Looking at changes in popularity in the Maven ecosystem

Michel Bulten
Supervisor(s): Mehdi Keshani, Sebastian Proksch
EEMCS, Delft University of Technology, The Netherlands

June 19, 2022

A Dissertation Submitted to EEMCS faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering

Abstract

We look at the Maven eco-system and how popularity of packages and its methods change. We want to know if there are any trends that can help developers more efficiently use their time. To look at the popularity we do package analysis and method analysis. We find that there is no correlation between time and popularity on both the package and the method level. We do seem to find that developers are slow to adopt newer versions. We also found that library maintainers tend to re-release older versions, these versions often have low usage.

1 Introduction

No one likes to reinvent the wheel, and neither do developers. Developers can reuse code by manually downloading and adding the relevant source files to their projects. A better, less time-consuming way to do this is using build tools like Maven or Gradle. These build tools take care of so-called dependency management. By adding an artifact ID to a dependency list, build tools will resolve the artifact IDs into relevant JAR files. Repositories store these artifacts, and because these artifacts are immutable, changes to a library require a new artifact to be released. The immutability policy makes the Maven ecosystem interesting to research, as it allows us to see what happens to libraries over time.

Some previous studies[2, 7] have researched the Maven ecosystem before Benelallam et al. [2] looked at the emergence of diversity in library usage, and Soto-Valero et al. [7] presented a Maven Dependency graph enabling us to analyze the Maven ecosystem. Some other studies, like Decan et al. [3] concluded that dependency networks are growing over time, but only a minority of artifacts are in use.

It is often the case that library maintainers have to maintain many different APIs simultaneously. The larger the size of maintained APIs, the more work for these library maintainers. We know from Harrand et al. [4] that usage of APIs concentrates around a small set of APIs. It is currently impossible for library maintainers to know which methods are popular and which are not, so they have no idea what to focus their time on, which can be inefficient.

We aim to show how libraries' popularity and its methods change over time. Looking at how the popularity of packages changes over time might give an idea on what kind of packages are popular and if there is a trend in popularity. Knowing how the popularity of packages changes may not be sufficient as this only gives a broad view. To get a clearer picture, it is also important to look at how the popularity of methods within a package changes over time. We hope to find useful information that can help library maintainers focus on a smaller size of APIs.

To further help developers sustain popularity, we like to know how the popularity of methods within a package changes over time. Answering this question can show if popular methods tend to stay popular across versions. If they decline, what the reason could be, and if their popularity increases, what might cause it. It might be that a method declined in popularity due to breaking changes, an alternative better way to achieve the same result, or because of known previous vulnerabilities.

Before we analyze method-level popularity, we sample a set of packages to analyze. We use random weighted sampling to sample the set of packages for analysis. The weight of our sampling technique is on the number of unique dependents a package has. We do this as these packages will most affect the Maven Ecosystem.

We analyze packages by looking at how their popularity changes over time and how it changes depending on the number of versions. Furthermore, we analyze methods by looking at how the maximum relative popularity of methods changes over time and how the median of these values changes over time.

We use the FASTEN project to analyze the packages at the method level. The FASTEN project allows us to generate call graphs, which contain the interaction between different methods. Analyzing these graphs, we can develop a popularity metric, which we will use to see how the popularity of methods changes over time. We hope to find critical takeaways to help developers efficiently maintain their libraries.

We find that there is no correlation between time and popularity on both the package and method level. What we do see is that some packages are tightly coupled, the same is the case for methods. It might also not make sense to have a high frequency of releases as developers do not seem to update their dependencies as frequent

We structure this paper into six chapters, including the introduction. In the second chapter, we discuss the background and related works, we move to the third chapter explaining our approach and methodology to tackle this research. We then move to the results in chapter four. In chapter 5 we discuss our results and also reflect on the ethical aspects of this research. We conclude with the sixth chapter talking through our conclusion.

2 Background

In this section I will go through the main concepts and explain some related works of this project.

Main Concepts

Dependents and Dependencies Dependencies are artifacts on which an artifact dependents on. For example, when artifact A requires code from another artifact B, it must declare B as its dependency, making A a dependent of B.

Maven is a popular build tool that automates and streamlines project setup. Maven uses a POM file, where developers can declare project specifics.

Artifact will have the form of *groupId:artifactId:version* where GroupId refers to all projects created by the same groups, artifactId is the name of the JAR without version number, and version is the particular version.

Repository is a location where Maven Artifacts are stored. Maven central is one of the largest repositories.

Application Programming Interface APIs for short are an interface that allows users to call specific applications. APIs come in a lot of different shapes. Sometimes APIs are used in the context of communicating between servers, or APIs are used for a client to be able to interact with a server. In this study, we will use APIs in the context of public methods.

FASTEN Project is a collaborative project with the intention to enhance robustness and security in software ecosystems. It helps shed light on things not directly apparent to library maintainers.

