Enabling Data-Driven API Design with Community Usage Data: A Need-Finding Study

Tianyi Zhang[†], **Björn Hartmann**[§], **Miryung Kim**[‖], **Elena L. Glassman**[†]
[†]Harvard University, MA, USA
[§]UC Berkeley, Berkeley, CA, USA
[‖]UC Los Angeles, Los Angeles, CA, USA
{tianyi, eglassman}@seas.harvard.edu, bjoern@berkeley.edu, miryung@cs.ucla.edu

ABSTRACT
APIs are becoming the fundamental building block of modern software and their usability is crucial to programming efficiency and software quality. Yet API designers find it hard to gather and interpret user feedback on their APIs. To close the gap, we interviewed 23 API designers from 6 companies and 11 open-source projects to understand their practices and needs. The primary way of gathering user feedback is through bug reports and peer reviews, as formal usability testing is prohibitively expensive to conduct in practice. Participants expressed a strong desire to gather real-world use cases and understand users' mental models, but there was a lack of tool support for such needs. In particular, participants were curious about where users got stuck, their workarounds, common mistakes, and unanticipated corner cases. We highlight several opportunities to address those unmet needs, including developing new mechanisms that systematically elicit users' mental models, building mining frameworks that identify recurring patterns beyond shallow statistics about API usage, and exploring alternative design choices made in similar libraries.

Author Keywords
API design; community; information needs; tool support

CCS Concepts
•Human-centered computing → Human computer interaction (HCI); Empirical studies in HCI; Interactive systems and tools;

INTRODUCTION
APIs are programming interfaces exposed by software development kits (SDKs); libraries and frameworks; and web services such as REST APIs and remote procedure calls [40]; they are one of the primary interfaces for programmers to give instructions to computers.[1] The use of APIs is ubiquitous, powering software applications, systems, and web services in nearly every domain. Given the increasing number and complexity of APIs, learning and using APIs is becoming a common activity and a key challenge in modern programming [27, 47, 48, 40].

User-centered design can produce usable APIs with great clarity, learnability, and programming efficiency [40, 39, 52]. Traditional usability testing methods such as user studies are often deemed too expensive to conduct during API design [19]. For example, a software library may have hundreds of APIs, which can be composed in various ways to implement different functionalities. Therefore, it is costly to conduct user studies to comprehensively evaluate a large number of possible usage scenarios of an API. In addition, recruiting participants is hard since users need to have adequate programming skills to use an API. These unique challenges make it appealing to leverage the sheer amount of API usage data that are already produced by the community, e.g., public code repositories, issue reports, and online discussions, to inform better API design. By grasping a comprehensive view of real-world use cases and common mistakes made by API users, API designers could adjust their design choices accordingly and close the gap between API design and user expectations.

Unfortunately, a recent study found that API designers still have difficulties in gathering and interpreting user feedback from their communities [37]. Large-scale community data has driven big success stories in domains such as bug detection [24, 60] and code completion [25, 45, 3], but remains under-utilized in the context of human-centered API design and usability evaluation. Prior work on API design and usability evaluation either focuses on small-scale methods that only involve a small group of stakeholders to review API design [52, 19, 32], or only leverages pre-defined heuristics that do not account for real usage scenarios or user feedback [10, 38]. There is also a lack of guidelines to enable API designers to make data-driven decisions based on the community usage data.

In this work, we investigate how community usage data could inform design decisions that account for real-world use cases and user feedback. We conducted semi-structured interviews with 23 API designers that worked on different kinds of APIs,

CHI '20, April 25–30, 2020, Honolulu, HI, USA.

ACM ISBN 978-1-4503-6708-0/20/04 ...$15.00.
http://dx.doi.org/10.1145/3313831.3376382

[1] While the term "API" is often specifically referred to web APIs in some literature, in this paper, we use "API" as a shorthand for all kinds of application programming interfaces mentioned above.

including software libraries (11), web APIs (6), and domain-specific languages and toolkits (6). Among those participants, we observed a spectrum of API designers, including *user-driven*, *visionary*, *self-driven*, and *closed-world*. Despite various design decisions and trade-offs made by different kinds of designers, all of them acknowledged the importance of keeping users in mind while building APIs. However, in practice, participants gathered user feedback in an informal and passive manner, primarily through peer reviews and bug reports. We identified a diverse set of unmet needs of API designers. For example, while existing API usage mining techniques [62, 59, 41, 22] mainly focused on identifying common patterns of correct usage, designers wished to gather more unanticipated corner cases and user mistakes from a broader scope of community data such as online discussions, emails, and issue reports. In particular, *user-driven* and *visionary designers* expressed a strong desire to get a holistic view of API usage in the wild and interactively validate their own hypotheses with real-world use cases. Therefore, it is important to extend existing mining techniques to address these unmet needs.

