Mining repositories to analyze the lifecycle of frameworks and libraries

Ronaldo Rubens Gesse Júnior
Department of Computing
School of Sciences, São Paulo State University (UNESP)
Bauru, SP, Brazil
ronaldo.rubens@unesp.br

Higor Amario de Souza
Department of Computing
School of Sciences, São Paulo State University (UNESP)
Bauru, SP, Brazil
higor.amario@unesp.br

ABSTRACT

In a constantly evolving technological landscape, it is crucial to choose the right components and technologies for a software project to ensure its successful development. Frameworks and libraries are essential components that provide functionality to the code and accelerate the development process. They assist teams in delivering results more efficiently to the end user through software reuse. This work proposes using Mining Software Repositories (MSR) to analyze the lifecycle of frameworks and libraries. We aim to understand whether a framework or library is properly updated, maintained, and sought after by the community, which may indicate its lifecycle stage. We explored data from several open source projects: the number of commits and contributors over time. Also, we are using data from Google Trends to explore the developer community's interest in such libraries and frameworks. We are using a trend metric – Exponential Moving Average (EMA) – over the prior mentioned variables to indicate the lifecycle stage of such frameworks and libraries. The initial results show that our approach can distinguish lifecycle trends for frameworks within the same domain. Our future research will involve examining additional MSR data (such as pull requests, issues, and code changes), obtaining other data sources (Q & A sites), and applying time series Machine Learning techniques to improve the analysis.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; **Development frameworks and environments**; **Software evolution**; **Maintaining software**; **Software libraries and repositories**; **Software maintenance tools**; **Reusability**.

KEYWORDS

Mining software repositories, software reuse, frameworks, libraries

1 INTRODUCTION

The software development process has become more complex over the years. Software reuse aims to reduce the development effort by providing generic code structures and specific functions for common software needs. Frameworks can be defined as code structures of all or part of a system that must be implemented when building software systems [10]. Those structures are related to an architectural domain (e.g., web, mobile). A software library is a collection of strictly related code functions from a specific domain ready for reuse in other software projects. Some examples are libraries for building maps, parsing file formats, or building machine learning models. Libraries are functionalities that can be directly called in the code, whilst frameworks are characterized by an inverted control, in which they call their own sequences of activities [11].

Although software reuse's benefits are known, keeping libraries and frameworks[1] updated during software maintenance is challenging. In most domains, there are several options to choose from. Some tools are up-to-date and have a growing and active community. Others are no longer maintained or deprecated and have no support from Q & A sites, such as Stack Overflow or CodeProject.

When evaluating software lifecycle, it is crucial to consider their relevance and robustness over time. Well-established software tools typically have an active community of developers, which is an indication of their long-term sustainability. Furthermore, it is important to check whether support and source code updates are frequent, as this indicates the health and vitality of the tool. Over time, these tools can go through several stages of evolution: initial development, maintenance, improvement, and eventually, decline.

Even well-established frameworks and libraries may have a decreasing usage over time, being replaced by others with better quality or popularity. Other tools may be poorly documented [19, 23] or difficult to use [17]. Another problem regards changes in the Application Programming Interfaces (APIs) of those software tools, which leads to bugs and backward incompatibilities [4, 14, 18].

Mining software repositories (MSR) uses historical data from code repositories, mainly the open source ones, to retrieve and understand several characteristics of software projects [8]. Discoveries from MSR may benefit the software engineering community in improving knowledge and developing techniques to advance the area: identifying trends and patterns, finding content on bug reports, collaborator interactions, and so on. Thus, MSR may also help to understand how frameworks and libraries are used through software projects. Can we use such data to provide a decision model that may help software teams choose suitable frameworks and libraries for their needs?

This study aims to determine if using MSR – along with Google Trends, Q & A sites, and other data sources – can indicate the lifecycle stage of frameworks and libraries. This approach may be useful for software teams to decide which tools should be chosen in software development projects [7]. Google Trends has been used in several domains to describe topics of interest over time [12].

The preliminary analyses presented in this work are based on data from two main sources: code repositories and Google searches about frameworks and libraries. Such data is used to verify their usage trends and to identify their lifecycle stages. This may provide insights on whether to adopt or replace these software tools.