Uniform Resource Identifier is used in the generated call graphs to refer to nodes uniquely. These Uniform Resource Identifiers, which we will shorten to URI, can determine which artifact a call originates from or the calling method. URI as formatted in the following way:
`fasten://mvn![Artifact]/[Package]/[Class].[Method]([Method parameters])[Return Type]`

Call Graph is a directed graph which in this case is generated by the FASTEN project. This call graph includes a set of nodes that each have a unique number linked to a URI and also includes the directed edges representing the calls made between the nodes. Reading this graph can show us the callers of a particular API, which helps in calculating its popularity. It is possible to include transitivity in the generation of these call graphs.

Transitive dependencies are not directly declared but are the dependencies of one of the package dependencies, hence *transitive* as it does not require a developer to explicitly declare these packages as dependencies.

Related Works

Xavier et al. [10] analyzed the historical impact of API breaking changes as software often changes to accommodate new features, fix bugs, or refactor code. However, these changes can cause previous code to break, impacting its users. This study looks at the frequency of breaking changes, their behavior over time, the impact on users, and the characteristics of libraries with a high frequency of breaking changes. It found a high rate of breaking changes as on the median, 14.78% of changes break contracts with its users. These breaking changes mainly occur on the method level. The paper also found that the frequency of breaking changes increases over time and that more extensive or active libraries have a higher chance of breaking changes. However, these breaking changes do not impact the user that often, as the study found that only 2.54% of its client were impacted by breaking changes. Because this study also looks at changes over time, it is relevant to our work. The difference, however, is that they looked at breaking changes.

Zapata et al. [11] is a small case study examining how developers react to known vulnerable dependencies. It does this because other studies have shown that developers struggle to keep up with the updates and keep using vulnerable libraries. Most of those studies do not look at the method level. This study shows that 73.3% of clients using vulnerable dependencies are not running vulnerable code, which confirms that analysis at the library level is overestimated and further method-level analysis is needed. However, further research is necessary as this study only looked at 60 projects.

Lima et al. [6] looked into the characteristics of popular APIs. The study makes the distinction between popular, ordinary, and unpopular. To find out what makes an API popular, it looks at numerous characteristics. It considers the following characteristics: documentation, stability, changeability, complexity, and legibility. These characteristics could give a picture of how best to implement and maintain APIs.

The study sees that popular APIs often have more public methods and more lines of code but no change in relative lines of code between the three categories of APIs, which means that comments are not a factor in popularity. If we look at complexity, we see that popular APIs tend to be more complex than ordinary or unpopular libraries. This is also the case if we consider the relative complexity per method. Method name length and amount of parameters do not seem to be factors in the popularity. Popular APIs also change frequently and have more contributors than ordinary or unpopular APIs. Another thing they noted is that popular APIs are often used early on in the development cycle, while unpopular ones are adopted later on.

The conclusion is that popular APIs are different from ordinary ones. They have more changes, are less stable, and used earlier in the development cycle than non-popular APIs. Knowing this can be useful to relate to our study as we look at the popularity of APIs overtime.

Alrubaye et al. [1] look into automating migration to another library as this requires developers to have a thorough knowledge of both libraries, which means it can be a timely procedure. They propose a mining approach by looking at function-change patterns with function-related lexical similarity to map replacing functions accurately. Migration to other libraries can impact the popularity of methods over time, so this study is related.

Hejderup et al. [5] propose a fine-grained dependency network beyond looking at the package level and into call graphs. Newly built libraries often use already existing libraries, leading to a highly interconnected ecosystem, which can lead to trust and security implications. This is even more apparent in recent issues at Equifax, where not updating a vulnerable dependency caused the leakage of over 100.000 credit card records. The paper outlines constructing the proposed graph, which will make it more clear what impact dependencies have. A more fine-grained dependency graph can help see the spread of a security bug and possibly have prevented the Equifax problem. The FASTEN project, which we use for our study, can generate these dependency graphs, and reading this study could lead to more in-depth details on how to generate these graphs.

Harrand et al. [4] look at the two well-documented intuitions about client-library dependencies, as they seem contradictory. The contradiction is that, on the one hand, Hyrum's law states that with sufficient users, all APIs will have users. On the other hand, recent research concluded that APIs are unnecessarily extensive. This study concludes that there is a continuum between these two intuitions. With sufficient clients, every API will have a client, but the majority of clients will use the same small set of APIs. Knowing that clients will only use a small set of APIs means developers could focus their time more efficiently. We will look into which methods are the most popular and how and why their popularity changes over time, indicating what developers should focus their time on.

3 Approach

In this section, we discuss our approach to answering our research question. We first will explain our research question, then go through our data specification before concluding with our methodology for answering our questions.

3.1 Research Questions

Our primary research question is how the popularity of library methods changes over time. To answer this question, we divided the question into the following:

RQ1: How does the popularity of packages change This question will be answered by looking at how the popularity of packages changes over time and over the different versions. We will also do a linear regression to see if there is any trend between time and popularity.

