The Long Tail: Understanding the Discoverability of API Functionality

Amber Horvath*, Sachin Grover*, Sihan Dong*, Emily Zhou*, Finn Voichick†, Mary Beth Kery*,
Shwetha Shinju*, Daye Nam*, Mariann Nagy‡, and Brad Myers*
*Carnegie Mellon University
{ahorvath,sachingr,sihand,emilyzho,mkery,sshinju,namdy,bam}@cs.cmu.edu
†Washington University in St. Louis
fvoichick@wustl.edu
‡Google Inc.
mknagy@google.com

***Abstract*—Almost all software development revolves around the discovery and use of application programming interfaces (APIs). Once a suitable API is selected, programmers must begin the process of determining what functionality in the API is relevant to a programmer's task and how to use it. Our work aims to understand how API functionality is discovered by programmers and where tooling may be appropriate. We employed a mixed-methods approach to investigate Apache Beam, a distributed data processing API, by mining Beam client code and running a lab study to see how people discover Beam's available functionality. We found that programmers' prior experience with similar APIs significantly impacted their ability to find relevant features in an API and attempting to form a top-down mental model of an API resulted in less discovery of features.**

***Index Terms*—API discoverability, API usability, API learnability, API usage analysis**

I. Introduction

Application programming interfaces (APIs) are the backbone of most modern software development. Programmableweb.com has registered over 20,000 APIs, with countless more private APIs existing for private, internal use. Despite the prevalence of APIs, understanding how they are used and areas for improvement or tooling support is still an ongoing effort. One aspect of usage is how users find available functionality within an API that will aid them in their tasks.

Previous work in API usage analysis has found that the frequency distribution of API method calls in the Java Development Kit (JDK) follows the Zipf distribution, with over 90% of the over 5000 analyzed projects using less than 5% of the methods in the core JDK [1]. Zipf's Law states that the most frequently used term in a corpus will appear approximately twice as often as the second term, and so on.

While the Zipf distribution has been observed in API usage, it remains unclear what causes it. One possibility may be that users are not *discovering* the proper functionality. One API that may suffer from poor discoverability is Apache Beam [2], a distributed data processing API available for Python, Java, Go and other languages, as it is a relatively new API with a less-established user base.

In this paper, we present a formative study about how Beam programmers use available functionality and a lab study to investigate the facets of API discoverability with Apache Beam.

978-1-5386-5541-2/18/$31.00 ©2019 IEEE

II. Related Work

A. Studying Learnability and Usability of APIs

Much research effort attempts to understand what constitutes a "usable" API and how to design one. Currently, apiusability.org has indexed 71 publications over the last 22 years. Despite this ongoing effort, API usability is still seen as an evolving field tackling multiple important issues such as methods for evaluating and improving API usability [3]–[6] and documentation's affect on API usability [7]. One aspect of API usability is its learnability [8], [9]. API learnability and discoverability go hand in hand, as discovery is an integral aspect of learning. Previous work has investigated what makes an API learnable, especially with respect to documentation, through investigating developer's most pressing questions [10] and obstacles [11] when learning an API. Some questions appear to be discoverability issues, such as "which keywords best describe a functionality provided by the API?" and "which method from a list of overloaded methods is relevant to my task?", questions which are not easily answered through simply looking at documentation. Stylos and Myers found that discoverability was significantly hampered if a method or object was not associated with the user's chosen "starter" class [12]. Discoverability through autocomplete has been shown to help with this by clearly documenting the relationship between a class and its methods by showing them as options [13], [14].

B. API Usage Analysis

One way of understanding the discoverability of an API's functionality is seeing how methods, classes, packages, and interfaces are used at scale by analyzing the API's usage. Qiu et al. performed a large scale analysis across 5000 Java projects to understand the frequency in which API calls were made and found that the top 1% of packages make up 80% of

all API usage, with a similar pattern emerging for method, interface, and class usage in the JDK [1]. Thummalapenta and Xie developed a program which finds the "hotspots" and "coldspots" of open source frameworks, where hotspots are heavily reused portions of code and coldspots are API methods that are rarely used. While the authors emphasize how hotspots can serve as a fertile ground for tool and documentation usage by highlighting popular patterns and methods, they do not discuss where the coldspots come from or if they would be beneficial to API users if they were to be found and used [15].