[1] For brevity, we will refer to frameworks and libraries as software tools or projects.

Section 2 describes how our proposal is structured, followed by Section 3, which presents the results we obtained until now. Section 4 presents studies with objectives similar to ours. Then, Section 5 brings our conclusions and the next steps we will pursue.

2 METHODOLOGY

To evaluate the lifecycle of frameworks and libraries, we are selecting open source repositories to explore their features along with data from Google Trends. Then, we use trend analysis metrics to indicate the lifecycle stages of the analyzed tools.

2.1 Frameworks and libraries

We choose tools from distinct application domains, programming languages, and maintenance statuses. This choice was based on our knowledge of common application domains and popular languages, frameworks, and libraries. The six chosen domains were machine learning, data science, web, REST APIs, software testing, and security. Python, C++, Java, Ruby, and JavaScript were the five chosen programming languages. Regarding maintenance, we choose active projects, which are currently being evolved, and well-known legacy projects, with no current updates or those few used, which have presented a decreasing interest in the community. Our intention in choosing legacy projects is two-fold: leveraging data from those projects that still had a complete lifecycle to improve the analysis of active ones, and evaluating our method over those legacy projects.

The selection process was carried out manually by the authors as a way of exploring known frameworks and libraries since not all project repositories clearly state when it is a framework, library, or end-user software. Moreover, it was not easy to find some of the chosen legacy projects as most of their links to the original repository had expired. For instance, we did not find legacy REST API tools for Python and C++.

For each domain and language, we selected two active projects and one legacy project, resulting in 88 projects: 60 active and 28 legacy ones. Tables 2 and 3 (in Section 3) show the selected projects.

2.2 Data collection procedure

We used pyDriller[2] [21] to collect commits from the selected projects. The pyDriller library lets us collect several data from the commits: date, inserted and deleted rows, messages, author, etc. For now, we are using the number of commits and the number of authors over time to evaluate how much a project is being maintained and how many contributors the projects have.

To collect data from the Google Trends API, we chose the pyTrends library[3], which is being used to measure the popularity and the community engagement of each selected tool over time. The resulting measure is the relative interest per month. The relative interest is a normalized integer value that ranges between 0 and 100, which represents an increasing or decreasing interest in the searches for a particular term regarding the total number of Google searches.

Searching for some terms may bring results not related to the software tools (e.g., Pistache or Hanami). The pyTrends let us choose the category of interest: *Computers and Electronics* in our case. Furthermore, several tools have been searched using variations of terms (e.g., junit5, junit 5, and java unit testing 5) such that the sum of relative interest for a tool can exceed 100 in total.

[2]https://github.com/ishepard/pydriller
[3]https://github.com/GeneralMills/pytrends

2.3 Evaluation metrics

We used some evaluation metrics to allow us to identify variations in the number of commits, number of authors, and relative interest over time for each select project. To do that, we opted to use metrics related to trend analysis of time series data. Also, we applied correlation analysis to evaluate the chosen variables.

Trend analysis is a process of estimating a gradual change in future events from past historical data [5, 20]. Trend analysis has been used in distinct domains (e.g., stock market, ecology, climate) for forecasting pattern changes over time. Several statistical methods (e.g., Simple Moving Average (SMA), Mann-Kendall), and machine learning algorithms (e.g., Autoregressive Integrated Moving Average – ARIMA) can be used for trend analysis.

The idea behind using trend analysis is to identify a changing point – a moment in which a software project may have an increase or a decrease in its usage or maintenance – that would indicate a change in its lifecycle stage. For now, we are using one statistical trend analysis method: exponential moving average, as follows.

2.3.1 Exponential moving average (EMA). The exponential moving average is used to smooth variations in time series data, providing a result that is more sensitive to recent changes than long-term changes. EMA takes into account the current value, the previous EMA value, and a weighting factor. This factor determines how quickly the EMA reacts to new data.