Furthermore, the majority of participants were interested in getting a rich description of users' mental models. For example, they wanted to understand "why do people use my API?", "how they discover it?", "what steps do they take to learn the API and how long?", and "where do users give up?". However, such information is barely reflected in code repositories and thus cannot be easily surfaced from a pile of users' code. Therefore, it requires developing new mechanisms to elicit rich, fine-grained descriptions of user feedback and mental models. For those designers who work on popular domains such as machine learning and data visualization, it may be beneficial to surface design decisions made in alternative libraries and identify features supported by alternative libraries but not by their own. This requires novel abstractions that model use cases of different libraries or DSLs in the same design space so that they are comparable regardless of syntactic differences.

In summary, this paper's contributions to HCI are:

- We conducted semi-structured interviews with 23 API designers from industry, academia, and nonprofit organizations to understand their practices and information needs in terms of gathering and interpreting user feedback.
- We presented an in-depth analysis of their design styles, design decisions, evaluation methods of API design, common issues of API usability, and unmet information needs.
- We proposed several tool design implications for leveraging community usage data to address the unmet needs of API designers and enable them to make data-driven decisions.

RELATED WORK

API Design and Usability

Building usable APIs is crucial to programming efficiency and software quality in modern programming. Previous studies have shown that programmers at all levels, from novices to experts, often find it difficult to learn new APIs [27, 40, 47, 15, 49]. Making mistakes in API usage could lead to severe program failures such as runtime exceptions [60] and security vulnerabilities [21, 18, 16]. Therefore, it is crucial to design APIs that meet user requirements and are easy to use, not only for the sake of driving API adoption and sustainability, but also to provide high-quality and reliable software products.

Prior work has investigated the usability impact of particular design decisions, e.g., the use of factory pattern [17], the use of default class constructors [51], and method placement [54]. Stylos and Myers proposed a list of API design decisions in object-oriented programming languages such as Java and C#, based on API design principles and recommendations from textbooks, guidelines, lab studies, and online discussions [53]. Macvean et al. described six challenges of designing and maintaining web APIs at Google, including resource allocation, empirically grounded guidelines, communication issues, API evolution support, API authentication, and library support across different programming languages [31]. There are also a number of studies that propose new methods to improve API usability, including user-centric API redesign [52], heuristic evaluation [10], API usability peer review [19, 32], and static code analysis [38]. Our work extends previous studies by characterizing current practices and unmet information needs of API designers. We propose a set of tool design implications for leveraging community-generated API usage data to inform data-driven design decisions.

Murphy et al. [37] interviewed 24 professional developers and identified the practices and challenges broadly related to API design workflow, API design guidelines, API usability evaluation, API design review, and API documentation. Our work is mainly motivated by a particular finding in Murphy et al.—*gathering user feedback and real-world use cases of APIs is useful but challenging due to a lack of tool support.* However, it is unclear what kinds of user feedback designers would like to discover from the API community, what kinds of information cues they consider valuable, and what tool support is needed to systematically surface those information cues from the community data. This motivates us to conduct an in-depth analysis of the unmet needs of API designers and how we could design better tool support. In addition, Murphy et al. only interviewed professional developers that worked in a company. In this work, we recruited a broader scope of API designers, including those from open-source projects, academia, and non-profit organizations. Compared with professional developers who have more resources and communication channels to their users, those open-source developers are in greater need of gathering and comprehending real-world use cases of their APIs.