III. Background

Apache Beam [2] is a distributed data processing API for creating data pipelining programs. Beam can be configured to read from streaming data sources and can run jobs on large-scale processing platforms such as the Google Cloud Platform. The way Beam programs are typically structured is that the user defines a processing pipeline where data enters the pipeline and is operated on, resulting in a new dataset which is operated on, and so on. In the scope of this work, we focus on how users find and adapt the built-in PTransforms (or "transforms") for their programs, since they provide the structure for the pipeline.

IV. Repository Mining

To see how people discovered functionality at scale, we wrote a script to mine from GitHub the uses of Apache Beam's transforms. Since there are fewer help resources available for Beam, and less client code available as a reference on GitHub, we hypothesized that this would lead to certain transforms which are featured in the Programming Guide to be used more often than other transforms.

1) Method: We used Selenium, an open source web automation tool (with a headless browser), to extract user-repository pairs from Github search results pages. The parameters of the search were set to look for repositories where Beam's Python SDK was used. We used string parsing techniques with a global dictionary of Beam transforms (created from Beam's documentation) in the corresponding Python files to count the occurrences of each transform. As a comparison, the same process was repeated to extract the count of transforms for PySpark. Due to GitHub's terms of service, we were limited to mining the first 100 pages of the GitHub search results for each API. For the Beam Python SDK, we found 288 GitHub users with a total of 308 repositories. In all of the files, a total of 3079 transforms were used. For PySpark, we found 502 users with 517 repositories. To have a comparable number of repositories for the 2 APIs we randomly sampled 310 repositories from this collection (60%) which contained a total of 3962 PySpark transforms.

2) Analysis and Results: Figure 1a and Figure 1b show the frequency distribution of transforms in the mined repositories for Beam and Spark, respectively. As expected, we observe a long tailed distribution, matching the trend observed in the JDK [1]. A closer look at the distribution reveals some interesting trends. For example, *Map* was the most frequently used transform in Beam (appearing in 65% of the projects) despite not being included in the Programming Guide's list of Beam's *Core Transforms*, which represent transforms corresponding to the main processing paradigms in Beam. The popularity of *Map* among users may be attributed to the fact that it is used as a construct in many data processing APIs. The second most frequent transform is *ParDo*, which is a core transform appearing in 34.4% of the projects. This is not surprising as it is the most "advertised" transform in the programming guide. Other common transforms are also the ones which are either explicitly mentioned in the text (green) or appear in the examples (red) as shown in Fig 1a.

We observe a similar distribution of transforms in Spark – most of the frequently used transforms in Spark appear in the programming guide (green in Figure 1b). In general, more transforms explicitly appear in the guide but surprisingly quite a lot of them are rarely used (tail in Figure 1b). We hypothesize that their long obscure names or very specific functionality (required for rare use cases) might be reducing the use of these transforms. Unsurprisingly, across both APIs, more emphasized functions in the programming guides are more frequently used. This could either mean that the documentation team correctly predicted that these are the transforms most users would want to know or it could be that, since they are the easiest to learn about, developers adapted their programs to use them. However, being in the Programming Guide does not guarantee usage – for example in Beam, *FixedWindows*, a helper function for the window transform, is used in a code example in the programming guide but never appeared in any of the files we analyzed. Moreover, some transforms which are never explicitly discussed in the programming guide are still used. The most frequently used transform in Beam that is not in the programming guide is Filter (5.6% of client files), which we speculate is a particularly easy name to guess.

V. Lab Study

Since we are interested in understanding *how* people find functionality in Beam, make sense of it, and choose to adopt it into their programs, we developed an exploratory lab study.

We applied pre-existing learning-based models to discoverability and learnability of API functions. We chose to look at familiarity of concepts and robustness of a user's mental model [5] and programming learning styles [9] as a framework of discoverability as these theories impact how the programmers make sense of an unfamiliar API. This can be summarized in the following 2 research questions:

- *RQ1. How does familiarity with the API's core concepts affect the discoverability of API functionality?*
- *RQ2. How do different learning styles affect the discoverability of API functionality?*

A. Method

We recruited undergraduate and masters students studying computer science who had some amount of experience with Python and a data processing or analysis API such as Apache Spark, numpy, or pandas. In total, we recruited 10 participants,

