminne, M., Claes, M., Mens, T.: A historical dataset for the GNOME ecosystem. In: Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, pp. 225–228. IEEE Press, Piscataway, NJ, USA (2013)
21. Goeminne, M., Mens, T.: Analyzing ecosystems for open source software developer communities. In: M.A.C. Slinger Jansen Sjaak Brinkkemper (ed.) Software Ecosystems: Analyzing and Managing Business Networks in the Software Industry, pp. 301–329. Edward Elgar Publishing, Incorporated (2013)
22. Gonzalez-Barahona, J.M., Robles, G., Michlmayr, M., Amor, J.J., German, D.M.: Macro-level software evolution: a case study of a large software compilation. Empirical Softw. Engg. **14**(3), 262–285 (2009)
23. Grissom, R.J., Kim, J.J.: Effect sizes for research: A broad practical approach, 2nd edition edn. Lawrence Earlbaum Associates (2005)
24. Hazewinkel, M.: KolmogorovSmirnov test. Springer (2001)
25. Hou, D., Yao, X.: Exploring the intent behind api evolution: A case study. In: 18th Working Conference on Reverse Engineering (WCRE'11), Limerick, Ireland, Oct 17-20, 2011, pp. 131–140 (2011)
26. Jansen, S., Finkelstein, A., Brinkkemper, S.: A sense of community: A research agenda for software ecosystems. In: 31st International Conference on Software Ecosystems, New and Emerging Research Track, pp. 187–190 (2005)
27. Jergensen, C., Sarma, A., Wagstrom, P.: The onion patch: migration in open source ecosystems. In: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, pp. 70–80. ACM, New York, NY, USA (2011)
28. Kidane, Y.H., Gloor, P.A.: Correlating temporal communication patterns of the Eclipse open source community with performance and creativity. Comput. Math. Organ. Theory **13**(1), 17–27 (2007)
29. Kilamo, T., Hammouda, I., Mikkonen, T., Aaltonen, T.: From proprietary to open sourcegrowing an open source ecosystem. Journal of Systems and Software **85**(7), 1467 – 1478 (2012)
30. Koch, S., Schneider, G.: Effort, cooperation and coordination in an open source software project: GNOME. Information Systems Journal **12**(1), 27–42 (2002)
31. Krinke, J., Gold, N., Jia, Y., Binkley, D.: Cloning and copying between GNOME projects. In: J. Whitehead, T. Zimmermann (eds.) 2010 7th IEEE Working Conference on Mining Software Repositories, MSR 2010, pp. 98–101. IEEE
32. Levenshtein, V.: Binary codes capable of correcting deletions, insertions, and reversals. Soviet Physics Doklady **10**, 707–716 (1966)
33. Lungu, M., Robbes, R., Lanza, M.: Recovering inter-project dependencies in software ecosystems. In: In Proceedings of ASE 2010, pp. 309–312. ACM Society Press (2010)

34. Mens, T., Fernández-Ramil, J., Degrandsart, S.: The evolution of Eclipse. In: 24th IEEE International Conference on Software Maintenance (ICSM 2008), September 28 - October 4, 2008, Beijing, China, pp. 386–395. IEEE (2008)
35. Ossher, J., Bajracharya, S.K., Lopes, C.V.: Automated dependency resolution for open source software. In: Proceedings of the 7th International Working Conference on Mining Software Repositories, MSR 2010 (Co-located with ICSE), Cape Town, South Africa, May 2-3, 2010, Proceedings, pp. 130–140. IEEE (2010)
36. Raemaekers, S., van Deursen, A., Visser, J.: Measuring software library stability through historical version analysis. In: 28th IEEE International Conference on Software Maintenance (ICSM'12), Trento, Italy, Sep 23-28, 2012, pp. 378–387 (2012)
37. Robbes, R., Lungu, M., Röthlisberger, D.: How do developers react to API deprecation?: the case of a smalltalk ecosystem. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, pp. 56:1–56:11. ACM, New York, NY, USA (2012)
38. Scacchi, W., Alspaugh, T.A.: Understanding the role of licenses and evolution in open architecture software ecosystems. Journal of Systems and Software **85**(7), 1479–1494 (2012)
39. Singh, P.V.: The small-world effect: The influence of macro-level properties of developer collaboration networks on open-source project success. ACM Trans. Softw. Eng. Methodol. **20**(2), 6:1–6:27 (2010)
40. Vasilescu, B., Serebrenik, A., Goeminne, M., Mens, T.: On the variation and specialisation of workload A case study of the Gnome ecosystem community. Empirical Software Engineering pp. 1–54 (2013)
41. Watts, D.J., Strogatz, S.H.: Collective dynamics of'small-world'networks. Nature **393**(6684), 409–10 (1998)
42. Wermelinger, M., Yu, Y.: Analyzing the evolution of Eclipse plugins. In: Proceedings of the 2008 international working conference on Mining software repositories, pp. 133–136. ACM, New York, NY, USA (2008)
43. Wermelinger, M., Yu, Y., Lozano, A., Capiluppi, A.: Assessing architectural evolution: a case study. Empirical Software Engineering **16**(5), 623–666
44. Yu, L., Ramaswamy, S., Bush, J.: Software evolvability: An ecosystem Point of View. IEEE International Workshop on Software Evolvability **0**, 75–80 (2007)
45. Zar, J.H.: Significance testing of the spearman rank correlation coefficient. Journal of the American Statistical Association **67**(339), pp. 578–580 (1972)

View publication stats