RQ2: How does the popularity of methods change We want to look at libraries in the Maven ecosystem and how the popularity of their methods changes over time. We intend to do this by looking at the relative percentage of methods used per package over time, we also look at a median line between all the packages to see if there is any trend. We end with looking into the distribution of popularity per month.

3.2 Data Specification

The Maven ecosystem has over 28 million artifacts, to analyze this we need to decide on a sample size. For processing artifacts, it is necessary to include all dependents and their

dependencies. To be able to do this we will need to limit ourselves to a small set of artifacts, we decided upon a confidence level of 95% and a maximum margin of error of 5%, which according to Taherdoost [9], means a sample size of 384. Because we wanted the individual research together to form a coherent story, we decided first to sample 384 unique packages used in each research. Using these shared 384 unique packages, we would choose a smaller set of packages but include all their versions.

Data Selection In this study, we will be using the FASTEN project running on one of TU Delft's servers, we will only be able to analyze ingested packages. We chose a time frame of 6 months starting in October 2021 as FASTEN has the most recently released artifacts. We exclude testing frameworks from our selection, as we only analyze source code which should not include testing. Large library maintainers often split their projects into multiple smaller artifacts, having many dependencies amongst artifacts within a group, potentially skewing our data. To prevent these libraries from skewing our data, we only kept a single artifact per groupId in the dependents. We retrieve 10 thousand unique packages from FASTEN.

Sampling Technique Taherdoost [8] proposes several techniques and discusses their advantages and disadvantages. We decided on a sampling technique that we consider a combination of two techniques, *simple random sampling* and *judgemental sampling*. In the simple random sampling technique, every package has the same chance of getting picked. In judgemental sampling, we purposefully select packages for research that we think would be more relevant to our research. The chosen technique is *weighted random sampling*. The weight is the number of dependents a package has, as we believe that packages with a higher number of dependents will also have a higher impact on the Maven ecosystem. We found that out of our remaining 10 thousand unique packages, about 7500 did not have any dependents and thus cannot be selected by our sampling, leaving approximately 2500.

Final selection After doing the weighted random sampling, we end up with 384 unique packages, and this will be our starting point to derive multiple versions. We find out that from these 384 packages, one is duplicated because of a change in groupId, meaning that we have 383 unique packages. These 383 unique packages have 1977 artifacts between their release and the 31st of March. As the first version of a package might be anywhere in our time frame, we decided to limit our selection to packages with at least one release in October. Leaving 165 packages remaining.

For method-level analysis, we need an even smaller sample as we will not be able to analyze all packages and their versions in a reasonable time. We sample this smaller sample by doing random sampling.

3.3 Methodology

To answer how the popularity of library methods changes over time, we will undertake three main steps: call-graph generation, package analysis and method analysis. We will outline in detail how we did this.

Call-graph generation We generate a call graph for each artifact that we want to analyze. To generate a call graph, we need to resolve the dependents of the target artifact. Resolving dependents requires us to resolve potential dependents and confirm them by resolving their dependencies. From the generated call graph, we then remove all the internal edges and edges not going towards our target artifact. The nodes being uniquely identifiable enables us to combine calls to specific methods.

Package analysis We look at the unique dependents per artifact. To compare popularity amongst different packages, we came up with a relative dependent metric seen in figure 1. Where the totalDependents is the summation of each version dependents. For analysis, we look at how the popularity of packages changes over time, how this differs per number of releases in our time-frame and we conduct a linear regression analysis to see if time and popularity correlate.

$$dependentMetric = totalDependents/versionDependents$$

Figure 1: Metric used to normalize the dependents of artifacts

Method analysis After generating the call graphs, we can do method-level analysis, as we are able to see the popularity of each method within a package. We define the popularity of methods by the number of unique dependents calling the method. We call this callCount. What we think could be interesting is how the popularity of methods within a package changes over time. Figure 2 shows this metric. We divide the calls a method receives by the summation of the number of calls all the methods within an artifact receive. To look and see if there is a trend we will use the max value of the metric per version and draw the lines on a graph, to make it easier to see if there is a trend we will use the median of all lines, we conclude by doing the Mann Kendall Test as this is can help identify trends in time-series.