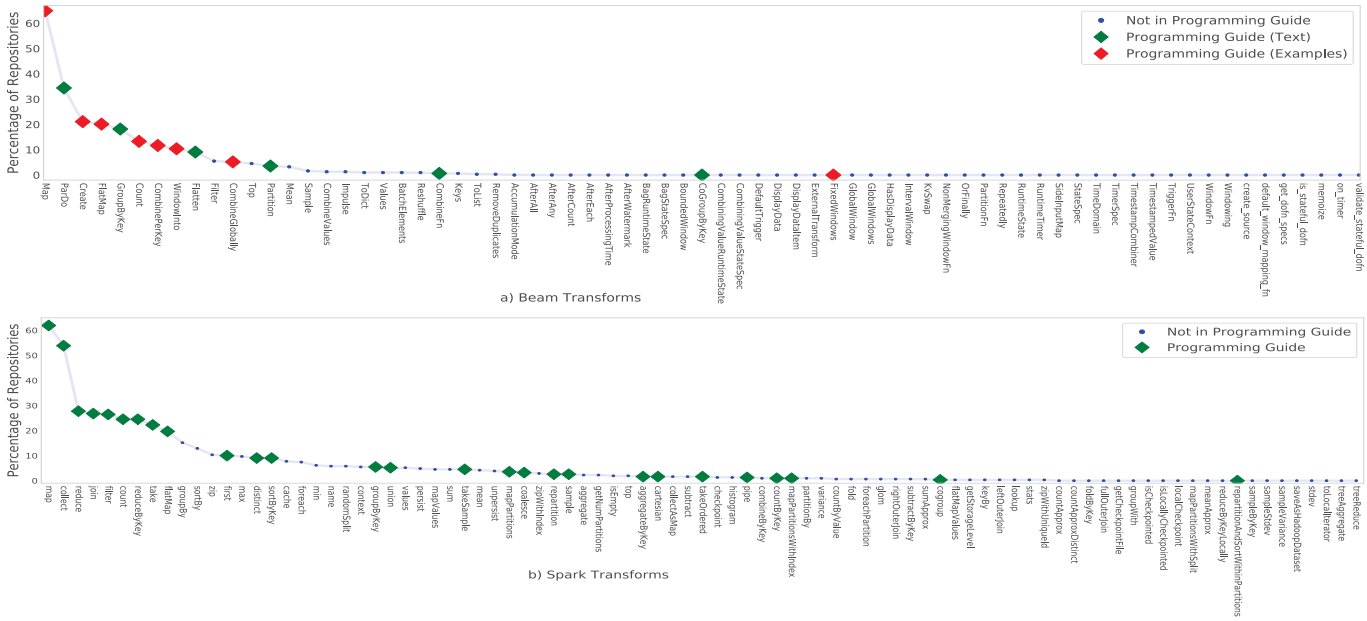

Fig. 1: Distribution of Transforms in (a) Beam and (b) Spark

4 men and 6 women. None were familiar with Beam. We administered a think-aloud study with two experimenters, one running the study while the second took notes. The participant's computer screen and audio were recorded while they completed the task. Participants were compensated $10 for their time.

We adapted an existing Python SDK Beam word count sample program to be the task, as we felt that it served as a benchmark for what the Beam designers felt were essential transforms a user should be familiar with. The script includes a class that must be called with Beam's ParDo transform to extract the words, and then uses two Python functions to count each time a word appears and to format the results. We refer to this code as "helper" code. The correct script had a total of either 6 or 7 possible correct transforms in the given order, as the grouping keys and summing 1's may either be done with two transforms (GroupByKey and Map) or one (CombinePerKey).

Participants were introduced to Beam, PyCharm, and the autocomplete tools for 10 minutes. Participants were then given 30 minutes to complete the word count task followed by up to 15 minutes for post-task questions, which asked about the participants problem-solving style in general, where they looked for information, and the help resources they used.

B. Analysis

Videos of the participants completing the task were coded to indicate when they used autocomplete, when the users were in the editor versus looking at the programming guide, every time they added, deleted or modified code and what the changes to their code were, and when they ran their code. From this we analyzed how participants spent their time while they tried to understand and use the API.

We correlated these data with the success the participants had on the task, where success was measured based on the correctness of the user's chosen transforms. Due to recording failures, only 8 of the 10 participants were coded for time-spent during the task and help resources used during the task. The first, third, and sixth authors completed the analyses of the videos.

C. Results

None of the participants fully succeeded on the task, however, all participants were able to get at least 1 correct transform. The average score was 3 correct transforms out of a possible 6 or 7, with a mode of 2. The lowest score was 1 correct transform, while the highest score was 5. We are interested in understanding this large variability in terms of correct transforms and how the models we adapted for our RQs are interconnected in the way they characterize discovery.