Influence of Community Values on API Ecosystems

There has been a large body of empirical studies about how information signals from software communities such as GitHub can influence the decision making process in software development, e.g., which project to depend on [56], which developer to follow [11, 33], how to test a project [42], which project to contribute to [6, 44, 43]. Of particular interest to us are two recent studies about how community values influence the evolution and sustainability of API ecosystems [4, 58]. Bogart et al. [4] studied how community-specific values, policies, and tools may influence how developers respond to API changes that may break client code. They found that expectations

about how to handle changes differ significantly among three ecosystems, `Eclipse`, `CRAN`, and `npm`. For example, long-term stability was a key value of the `Eclipse` community, while developers in the `npm` community considered ease of development more important and thus were less concerned about breaking changes. Valiev et al. [58] analyzed how project-level factors (e.g., the number of commits, the number of contributors) and ecosystem-level factors (e.g., the number of upstream and downstream dependencies, the licensing policy) may influence the sustainability of Python packages in `PyPI`. These studies focus on the backward compatibility and sustainability of library packages. By contrast, we propose a new perspective on leveraging community usage data to enable API designers to make data-driven, community-responsive decisions.

Mining from Online Coding Communities

There has been a lot of interest in mining programming insights from online coding communities, e.g., API usage tips [29, 55], test cases [36, 34], analogical libraries [7, 8], bug fixing hints [9, 20, 61]. In particular, many software mining techniques focus on mining and visualizing common API usage patterns from large code corpora [62, 35, 59, 41, 60]. However, their goal is to teach programmers how to use an API, rather than helping API designers to glean useful insights about API design and usability. For example, Glassman et al. designed an interactive system called EXAMPLORE that visualizes hundreds of API usage examples so that users could quickly grasp the different ways of using an API in various scenarios [22]. Compared with the needs of API users identified by prior work [47, 15], this study shows that the information needs of API designers are quite different. For example, API designers often want to establish a comprehensive understanding of what mistakes users have made and where users were stuck before they eventually figured out the correct usage or even worse, before they gave up. However, a large portion of user mistakes are never committed to code repositories. Therefore, compared with mining correct API usage in code repositories, it may be more beneficial to build techniques that systematically gather API-related errors and identify error-inducing usage patterns.

PARTICIPANTS

We recruited 23 participants who built and maintained APIs using snowball sampling. Initially, we invited a small group of participants through personal contacts. Those participants further referred us to their colleagues and friends who also developed APIs and might be interested in our study.

Table 1 shows participants' information. The project names and descriptions were anonymized for participants working on proprietary APIs in a company. We also anonymized the occupation of participants to avoid the potential risk of disclosing their identities. Ten participants were from large technology companies, while thirteen were from academia and non-profit organizations. These participants worked on eleven open-source projects and seven proprietary projects, which spanned across various domains, including statistics, data science, machine learning, circuit design, formal methods, distributed systems, and probabilistic programming. Participants had a median of 6 years of API design experience (mean: 7 years). Regarding API types, eleven participants worked on software libraries and frameworks written in general-purpose programming languages; six participants worked on frameworks that provided domain-specific languages (DSLs); six participants worked on web APIs. All of these APIs were either public APIs with at least a hundred or even thousands of users, or proprietary APIs that were deployed within companies and used by internal teams or external customers. There was a wide spectrum of API users, e.g., regular programmers, data scientists, machine learning engineers, hardware designers, statisticians, computational biologists, ecologists, etc.

METHODOLOGY

We conducted semi-structured interviews, giving its advantage of allowing unanticipated information to be mentioned [30]. Each interview started with an introduction, a short explanation of the research being conducted, and demographic questions. Participants were then asked about the APIs they had worked on, the usability evaluation methods adopted in their projects, and their information needs if presented with a group of users and real use cases. We also asked about the API usability issues reported by their users, the challenges of gathering user feedback, and the desired tool support.

Interviews were conducted either in-person or via video conferencing software. Interviews took an average of 61 minutes, ranging from 30 minutes to 90 minutes. Three interviews were done in 30 minutes only, due to the time constraint of participants. We recorded each interview under the permission of participants. We transcribed a total of 23.3 hours audio recording of these interviews. The first author first conducted an open-coding phase over all transcripts using a qualitative analysis tool called MaxQDA.[2] This coding phase was done thoroughly by highlighting everything that is relevant or potentially interesting. A code was generated by summarizing a relevant phrase or sentence with a short descriptive text. Then the first author conducted an inductive thematic analysis [5] by grouping related codes into themes. The themes were regularly presented and discussed with the entire research team as they emerged from the interview data. In addition, the second author inspected the generated codes and themes separately, validating how the raw data supports the denoted codes and themes and adjusting their descriptions and boundaries. The two authors then discussed the disagreements and refined the codes and themes together across multiple sessions.