The EMA formula is presented in Equation 1, where V is the current average value, EMA_{t-1} is the value of the previous time period $t-1$ and K is the weighting factor. In the weighting factor formula (Equation 2), N is the chosen time period.

$$EMA = V * K + EMA_{t-1} * (1 - K) \quad (1) \qquad K = \frac{2}{N-1} \quad (2)$$

2.3.2 Correlation analysis. Correlation analysis is a statistical measure useful to compare the dependency relationship between two variables. It is particularly useful when coping with big data to explore multivariate variables to reduce data dimensionality [22]. It is also useful to evaluate data from distinct datasets. The Spearman's rank correlation coefficient is nonparametric (i.e., it does not assume that data has any specific distribution) and its values range from -1 (strong negative correlation) to 1 (strong positive correlation. Values close to 0 have weak or no correlation.

We used the Spearman's rank correlation coefficient to assess the correlation among our study's variables: number of commits, number of authors, and relative interest.

2.4 Analysis procedures

To evaluate the lifecycle of the selected tools, we performed quantitative and qualitative analyses. For the quantitative analysis, we considered the trends for the exponential moving average (EMA) using two time periods: a short-term period (12 months) and a long-term period (24 months). We defined that a project lifecycle can fit into one of three stages based on the possible trends: *uptrend*, *stable*, or *downtrend*. Table 1 shows the criteria to classify frameworks and libraries into each lifecycle stage based on the EMA short-term and long-term periods. We calculate the lifecycle stage for each variable

Table 1: Criteria to classify frameworks and libraries according to their EMA short-term and long-term periods

Criterion	Lifecycle stage
EMA short-term period > EMA long-term period	Uptrend
EMA short-term period = EMA long-term period	Stable
EMA short-term period < EMA long-term period	Downtrend

– number of commits, number of authors, and relative interest. To obtain a final lifecycle stage classification, we consider the prevalence of two or more variables with the same value. If there is a draw – one of each variable as stable, uptrend, and downtrend – the framework or library is considered stable.

For the qualitative analysis, we compared the tools from the same domain and language to analyze them in more detail. The idea is to exemplify how a software team would use the proposed method to decide on using a framework or library. We chose a sample of the selected tools to perform this analysis, which includes time series charts, lifecycle stages, and correlation results. In the next section, we present our preliminary results.

3 PRELIMINARY RESULTS

The quantitative analysis shows a general sight of the lifecycle stages of the evaluated frameworks and libraries. Then, we present a qualitative analysis that takes a closer look at some projects to understand their specificities regarding our classification method.

3.1 Quantitative analysis

The 88 tools were classified using EMA to calculate the trends for each variable: the number of commits, number of authors, and relative interest. The lifecycle stage is assigned based on the most frequent value trend value among the three variables.

Table 2 shows the results for the active projects. Projects in the uptrend stage (↑) are those in a gray background. The other projects are in the downtrend stage (↓). We can see that active tools vary regarding their lifecycle stages. From them, 31% (19 out of 60) were classified in an uptrend lifecycle. For example, the D3.js library is in an uptrend. The Chart.js library, from the same language and domain, is in a downtrend lifecycle. C++'s Pistache and Restbed tools are in a downtrend. Also, we did not assume beforehand whether an active tool has an upward or downward trend of usage as we did for the legacy ones.

Table 3 follows the same rationale of Table 2 for the legacy projects. It shows that most legacy projects were correctly classified in a downtrend, 78.5% of them. Most of the exceptions are tools for the Ruby language or security tools.

None of the projects was classified as stable. It only could occur if a project has the three variables assigned with distinct trends, or if at least two variables had a stable stage for the same project, which did not occur for the selected tools.

Of the 88 tools, 25 were classified in an uptrend lifecycle – 19 of them are active projects and 6 are legacy ones. The 64 remaining projects were classified in a downtrend stage. Of the 64, 29 have a high trend of relative interest – 25 are active and 4 are legacy.

The correlation analysis shows varied patterns of results. For example, pandas, pytorch, playframework, and cppunit had a strong positive correlation among all variables. A few projects, such as minitest, have presented no correlation between the three comparisons. Table 4 shows the aggregated number of projects by correlation value levels: medium-to-strong positive correlations (> 0.5), medium-to-strong negative correlations (< -0.5), and weak positive and negative correlations or no correlation (> -0.5 and < 0.5). There were 11 projects without enough information of relative interest, which were in the last row (no data).