$$callMetric = (callCount/sumOfCallCount) * 100$$

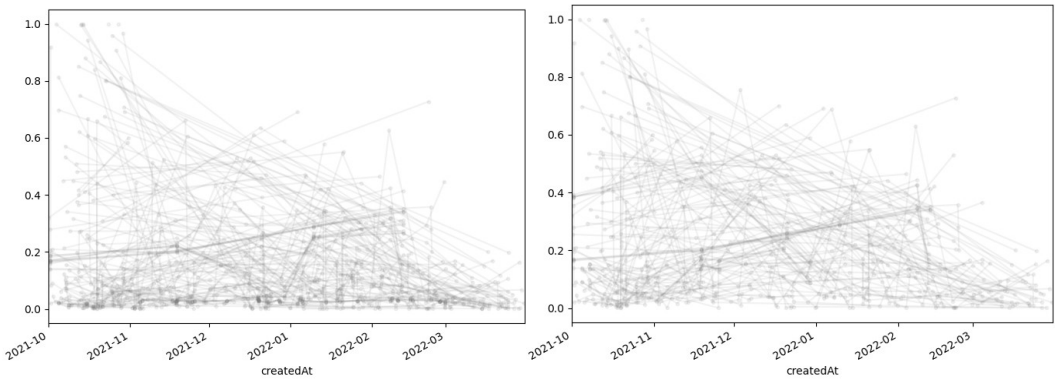
Figure 2: Metric used to analyze methods

4 Results

Package Analysis

To choose what packages we want to analyze at the method level, we first look at the popularity of packages over time. Because the amount of dependents varies heavily between packages, we normalize the data. Figure 3a shows the original data. On the y-axis we see the dependent metric and on the x-axis we see the release date of the artifacts. From this data, we noticed that developers tend to re-release older versions of their packages to the Maven Ecosystem. These releases seem to have a low amount of dependents. In figure 3b

we see the graph with these cases filtered out. We did the filtering by looking at subsequent version numbers.



(a) Before (b) After

Figure 3: Comparison of subsequent version filtering

Figure 4 shows the popularity per version. The y-axis being the dependentMetric and the x-axis being the quartiles. There seem to be no clear trends. Some packages do seem to be correlated.

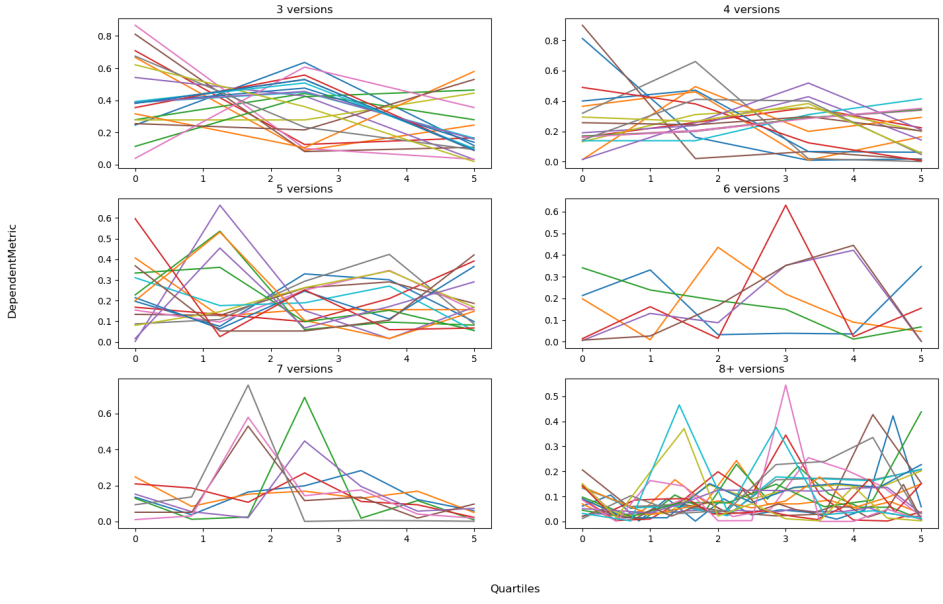


Figure 4: Popularity per amount of versions released

As it seems like the number of dependents lowers over time. We decided to do a linear regression on the dependent count and the timestamp. This is seen in figure 5. Although

the linear regression line seems to have a slight downwards trend, it is not significant as the P-value is 0.1866.