DESIGNER SPECTRUM AND DESIGN DECISIONS

Among 23 participants, we observed four types of API designers—*user-driven*, *self-driven*, *visionary*, and *closed-world*, based on how users fit into their design process and how they responded to user feedback. Column Design Style in Table 1 shows where each participant falls on the spectrum.

User-driven designers often proactively talked to their users, designed APIs based on user expectations, and were very responsive to user feedback. Many customer-facing API teams in large technology companies were user-driven. They often took a user-centered design process that solicited user feedback early via surveys or public proposals that everyone could

[2] https://www.maxqda.com/

Table 1. Participant Information

	Organization	Project	Description	Language	API Users	Design Style
P1	Large tech company	—	—	C#	Internal developers	visionary
P2	Open-source nonprofit	rOpenSci	A collection of R packages	R	Statisticians, computational scientists	self-driven
P3	Open-source nonprofit	rOpenSci	A collection of R packages	R	Statisticians, computational scientists	visionary
P4	Large tech company	—	—	Web API	Customers	user-driven
P5	R1 university	Chisel	A hardware design language & framework	DSL	Hardware designers	visionary
P6	R1 university	Vega	A data visualization language & library	DSL	Students, external users	self-driven
P7	R1 university	Chisel	A hardware design language & framework	DSL	Hardware designers	visionary
P8	R1 university	Chisel	A hardware design language & framework	DSL	Hardware designers	self-driven
P9	R1 university	rOpenSci	A collection of R packages	R	Students, statisticians, ecologists, etc.	self-driven → user-driven
P10	R1 university	Alloy	A model checking language & library	DSL	Students, academic researchers in PL, networking, embedded systems, etc.	self-driven
P11	Large tech company	—	—	Web API	Internal developers	closed-world
P12	Large tech company	—	—	Web API	Internal developers	closed-world
P13	Large tech company	MXNet	A deep learning library	Python	Data scientists, ML engineers	user-driven
P14	Large tech company	—	—	Web API	Internal developers	closed-world
P15	R1 university	Julia	A distributed arrays package in Julia	Julia	Computational scientists, economists, etc.	self-driven
P16	R1 university	Julia	A high-performance computing package	Julia	Computational scientists, economists, etc.	visionary
P17	R1 university	Partisan	A messaging library for distributed systems	Erlang	IoT developers, database developers	self-driven
P18	Large tech company	—	—	Web API	Internal developers	closed-world
P19	R1 university	Gen	A probabilistic programming framework	Julia	Students, researchers in ML and statistics	self-driven
P20	Large tech company	TensorFlow	A deep learning library	Python	Data scientists, ML engineers	user-driven
P21	Large tech company	—	—	Web API	Internal developers and customers	user-driven
P22	R1 university	Caffe	A deep learning library	Python	Data scientists, ML engineers	self-driven → user-driven
P23	Large tech company	Z3	A SMT solving library	DSL	Both industry developers and academic researchers in PL and systems	self-driven

comment on. The top concern of *user-driven designers* was API stability, especially if they had a large community. Once the design was finalized, user-driven designers were careful about changing their APIs, since it could bring the risk of breaking users' code. Yet for designers in other three categories, it was relatively flexible—they were more willing to first implement core features and then iterated on the API design based on user feedback.

Self-driven designers made their own design decisions based on their domain knowledge and designed APIs that suited their own needs first. Though most of them were interested in listening to their users, they often had their own goals and priorities. Thus, if user feedback was not aligned with their own goals, they were less willing to change their design decisions. Compared with making simple things easier, self-driven designers cared more about making hard things possible. They were excited about adding more functionalities to their APIs and making their APIs more capable. Therefore, they cared a lot about the extensibility of their API design and did not want to limit themselves in the future. In fact, some of them introduced extra layers and interfaces to make their APIs modularized and extensible, which caused indirections that hindered API usability.

Visionary designers put themselves in the shoes of users and envisioned potential use cases when designing APIs. Oftentimes, they had a good wish of building simple and easy-to-use APIs but lacked direct communication channels to their users. Open-source developers that did not have close collaborations with industry were often visionary. Because their users were often external users and they also had limited resources to connect to their users, e.g., organizing annual workshops as TensorFlow and MXNet teams did. Since *visionary designers* did not have a good idea about the API use cases in the wild, they had a tendency to make conservative design decisions such as adding strict runtime checks. If users used an API incorrectly, these runtime checks would fail early and prevent users from shooting themselves in the foot.