We began by investigating RQ1 - the extent to which each participant had familiarity with data processing APIs and how that impacted discoverability. All participants had *some* amount of experience with data processing and analysis APIs, but there was a divide in terms of type of experience. 3 participants had experience using distributed, map-reduce paradigm APIs (Hadoop, Apache Spark, and MapReduce), while the remaining 7 only had experience using non-distributed data analysis APIs (numpy, scipy, and pandas). We found that the participants who had experience using at least one of the distributed data processing APIs performed significantly better on the task with a mode of 5 correct transforms (T-test: $p < .02$, *Cohen's* $d = 2.439751$). The participants who only had experience with non-distributed APIs such as numpy had less success, with a mode of 2 correct transforms.

Not surprisingly, we saw evidence that familiarity with similar APIs allowed participants to draw upon their prior

knowledge. For example, P1, who had prior experience using Apache Spark, stated in the post task interview that their search for functionality was driven by "the Spark API in the back of my head." Since they were able to draw analogies between the APIs, they were able to more accurately guess what functionality the API would and would not provide. However, this strategy broke down for Beam-specific constructs. For example, P5, who had experience with Spark and got 5 transforms correct, said that, with autocomplete, they were able to find and figure out everything except for "ParDo," Beam's "parallel do" function. In contrast, participants without distributed API processing experience were less likely to know what the API would be able to do as they made incorrect assumptions about what functionality the API would provide.

In order to answer RQ2, how differing programming styles impacted discoverability, we analyzed how programmer's spent their time during the task. We characterized these activities as either enrichment activities such as reading "helper" code and the programming guide, versus task-progression activities such as adding and editing code. We were interested in how the focus on these different activities would affect how participants discover and use transforms.

Attempting to discover elements of Beam primarily through the programming guide was not a successful strategy. P7, the only participant who spent the majority of her time in the programming guide, successfully discovered 2 transforms and gained a comprehensive understanding of ParDo, Beam's most discussed transform. She chose to use it for multiple steps of the pipeline, despite only needing it for the first step. The other transform she discovered, GroupByKey lacked an example in the programming guide, leaving her unsure how to use it and what data type it expected.

The programmers who started off with enrichment activities, then spent a good amount of time attempting to program before going back to enrichment activities such as P3, P4, P5 and P8 were mostly successful. For example, P5 began by writing code and, through the code completion, discovered that there was a Map transform, a Filter transform, and a Sample transform which he felt he could use for nearly every step of the pipeline until he realized this would not work for the first step of the task, which requires ParDo, and moved to the documentation. Our findings are generally consistent with prior work [9], [16]–[19] that finds that programmers have different learning styles – some are systematic and prefer a more top-down approach of reading documentation first, whereas others are more pragmatic and prefer to focus on coding until they realize they need to learn something from the documentation.

VI. Discussion and Implications

We investigated two facets of discoverability – developers' previous API usage and programming styles. We are interested in how these facets affect one another and result in a programmer's ability to discover the functionality they are interested in. While our evidence supports the claim that a developer's past experience best predicts their ability to discover relevant information in a new, similar API, P4 serves as an interesting counterpoint – despite having no prior experience with distributed data processing APIs, he still discovered 5 transforms. One way in which P4 shared similarity with the other top performers was in programming style. P3, P4, and P5 all spent some time in the beginning of the task familiarizing themselves with the programming guide and "helper" code prior to writing code with P3 also spending time looking at autocomplete, and then proceeding to move into implementation. Those who spent more time attempting to comprehensively understand everything about Beam before attempting the task had less success. Essentially, programmer's prior experience was the most important factor in their ability to discover new functionality, but programming style also benefited or hindered discovery.

We propose some potential implications for API designers to consider to improve the discoverability of API entities through modifications to documentation and tooling. One way in which participants struggled was by attempting to search for transforms in the programming guide. Both P2 and P10 attempted to search for "pair" or "key/value pair" but were unsuccessful. P6 searched for "tuple" in the programming guide to no avail. These unsuccessful queries hindered discovery of applicable transforms as they returned irrelevant results. One possible solution is supplying multiple keywords that relate to vocabulary from other APIs or the function's intended purpose. API designers should ensure that method names are easily understandable and, for constructs that have naming specific to their API, provide thorough explanations or analogies relating these constructs to more well-known concepts.

VII. Threats to Validity

We chose to only study Apache Beam's discoverability, which may not be representative of other APIs. The task was also relatively short, so it is unclear that if the task was longer, the discoverability behavior may have changed over time as participants grew more familiar with the API. Future studies may benefit from seeing how growing familiarity with an API changes how participants discover features.

For our mining study, we are unsure if we looked at a truly representative sample of Beam repositories – for example it might be that most production level repositories are either private or stored outside of GitHub. We also only looked at the Python SDK Beam usage, which may not be representative of Java or Go usage.