There are stronger correlations between the number of commits and the number of authors, which makes sense since more authors tend to submit more commits. There are several cases in which the relative interest has a strong negative correlation with the number of commits and the number of authors (e.g., NSP – Node Security Protect), which may mean that projects with high interest and low maintenance are stable, with no need for frequent improvements. We will explore such issues in future work. Since almost all projects presented at least one strong positive or negative correlation, we intend to use the correlation as a weighting criterion for classifying the lifecycle stages in future work.

3.2 Qualitative analysis

For the qualitative analysis, we made three comparisons, each one by selecting two projects from the same domain and language. Thus, we intended to assess how the proposed method can be useful to decision-making on choosing between two similar tools. Moreover, we can show in more detail the characteristics of those tools and how they are related to our lifecycle classification method.

We explored different approaches in the comparisons: competing active projects (PyTorch vs TensorFlow and RubyOnRails vs Hanami); active and legacy projects (JUnit 4 vs JUnit 5); programming languages (Python, Java, and Ruby); and domains (machine learning, web, and testing).

3.2.1 PyTorch vs TensorFlow. PyTorch and TensorFlow are examples of competing machine-learning Python libraries. PyTorch has had commits since 2012 but was released in 2016, while TensorFlow was released in 2015. First, we can visualize trends in the number of commits, number of authors, and relative interest to understand how this data relates, which can help inform eventual design decisions. Figure 1 shows the relative interest over time. Table 5 shows the obtained EMA trends for PyTorch and TensorFlow, and Table 6 shows the correlations among the variables for both projects.

It is possible to observe an advantage of PyTorch trends compared to TensorFlow. According to Dai et al. [3], PyTorch is easier to use and has a lower learning curve, which helps to explain a certain advantage in the relative interest. Despite both being well-established tools, TensorFlow had a more pronounced growth in the number of commits and authors in the initial years when compared to PyTorch. Both libraries have strong correlations.

3.2.2 JUnit 4 vs JUnit 5. The Java testing framework JUnit 5 is the successor to JUnit 4. There is a big difference in lifecycle of both versions. Figure 2 shows the number of authors over time for JUnit 4 and JUnit 5. Table 5 shows the obtained EMA trends for JUnit 5 and JUnit 4.

As expected, the trends for commits and authors are higher for JUnit 5 compared to JUnit 4. However, for the relative interest, JUnit

Table 2: Lifecycle stage of active projects

Domain	Python	C++	Java	Ruby	JavaScript
Machine Learning	TensorFlow ↓	TensorFlow ↓	Spark ↑	TensorFlow.rb ↓	TensorFlow.js ↓
	PyTorch ↑	pytorch-cpp ↑	H2O ↓	SciRuby ↓	Brain.js ↓
Web	Django ↓	Wt↑	Play Framework ↓	Ruby on Rails ↑	React ↓
	Flask ↑	C++ REST SDK ↓	Apache Struts 2 ↑	Hanami ↓	Vue.js ↓
Data Science	pandas ↑	Armadillo ↓	MOA ↓	NMatrix ↓	D3.js ↑
	Dask ↓	ITK ↓	WEKA ↓	Statsample ↑	Chart.js ↓
Rest API	FastAPI ↑	Pistache ↓	Spring REST ↑	Grape ↓	Express ↓
	Eve ↓	Restbed ↓	RESTEasy ↓	Roda ↓	koa ↓
Testing	pytest ↑	GoogleTest ↑	JUnit 5 ↑	RSpec ↓	Jest ↓
	Robot Framework ↓	Catch2 ↓	TestNG ↓	minitest ↓	Cypress ↓
Security	ZAP ↑	OpenSSL ↑	OWASP Java Encoder ↓	Brakeman ↓	nsp ↓
	Paramiko ↑	Botan ↑	Bouncy Castle ↓	Sorcery ↓	Helmet.js ↓

Table 3: Lifecycle stage of legacy projects

Domain	Python	C++	Java	Ruby	JavaScript
Machine Learning	Orange ↓	CNTK ↓	Deeplearning4j ↓	RubyFANN ↓	Synaptic ↓
Web	Web2py ↓	CppCMS ↓	Apache Struts 1 ↓	Camping ↑	Backbone.js ↓
Data Science	PyTables ↓	mlpack ↓	Apache Mahout ↓	Scruffy ↑	jStat ↓
Rest API	–	–	Jakarta Faces ↓	Sinatra ↓	Restify ↓
Testing	unittest2 ↓	CppUnit ↓	JUnit 4 ↓	Test::Unit ↑	QUnit ↓
Security	PyCrypto ↑	LibTomCrypt ↓	JCE ↓	OpenSSL ↑	jsrsasign ↑