Finally, *closed-world designers* often built APIs to support other teams in the same organizations, where all design goals and use cases were clearly set and teams worked cooperatively to decide details that suited both designers and users. The user teams were deeply involved in the design process, where designers and users would sit down for weeks to sort out potential use cases together. Therefore, compared with other kinds of API designers, there was less tension between *closed-world designers* and their users. Oftentimes, stability is not really a big concern for them. Since they were closely connected to their users, they could inform user teams about proposed changes first and iteratively adjust their API design as needed.

Note that those API design styles were not always exclusive to each other. Many participants were, to some extent, a mix of different design styles at different stages of their projects. For instance, developers from early-stage academic projects were often *self-driven*, since their main goal was to build cut-edging technologies to achieve their own research goals rather than growing as a community. However, as those projects got more users and made more impact, those designers became more *user-driven*.

USABILITY EVALUATION METHODS IN PRACTICE

To understand the challenge and needs of evaluating API usability, we first asked participants how they evaluated their API design and gathered user feedback in practice. Specifically, we focused on eliciting different approaches and resources that our participants attempted to use. Table 2 shows a complete list of evaluation methods that participants adopted.

Looking through bug reports, emails, and online discussions. For all kinds of API designers, inspecting bug reports, emails, and online discussions was the major way to identify usability issues. Most participants just passively relied on users to submit bug reports or ask questions, rather than actively reaching out to users. One of the reasons was the lack of communication channels to their users, especially for visionary designers. Though closed-world designers could easily reach out to user teams in the same company, they also waited for users to come back to them since it was more efficient as one participant said. By just looking through bug reports, API designers could get a vague sense about which API was used more often and which API was more error-prone. Yet they did not get a comprehensive view of the real-world use cases and usability issues, which many participants wished to get tool support for.

"*The most important thing is a lot of users post issues if they run into problems. So in that way, you also learn about what they're doing and problems that they were in to. Sometimes people share what they've done on social media. Or they write a blog post about what they're doing. So nothing is in a formal way but you get a sense of how people are writing their software with your package.*" [P2]

Peer reviews. Another widely adopted mechanism was to conduct peer reviews. Peer reviews helped with identifying design inconsistencies and potential use cases that were initially unanticipated by designers. Compared with open-source projects, large technology companies had more formal and rigorous review process, where all stakeholders were invited to review the API design. Big companies also had well-established review guidelines for API design. For instance, Google published a list of general design principles for web APIs, e.g., naming clarity and consistency, error handling principles, etc [1]. Open-source teams adopted light-weight code reviews via pull requests. Several participants said their teams did not perform peer reviews since it was difficult to find good reviewers in open-source projects.

Regression testing on client applications. Six participants manually searched and curated a small set of applications that used their APIs. Everytime they made changes to APIs, they reran test cases in those applications to ensure backward compatibility of updated APIs. Package managers such as `CRAN` and `PyPI` exposed reverse dependencies, which could help identify a large number of downstream packages. However, participants were reluctant to run regression against all downstream applications. One participant explained that it would cause too much computation overhead. Another participant said he did not want to lock himself in the corner to handle all kinds of issues in downstream applications. This required an intelligent way to select a subset of representative client projects to test on. Mining-based approaches could help with identifying various API usage examples from large code corpus [62, 59, 60]. Yet it still remains a challenge to efficiently visualize a large volume of usage examples and enable designers to filter out corner cases that they do not care about. Such a technique would be extremely beneficial for visionary designers, since they had limited resources to connect to their users and often desired to identify real use cases in the wild.

Table 2. Usability Evaluation Methods in Practice

Methods to evaluate API usability and get feedback	Num
Looking through bug reports, emails, online discussions	18
Peer review	13
Regression testing on existing usage	6
Teaching in classrooms, training camps, and workshops	5
A/B testing and its alternatives	4
Building example galleries	5
Scheduling regular meetings with users	4
Monitoring and logging web API traffic	6
Sending out surveys	1
Cognitive dimensions	1
User studies	1

Teaching in classrooms and training camps. For participants from academia, a primary alternative to user studies was to teach libraries and packages in classrooms or training camps. Participants found it very helpful to have users within their arm's length and watch over their shoulders.