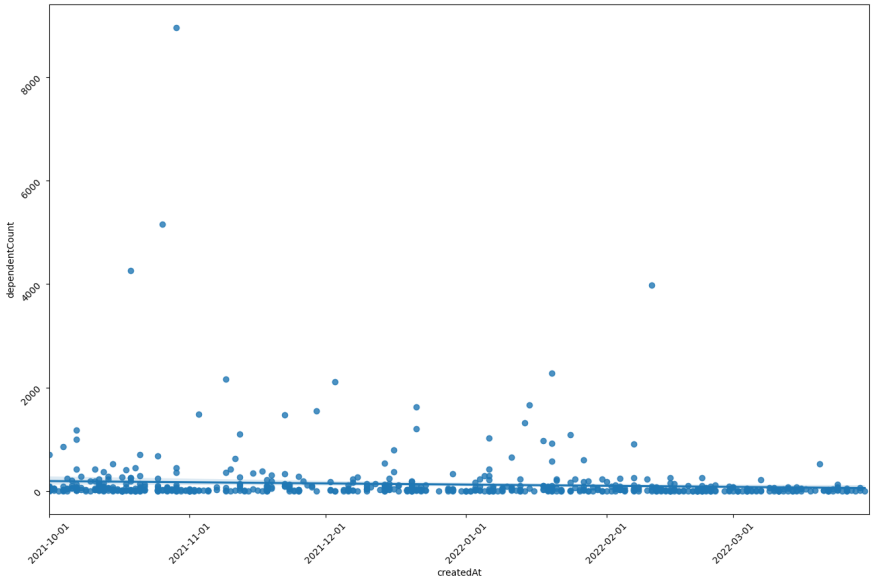


Figure 5: Lineair regression

Method Analysis

We drew a graph using the relative percentage metric described in our approach. We did this for each package but used the alpha value of 0.2. We also drew a median line, as it could be useful to identify any trends. This graph is seen in figure 6, where the y-axis is the timestamp and the x-axis is the percentage of methods used per artifact. The points for the median line are interpolated per week. It seems that there is no trend over time, to confirm this we ran a Mann Kendall Test, as this is useful to see trends in time-series. The test concluded that there is no trend, confirming our assumption.

We also thought it could be interesting to look at the popularity distribution of the max call count per month. This is seen in figure 7. Where the x-axis is the percentage of the maximum call count per artifact.

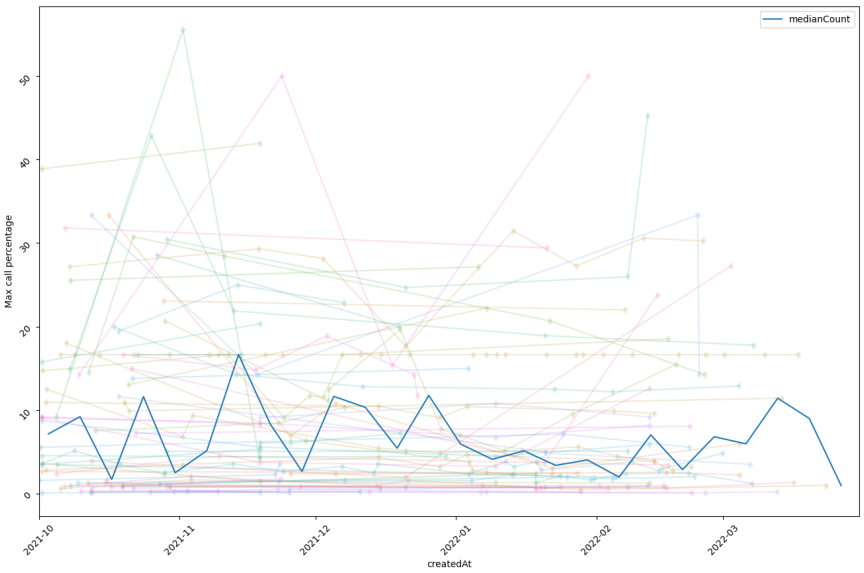


Figure 6: Graph showing the max percentage of methods used per package and the median line