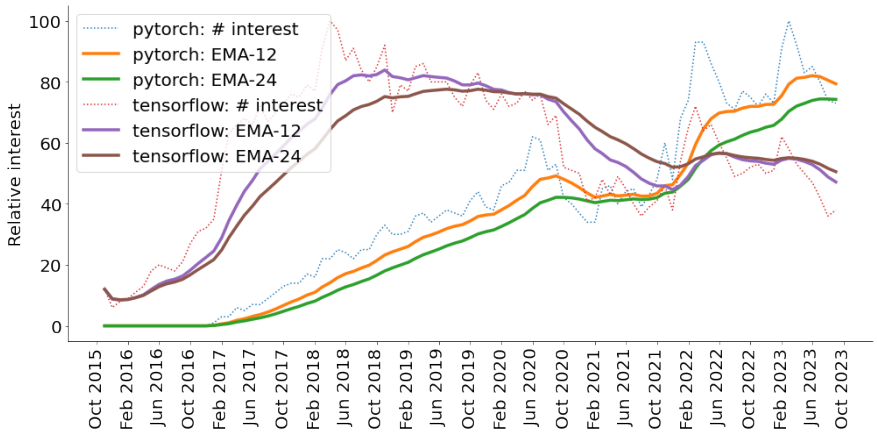

Figure 1: Relative interest over time for PyTorch and TensorFlow

Table 4: Number of tools for distinct levels of Spearman's correlation score for the compared variables: number of commits (C), authors (A), and relative interest (I)

Correlation	C/I	A/I	C/A
> 0.5	18	27	53
< -0.5	16	9	0
> -0.5 and < 0.5	43	41	35
No data	11	11	0

Table 5: EMA's short-term (12) and long-term (24) for the number of commits (C), authors (A), and relative interest (I) for the selected tools

Tool	EMA (C)		EMA (A)		EMA (I)	
	12	24	12	24	12	24
PyTorch	1017.15	1004.10	226.74	220.21	83.99	80.37
TensorFlow	1486.54	1533.46	207.49	209.97	51.85	53.65
JUnit 4	2.95	3.84	1.32	1.59	48.79	48.20
JUnit 5	44.74	43.90	6.79	6.69	145.90	146.64
Ruby on Rails	320.09	314.84	64.92	66.44	42.02	41.85
Hanami	7.75	8.68	2.27	2.52	16.41	14.49

Table 6: Correlations between the number of commits (C), authors (A), and relative interest (I) for the selected tools

Tool	C/I	A/I	C/A
PyTorch	0.952	0.953	0.972
TensorFlow	0.659	0.609	0.982
JUnit 4	0.443	-0.259	0.602
JUnit 5	-0.839	0.216	0.391
Ruby on Rails	0.081	0.113	0.87
Hanami	0.24	0.096	0.948

4 is in an uptrend while JUnit 5 is in a downtrend. A possible reason is that JUnit 4 is still used in many projects that have not been updated to JUnit 5 [6]. Although JUnit 5 is in a downtrend, it is almost three times more searched compared to JUnit 4 considering their relative interest. Thus, in future work, we should consider the absolute values of relative interest as a weighting criterion. Table 6 shows that the correlation results involving the relative interest are inversely proportional for JUnit 4 and JUnit 5.

3.2.3 Ruby on Rails (RoR) vs Hanami. The RoR and Hanami are competing active web frameworks from the Ruby language. Hanami is more recent, starting in 2014, ten years after RoR. Figure 3 shows the number of commits over time for both tools.

Despite the long time since its launch, Ruby on Rails continues to be widely used due to its stability. Hanami is still on the rise. It has less community support, implementations, and tutorials available when compared to RoR [1]. It can be seen by observing the trends (Table 5) and correlations (Table 6). RoR has a greater number of commits, authors, and relative interest. However, Hanami is in an uptrend regarding relative interest, as it is promising, lightweight, and has extensive documentation.