"*Being able to teach (my R packages) in the class room has been fantastic because I speak to students who have no programming experience or encounter the task for the first time. And I see the way they're trying to make things work, and what works and what fails. You immediately start seeing the pitfalls that you haven't anticipated.*" [P9]

A/B testing and its alternatives. Four participants said they did A/B testing or something similar to assess alternative API designs. Two web API designers mentioned that when they refactored an API or introduced a new API end point to replace an old one, they would add an interceptor to transform a subset of the incoming API requests to fit the data model of the new API and re-route them to the new API. Then they collected server-side metrics and compared the performance of two APIs. Compared with web APIs, it was difficult to instrument library APIs to gather API usage metrics. Two participants from the Chisel project said their team would mark new API alternatives as experimental before rolling them out. In this way, API users could experiment with those APIs and comment on them. However, this was not successful because some users built real products on top of those experimental APIs, which made it hard for Chisel developers to withdraw or revise those APIs based on the comments from other users.

Scheduling regular meetings and organizing workshops. Industrial developers often had more direct access to their users compared with open-source developers. Several participants from industry said they organized quarterly conferences with their customers to hear feedback. Participants who worked on

large open-source projects like Chisel, MXNet, and TensorFlow also organized annual workshops to demonstrate new features in their libraries and conduct informal interviews and surveys with attendees.

Building example galleries. Five participants manually curated a set of API usage examples by themselves based on the real or envisioned use cases of their APIs. In this way, they established good understanding about different usage scenarios and how easy it was to use their APIs in those scenarios. However, they found it hard to gather a comprehensive set of use cases in practice. When an unanticipated use case was identified, they also had difficulties assessing how representative this use case was and how many other developers may also use their APIs in the same way.

Monitoring API traffic. All six web API designers either used API gateways such as Apigee [2] or built their own server-side instrumentation to log API traffic, e.g., API headers and payloads in an API request or response. They queried the log data to understand real use cases and track errors. For example, by analyzing the device types captured in the API headers, developers could understand the distribution of different types of applications that sent this request. If a lot of requests were sent from Android devices, developers may decide to focus more on Android apps. However, web API designers found it difficult to reconstruct underlying user interactions and mental models from low-level, fragmented log data, especially given the enormous volume of log data.

Sending out surveys. P5 said their team tried to post surveys both online and at training camps to collect user feedback. However, the participant complained that gathering insightful feedback was very difficult—"*Every time we run the boot camp, we try to run a survey. Sometimes people respond but that didn't tell me anything. I am definitely very much a fan of making data-driven API decisions but somehow collecting feedback from people so far has not been super practical.*"

Cognitive dimensions. P6 used cognitive dimensions [23, 10] to evaluate the usability of the domain-specific language in their library. The participant explained, "*we only theoretically evaluate it with cognitive dimensions mostly because it's hard to know how to actually evaluate abstractions in a sort of controlled or semi-controlled way.*"

User studies. P11 was the only participant whose API was, to some extent, evaluated by user studies. In their company, there was a UX team who invited pilot customers to try out UIs built on top of their API. The UX team then shared insights such as which features or data provided by the API were never used by users in practice and thus should be removed. Other participants did not conduct user studies for several reasons. For *closed-world API designers*, design teams and user teams created API designs together cooperatively, which made it unnecessary to further conduct user studies. *User-driven designers* and *visionary designers* often found it hard to recruit a good sample size of users. Many API teams also lacked expertise in human-centered usability evaluation methods. Some participants preferred to understand real-world use cases in the wild, rather than observing user behavior in a controlled or semi-controlled setting.

COMMON USABILITY ISSUES

We asked participants what API usability issues they have encountered and identified five common usability issues. Understanding common usability issues could help us assess whether a data-driven approach could help with identifying and solving these issues.

Unanticipated use cases. API users often suggested new features or reported issues in cases that designers never anticipated. This barely occurred to *closed-world designers*, but it occurred quite often for other types of API designers, especially if they had a large user base. For example, P13 said that a major decision in MXNet was to represent every data type as tensors (i.e., multi-dimensional arrays). As a result, scalar variables were not supported in MXNet. But recently, the participant found that many users used `1x1` tensors to represent scalar variables. This led to many performance issues in client code, since most tensor-level optimizations in MXNet were not applicable to `1x1` tensors. Participants also explained that, in practice, it was impossible to support all use cases due to limited time and resources. Therefore, they wished to get a comprehensive view of real use cases and figure out how representative a corner case or an issue is. If there was a tool that enabled the MXNet team to interactively discover all common and uncommon ways of using their APIs, they could make an early decision about supporting scalar variables.