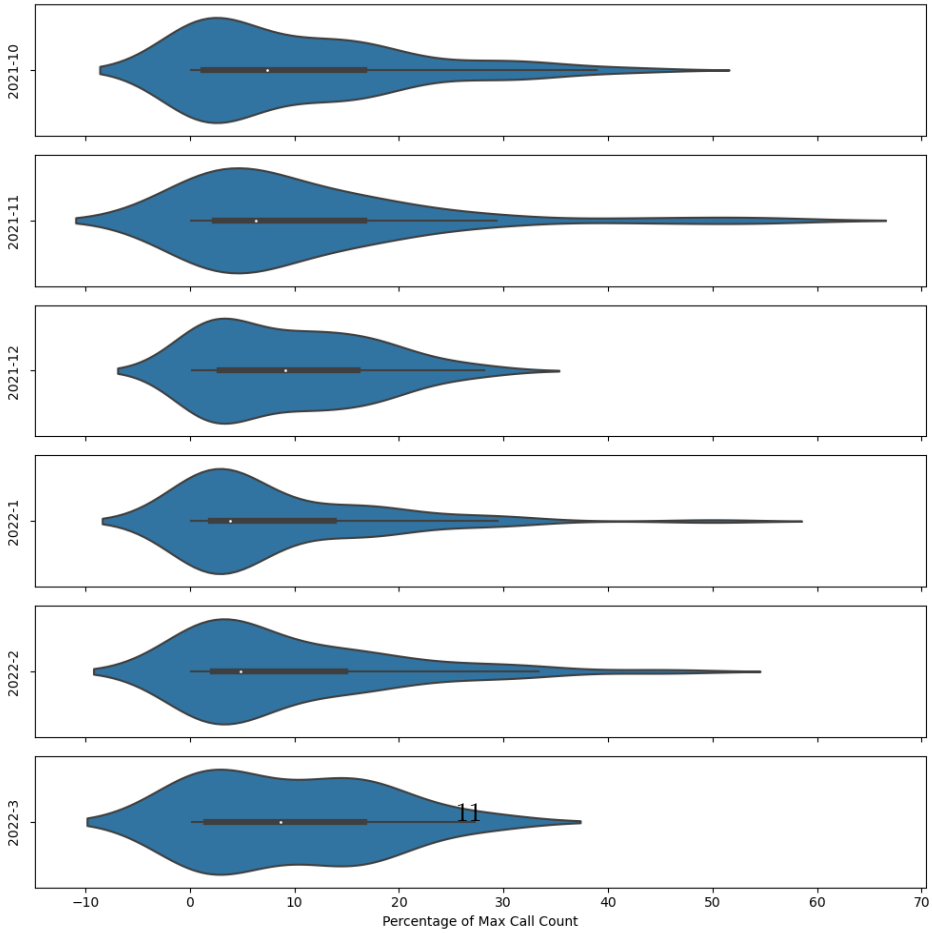


Figure 7: Graph showing the distribution per month

5 Discussion

Developers do not always directly release their newer versions onto public repositories. Some eventually do, but because newer versions are released, these older versions do not get many dependents. We can see a reduction of low-dependent artifacts in the filtered figure. It makes sense that developers looking to update will not opt for older versions.

Although not significant, it seems to be the case that newer releases have fewer dependents than older ones. It could be the case that developers wait a while after the release of a version, to decrease the chance of the new version having major bugs requiring a patch. This makes sense as developers do not want to spend too much time changing dependencies. Developers likely only update their dependencies when needed, for example, by new functionality.

What we also saw from our data, which makes sense, is that packages with frequent releases will also have fewer dependents per version. It might be better for developers to reduce the frequency of updates as developers do not tend to update their dependencies frequently. It might confuse other developers as it is less clear which versions are stable. Looking at GSON, for example, the package with the most dependents in our sample, only has two releases in our time frame.

We also looked at releases including "Alpha", "Beta" or "M" in their version tag to verify the idea that developers want to have some certainty in stability when choosing a version. If developers want some certainty that a library is stable, the dependent count of releases with a pre-release tag should be low. We drew a graph in our appendix A of packages with more than one pre-release version in our time frame, and except for the *drools-core-reflective* package, it does indeed seem to be the case that pre-release versions of packages have a low dependent count.

What is interesting in figure 4 is that some packages follow the same popularity trend. We found that library maintainers do not always use the group id to identify the organization. It can identify a group within an organization or a module, which is why it is in our sample, as the groupId differed. We see cases where the functionality of a package is extended by another package, and other developers often use these together.

By doing the Mann Kendall Test we can conclude that there is no trend over time. What we can derive from the graph however is that the maximum percentage of calls for each version on the median is 10%. This can be skewed by the number of methods each version has, to compensate for this we also made a normalized graphs seen in 10. Normalization is done by multiplying the max percentage metric by the amount of methods a version has. Here we also do not see a trend. If we look more in depth into the method data, we do see that there are coupled methods, which are always used together.

5.1 Threats to validity

In our study, we opted to sample our data by doing weighted random sampling on the number of dependents an artifact has, using a time frame of 6 months, starting in October. We chose this time frame because not all artifacts have been ingested by the FASTEN project yet. However, this time frame might be too short to identify major noticeable trends, or a trend can be short-lived.

Filtering is done by only including a package per groupId and ensuring that the package names do not include some keywords we defined. This does not guarantee that all testing frameworks are filtered out. We also saw a duplicate of a package. Because of a changed

groupId, we should have also filtered by package name as it is unlikely for packages with identical names to be different. It might also be necessary to filter out packages that add code during compile time, like Lombok which in itself does not have any method calls, but a lot of dependents.

FASTEN continuously processes artifacts from Maven repositories, which is needed to generate call graphs and analyses. The FASTEN project has processed only 30% of artifacts in our time frame. Because the number of processed artifacts is low, we are also limited by the packages we can analyze. The FASTEN project may be skewed towards packages with a low number of dependents, as these artifacts require less processing time.

Another threat to validity is that a lot of programming is involved in analyzing these packages. There can be bugs in our own codebase or even in FASTEN skewing our results.

5.2 Future Research

In our threats to validity, we mention that the time frame might have been too short to see any noticeable changes. For future research, we propose to take the 384 unique packages with the most dependents in a chosen year and look back on their entire history to see how their dependent count changes over time. Hopefully, analyzing this history can help identify trends in popularity for specific libraries, helping developers understand what is important for growth. Another proposal is to look at the unique packages that used to be popular a few years ago but declined in popularity, which can help identify why packages might lose their popularity. We also suggest trying to categorize packages by their type of usage. It might also be interesting to see how transitivity impacts the popularity of methods.

5.3 Responsible Research

Ethics Our research focuses on looking at trends in popularity. We do not look at how it impacts developers. Which means that there is low ethical risk.