4 RELATED WORK

There are other studies and initiatives to identify suitable projects for software reuse. Mileva et al. [13] used mining software repositories (MSR) through hundreds of open source Java projects to discover their external dependencies (libraries – APIs). They consider that a successful API is the one that is largely used, which is referred to as the wisdom of the crowds of API users. On the contrary, an API is considered decadent if it lacks popularity. APIs are classified into four types of trends: increasing, decreasing, stable, and undecided. This crowdsourcing-based approach may not deal with new emergent APIs, which could be of high quality but are not popular yet.

Hora and Valente [9] presented a tool (apiwave) that monitors 650 Java projects. The tool computes the popularity of an API from the differences in its usage in the observed projects, which can increase or decrease over time. Beyond the popularity chart, the tool presents a migration rank, which contains the most replaced APIs. Also, it presents code excerpts with usage examples. Currently, the tool platform is unavailable at the provided URL.

Coelho et al. [2] measured the level of maintenance of GitHub projects. To do that, the authors explored data from 2927 projects written in several programming languages during the year of 2018. The random forest classifier was applied to features related to commits, issues, pull requests, and contributors, among others, to classify projects as maintained or unmaintained.

Mujahid et al. [15, 16] investigate the decline of packages of the node package manager (npm) for software reuse. In [16], they propose an approach to identify package decline using a centrality trend metric. The centrality is calculated periodically considering the dependencies among all npm packages, which are represented by a dependency graph. The proposed technique is provided as a tool that can be used for packages from the npm ecosystem. In [15], the authors suggest alternative packages for those in decline. They first identify migration patterns to find candidates for replacement suggestions. Only packages among the ten percent best centrality scores are considered as replacement candidates to avoid the suggestion of immature packages.

Our study aims at identifying the lifecycle stages of libraries and frameworks. As in the work of Coelho et al. [2], we explore the internal features of the assessed projects. However, our study will also explore data from Q & A sites and Google Trends. This latter is not used in previous related work. Our proposal addresses software tools from several languages, which is not done in most related work, more focused on the Java language, while Mujahid et al. [15, 16] is focused on npm ecosystem (JavaScript). Moreover, our work also differentiates by using trend analysis (i.e., exponential moving average) to classify the assessed projects.

5 CONCLUSIONS AND NEXT STEPS

This work presents an approach to analyze the lifecycle of frameworks and libraries to help choose possible alternatives for use in software development projects. Our proposed method indicates lifecycle stages of software tools, which can serve as useful information for software maintenance and evolution. These lifecycle stages are based on trend results for the number of commits and authors over time, extracted from their repositories, and the relative interest over time obtained from Google Trends. To do this, we use the exponential moving average (EMA) trend metric. Furthermore, we evaluated the correlation between these variables.

We should consider that each project has its own characteristics and that its data may vary without following a specific pattern, depending on factors such as the programming language, application domains, and developers' collaborations.

The results show that the proposed method can identify the frameworks' and libraries' lifecycle trends in most cases. The quantitative analysis shows that most legacy projects (78.5%) were classified as in a downtrend lifecycle, as expected. A few legacy projects showed an uptrend lifecycle stage. Even small changes in commits among the periods or in the increase in the interest in a library or framework may lead to an uptrend status. The qualitative analysis shows that, for those selected projects, the lifecycle stages can be explained through the characteristics of the tools' historical data.

Regarding the selection of the active projects, we do not assume any previous trend about their popularity, current level of usage, or maintenance frequency. We simply selected well-known projects for each programming language and domain. Therefore, other relevant projects may have been out of this preliminary study.