VIII. Future Work

In future work, we will attempt to create responsive documentation, which, based upon the user's prior knowledge, recommends potential features to use. Considering prior experience was the largest influence on how well programmers were able to discover features, perhaps documentation could self-adapt by gauging the user's prior experience in order to supply differing levels of information to facilitate discovery. We are also interested in investigating how developer's interpersonal networks impact discoverability.

ACKNOWLEDGEMENT

This work was supported by NSF IIS-1827385 and a grant from Google.

REFERENCES

[1] D. Qiu, B. Li, and H. Leung, "Understanding the API usage in Java," *Information and Software Technology*, vol. 73, pp. 81–100, 2016.

[2] T. A. S. Foundation, "Apache beam: An advanced unified programming model," beam.apache.org, 2018, accessed: 2019-04-30.

[3] B. A. Myers and J. Stylos, "Improving API usability," *Communications of the ACM*, vol. 59, pp. 62–69, 2016.

[4] B. A. Myers, A. J. Ko, T. D. LaToza, and Y. Yoon, "Programmers are users too: Human-centered methods for improving programming tools," *IEEE Computer*, vol. 49, pp. 44–52, 2016.

[5] A. Horvath, M. Nagy, F. Voichick, M. B. Kery, and B. A. Myers, "Methods for investigating mental models for learners of APIs," in *Proceedings of the 2019 CHI Conference Extended Abstracts on Human Factors in Computing Systems*, ser. CHI EA '19.

[6] J. Gerken, H.-C. Jetter, and H. Reiterer, "Using concept maps to evaluate the usability of APIs," in *Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems*, ser. CHI EA '16. New York, NY, USA: ACM, pp. 3937–3942.

[7] S. Endrikat, S. Hanenberg, R. Robbes, and A. Stefik, "How do API documentation and static typing affect API usability?" in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE '14. New York, NY, USA: ACM, pp. 632–642. [Online]. Available: https://doi.org/10.1145/2568225.2568299

[8] M. P. Robillard, "What makes APIs hard to learn? answers from developers," *IEEE Software*, vol. 26, pp. 27–34, 2009.

[9] S. Clarke, "Measuring API usability," *Dr. Dobbs Journal*, pp. S6–S9, 2004.

[10] E. Duala-Ekoko and M. P. Robillard, "Asking and answering questions about unfamiliar APIs: an exploratory study," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, pp. 266–276.

[11] M. P. Robillard and R. DeLine, "A field study of API learning obstacles," *Empircal Software Engineering*, vol. 16, pp. 703–732, 2011.

[12] J. Stylos and B. A. Myers, "The implications of method placement on API learnability," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, ser. SIGSOFT '08. New York, NY, USA: ACM, pp. 105–112. [Online]. Available: https://doi.org/10.1145/1453101.1453117

[13] J. Stylos, A. Faulring, Z. Yang, and B. A. Myers, "Improving API documentation using API usage information," in *2009 IEEE Symposium on Visual Languages and Human-Centric Computing*, ser. VLHCC '08, pp. 119–126.

[14] A. L. Santos and B. A. Myers, "Design annotations to improve API discoverability," *Journal of Systems and Software*, vol. 126, pp. 17–33, 2017.

[15] S. Thummalapenta and T. Xie, "Spotweb: Detecting framework hotspots and coldspots via mining open source code on the web," in *23rd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '08, pp. 15–19.

[16] J. Stylos and S. Clarke, "Usability implications of requiring parameters in objects' constructors," in *Proceedings of the 29th international conference on Software Engineering*, ser. ICSE '07. Washington, DC, USA: IEEE, pp. 529–539. [Online]. Available: https://doi.org/10.1109/ICSE.2007.92

[17] L. Beckwith, C. Kissinger, M. Burnett, S. Wiedenbeck, J. Lawrance, A. Blackwell, and C. Cook, "Tinkering and gender in end-user programmers debugging," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '06, pp. 231–240.

[18] J. Brandt, P. J. Guo, J. Lewenstein, and S. R. Klemmer, "Opportunistic programming: How rapid ideation and prototyping occur in practice," in *Proceedings of the 4th international workshop on End-user software engineering*, ser. WEUSE '08.

[19] V. Grigoreanu, M. Burnett, S. Wiedenbeck, J. Cao, K. Rector, and I. Kwan, "End-user debugging strategies: A sensemaking perspective," *Transactions on Computer-Human Interaction*, vol. 19, 2012.