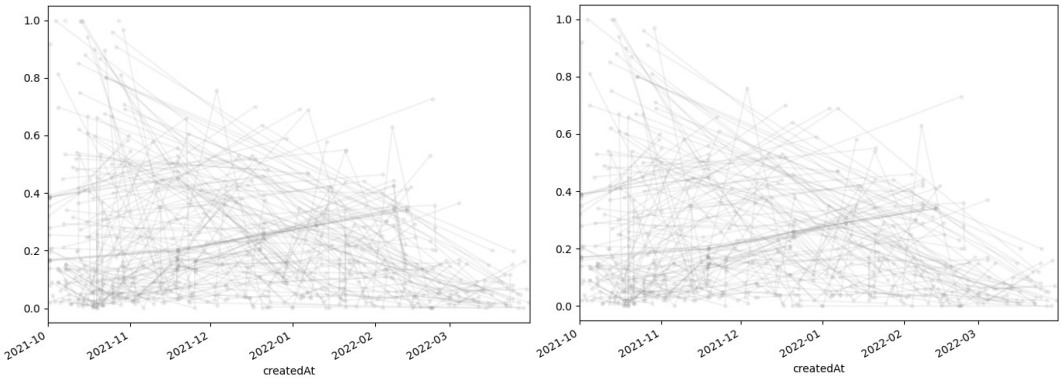
Reproducible We aim to make our work reproducible, we described our approach in detail in chapter 3. To allow other researcher to use the same data set we decided to upload both our code and a zip with the used data-set on Github. The dataset includes the following: dependents, dependencies of dependents, popularity of methods per artifact and the actual call graph for each artifact. The github page is: https://github.com/MJGBulten/FASTENResearch

6 Conclusion

This study looks at how packages and methods' popularity changes over time. We hoped to find useful information to help developers allocate their time. We looked at the package level and did not find a correlation between time and popularity. What we did find on the package level is that older versions of packages get a re-release on public repositories and have a low number of dependents as newer versions are already available. We also saw that some packages are strongly coupled, meaning that one is not used without the other. Although there was no correlation between time and popularity, it seemed that newer versions had fewer dependents than older ones. We combine this with the fact that Alpha, Beta, and Milestone releases also have a low number of dependents, which can hint that developers try to use stable releases and not always the newest ones. We also did not see a trend over time

at the method level. The concentration of methods used seems to remain the same, which on the median is 10%. For future research, we suggest using a larger time frame and trying to categorize packages and methods. Doing so might give developers help full insights into what is important to sustain popularity.

A Tables and figures



(a) Before (b) After

Figure 8: Comparison between original data and filtered out Alpha, Beta or Milestone releases