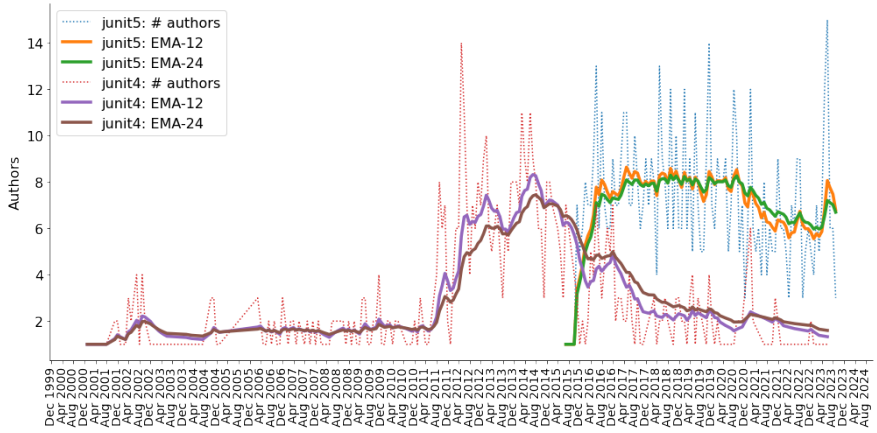

Figure 2: Authors over time for JUnit 4 and JUnit 5

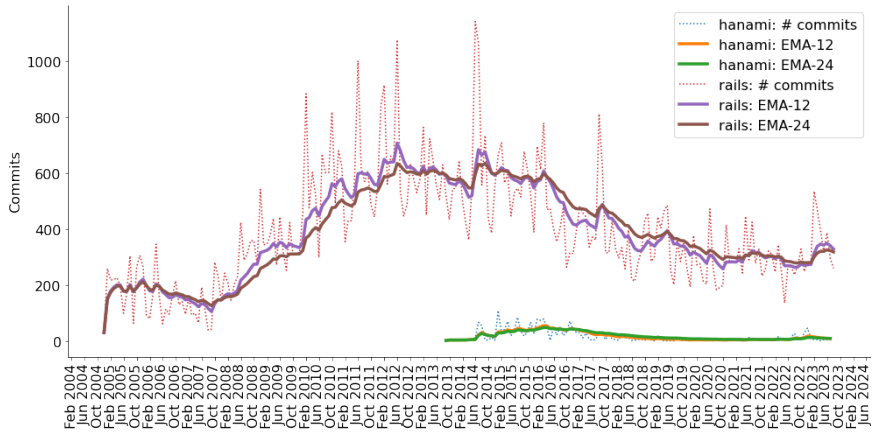

Figure 3: Commits over time for Ruby on Rails and Hanami

5.1 Future work

Although preliminary results are promising, there is a lot of useful data that can improve the proposed work. We will include data from projects that use such frameworks and libraries, and other data from repositories' popularity metrics: number of stars, forks, and watchers. We will use text mining to extract data from bug reports, issues, posts from Q & A sites, and IT-related sites.

We will explore time periods other than 12 and 24 months to calculate EMA, which was the choice for this current study. We will also assess if there is a threshold for the number of commits, authors, and relative interest that we could consider to avoid slight variations in those variables changing the lifecycle stage.

In our next steps, we will evaluate how time series machine learning techniques, such as ARIMAX (AutoRegressive Integrated Moving Average with Exogenous Regressors), SARIMAX (Seasonal ARIMAX), and LSTM (Long Short-Term Memory) can be used to improve the trend analysis prediction. We intend to cover more frameworks and libraries from other programming languages and domains. Finally, we will perform a user study to assess how useful such an approach can be in the software development workflow for the decision-making process of software reuse.

The repository data is available at: https://github.com/higoramario/software-lifecycle-msr.

REFERENCES

[1] Marek Buszman. 2023. Modern Ruby Frameworks Comparison: RoR vs Hanami. [Online]. Available from: https://www.netguru.com/blog/comparison-ror-vs-hanami. Acessed in: 20 oct. 2023.
[2] Jailton Coelho, Marco Tulio Valente, Luciano Milen, and Luciana L. Silva. 2020. Is this GitHub project maintained? Measuring the level of maintenance activity of open-source projects. *Information and Software Technology* 122 (2020), 106274. https://doi.org/10.1016/j.infsof.2020.106274
[3] Hulin Dai, Xuan Peng, Xuanhua Shi, Ligang He, Qian Xiong, and Hai Jin. 2022. Reveal training performance mystery between TensorFlow and PyTorch in the single GPU environment. *Science China Information Sciences* 65 (2022), 1–17.
[4] Alexandre Decan, Tom Mens, and Philippe Grosjean. 2019. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering* 24, 1 (2019), 381–416.
[5] Alexander Elder. 2002. *Come into my trading room: a complete guide to trading.* John Wiley & Sons, New York, NY.
[6] Boni Garcia. 2017. *Mastering Software Testing with JUnit 5: Comprehensive guide to develop high quality Java applications.* Packt Publishing Ltd, Birmingham, UK.
[7] Ronaldo Rubens Gesse Júnior. 2023. Mineração de Repositórios para análise de ciclos de software. Capstone project (Bachelor of Computer Science), São Paulo

State University, School of Sciences. https://hdl.handle.net/11449/251494

[8] Ahmed E Hassan. 2008. The road ahead for mining software repositories. In *Proceedings of the 2008 Frontiers of Software Maintenance* (Beijing, China) *(FoSM 2008)*. IEEE, Piscataway, NJ, 48–57.

[9] André Hora and Marco Tulio Valente. 2015. apiwave: Keeping track of API popularity and migration. In *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME'15)*. IEEE, Piscataway, NJ, 321–323.

[10] Ralph E Johnson. 1997. Frameworks=(components + patterns). *Commun. ACM* 40, 10 (1997), 39–42.

[11] Ralph E Johnson and Brian Foote. 1988. Designing reusable classes. *Journal of Object-Oriented Programming* 1, 2 (1988), 22–35.

[12] Seung-Pyo Jun, Hyoung Sun Yoo, and San Choi. 2018. Ten years of research change using Google Trends: From the perspective of big data utilizations and applications. *Technological Forecasting and Social Change* 130 (2018), 69–87. https://doi.org/10.1016/j.techfore.2017.11.009

[13] Yana Momchilova Mileva, Valentin Dallmeier, and Andreas Zeller. 2010. Mining API popularity. In *Proceedings of the 5th International Academic and Industrial Conference: Testing – Practice and Research Techniques (TAIC PART 2010)*. Springer-Verlag, Berlin, Germany, 173–180.

[14] Shaikh Mostafa, Rodney Rodriguez, and Xiaoyin Wang. 2017. Experience paper: a study on behavioral backward incompatibilities of Java software libraries. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '17)*. ACM, New York, NY, 215–225.

[15] Suhaib Mujahid, Diego Elias Costa, Rabe Abdalkareem, and Emad Shihab. 2023. Where to Go Now? Finding Alternatives for Declining Packages in the npm Ecosystem. In *Proceedings of the 2023 38th IEEE/ACM International Conference on Automated Software Engineering* (Luxembourg, Luxembourg) *(ASE'23)*. IEEE, Piscataway, NJ, 1628–1639. https://doi.org/10.1109/ASE56229.2023.00119

[16] Suhaib Mujahid, Diego Elias Costa, Rabe Abdalkareem, Emad Shihab, Mohamed Aymen Saied, and Bram Adams. 2022. Toward Using Package Centrality Trend to Identify Packages in Decline. *IEEE Transactions on Engineering Management* 69, 6 (2022), 3618–3632. https://doi.org/10.1109/TEM.2021.3122012

[17] Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. 2016. Jumping through hoops: why do Java developers struggle with cryptography APIs?. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. ACM, New York, NY, 935–946. https://doi.org/10.1145/2884781.2884790

[18] Romain Robbes, Mircea Lungu, and David Röthlisberger. 2012. 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 (FSE'12)*. ACM, New York, NY, 1–11.

[19] Martin P Robillard and Robert DeLine. 2011. A field study of API learning obstacles. *Empirical Software Engineering* 16 (2011), 703–732.

[20] Shilpy Sharma, David A Swayne, and Charlie Obimbo. 2016. Trend analysis and change point techniques: a survey. *Energy, Ecology and Environment* 1 (2016), 123–130.

[21] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. 2018. PyDriller: Python framework for mining software repositories. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'18)*. ACM, New York, NY, 908––911. https://doi.org/10.1145/3236024.3264598

[22] Chengwei Xiao, Jiaqi Ye, Rui Máximo Esteves, and Chunming Rong. 2016. Using Spearman's correlation coefficients for exploratory data analysis on big dataset. *Concurrency and Computation: Practice and Experience* 28, 14 (2016), 3866–3878.

[23] Hao Zhong and Zhendong Su. 2013. Detecting API documentation errors. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications* (Indianapolis, IN) *(OOPSLA '13)*. ACM, New York, NY, USA, 803—-816. https://doi.org/10.1145/2509136.2509523