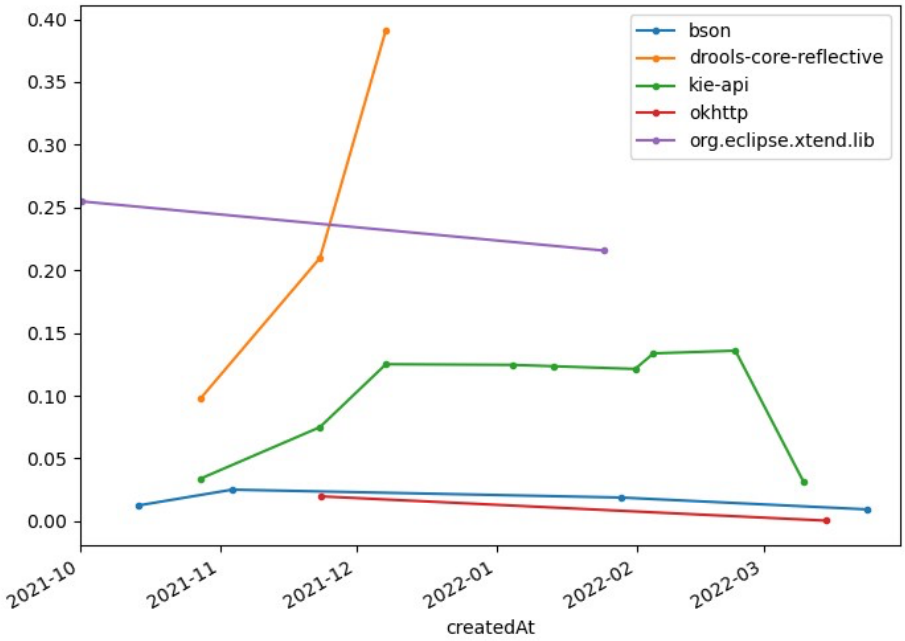


Figure 9: Only pre-release tag versions

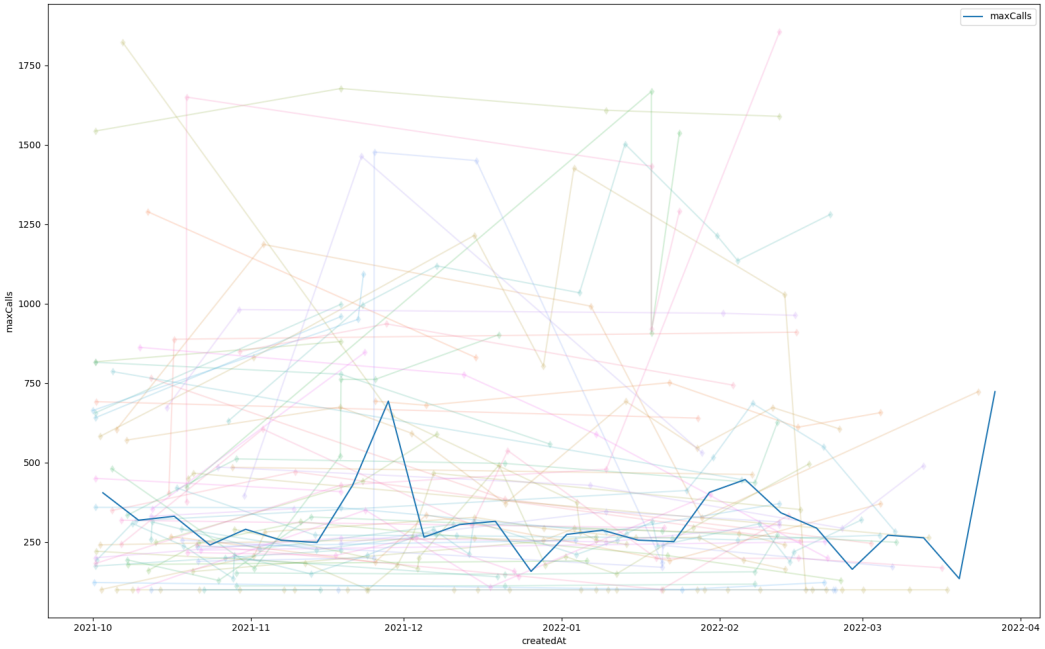


Figure 10: Normalized percentage by number of methods

References

[1] Hussein Alrubaye and Mohamed Wiem Mkaouer. "Automating the detection of third-party Java library migration at the function level." In: *CASCON*. 2018, pp. 60–71 (cit. on p. 5).

[2] Amine Benelallam, Nicolas Harrand, César Soto-Valero, Benoit Baudry, and Olivier Barais. "The maven dependency graph: a temporal graph-based representation of maven central". In: *IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)* (2019) (cit. on p. 2).

[3] Alexandre Decan, Tom Mens, and Philippe Grosjean. "An empirical comparison of dependency network evolution in seven software packaging ecosystems". In: *Empirical Software Engineering* 24.1 (2019), pp. 381–416 (cit. on p. 2).

[4] Nicolas Harrand, Amine Benelallam, César Soto-Valero, François Bettega, Olivier Barais, and Benoit Baudry. "API beauty is in the eye of the clients: 2.2 million Maven dependencies reveal the spectrum of client–API usages". In: *Journal of Systems and Software* 184 (2022), p. 111134 (cit. on pp. 2, 6).

[5] Joseph Hejderup, Arie van Deursen, and Georgios Gousios. "Software ecosystem call graph for dependency management". In: *2018 IEEE/ACM 40th International Conference on Software Engineering: New Ideas and Emerging Technologies Results (ICSE-NIER)*. IEEE. 2018, pp. 101–104 (cit. on p. 6).

[6] Caroline Lima and Andre Hora. "What are the characteristics of popular APIs? A large-scale study on Java, Android, and 165 libraries". In: *Software Quality Journal* 28.2 (2020), pp. 425–458 (cit. on p. 5).

[7] César Soto-Valero, Amine Benelallam, Nicolas Harrand, Olivier Barais, and Benoit Baudry. "The emergence of software diversity in maven central". In: *IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)* (2019) (cit. on p. 2).

[8] Hamed Taherdoost. "Sampling methods in research methodology; how to choose a sampling technique for research". In: *How to Choose a Sampling Technique for Research (April 10, 2016)* (2016) (cit. on p. 7).

[9] Hamed Taherdoost. "Determining sample size; how to calculate survey sample size". In: *International Journal of Economics and Management Systems* 2 (2017) (cit. on p. 7).

[10] Laerte Xavier, Aline Brito, Andre Hora, and Marco Tulio Valente. "Historical and impact analysis of API breaking changes: A large-scale study". In: *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2017, pp. 138–147 (cit. on p. 5).

[11] Rodrigo Elizalde Zapata, Raula Gaikovina Kula, Bodin Chinthanet, Takashi Ishio, Kenichi Matsumoto, and Akinori Ihara. "Towards Smoother Library Migrations: A Look at Vulnerable Dependency Migrations at Function Level for npm JavaScript Packages". In: () (cit. on p. 5).