Heavy-lifting API usage. Participants mentioned that they also discovered complex API usage that could be simplified via API refactoring. For example, P12 said their web API required a large payload to represent a complex data model, which was cumbersome to construct manually. To make it convenient for users, their team exposed another API that programmatically constructed a payload template with some data populated. To address this type of issues, it would be beneficial to provide tool support for mining recurring, complex API usage patterns to refactor.

Confusing names and terminologies. There were two kinds of confusion. First, designers named a function or a parameter with a general name that was interpreted differently by different users. Second, designers used names or terminologies that were inconsistent from common idioms in the same domain. For instance, P13 mentioned that their team named a loop transformation as "loop reorder" while some alternative libraries used "loop interchange" instead, though the underlying functionality was essentially the same. Interestingly, P18 surveyed all alternative libraries in the market and followed the most common naming conventions, terminologies, and interface definitions so that users could easily pick up their APIs without unnecessary confusion. This type of issues could be difficult to mine from static code repositories but could be recognized by analyzing user questions and issue reports. Therefore, it may be beneficial to investigate how integrate natural language processing methods into existing API usage mining techniques to analyze a broader scope of community usage data such as issue reports and questions beyond code. It

may also be beneficial to extend existing techniques to account for alternative libraries and check for naming consistencies.

Ambiguous API usage. API usage ambiguity was manifested in several ways. First, there were alternative ways of using an API to achieve the same goal, while users were unclear which one to use. Second, the same API usage could produce different outputs in different contexts, making it hard to understand its real meaning. Figure 1 shows an example in Chisel, where `Reg(UInt(3))` could be interpreted as one of the four things on the right side. This ambiguity was later fixed in Chisel 3.0.0 by making distinct usage explicitly. Third, sometimes users were caught up in the subtlety between overloading functions when there were many of them. Systematically identifying ambiguous API usage was challenging, since it required checking for behavioral similarities and variations among API usage examples. Thus, building new techniques that utilize dynamic analysis to disambiguate API usage was much needed.

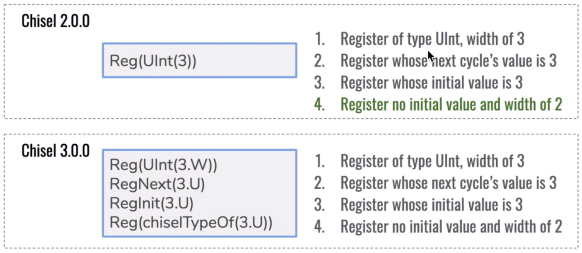

Figure 1. Ambiguous API usage in Chisel 2.0.0 that is fixed in 3.0.0.

During the interview, the majority of participants touched upon the tension between API designers and users. The tension often came from the fact that designers made certain implicit design decisions and tradeoffs that were not articulated in documentation. Many APIs were initially designed for a small group of users. As an API community grew bigger, participants found that more and more novice users, who were not aware of the initial design decisions and tradeoffs, started picking up their libraries. Of course, many of aforementioned usability issues could be addressed by clear documentation. Yet participants expressed several challenges related to API documentation. First, it was too much effort to always keep documentation up-to-date. Second, it was also difficult to track all representative use cases and include them in documentation. Third, participants were worried about the discoverability of important information if the documentation became too long, since not all users would read it carefully. Hence, participants expected not only tool support for identifying real-world use cases and usability issues but also new mechanisms for information delivery.

INFORMATION NEEDS

To solicit the unmet needs of API designers, we asked participants to reflect on what information cues they found difficult to discover from existing community data and what other data they wish to gather though such data may not even exist. Overall, we found that the information needs of API designers were quite different from the needs of API users identified by prior work [47, 15]. API users wanted to know about the mechanics of correctly using an API, such as what other APIs to use together and what exceptions to handle. By contrast, API designers were more curious about users' mental models and the mistakes made by users. Furthermore, API users were more like a "blank slate" when first learning an API, while API designers had already established implicit assumptions about how users should and should not use their APIs. Therefore, API designers (especially *visionary designers*) were eager to validate their own hypotheses of API usage with real use cases and identify unanticipated corner cases. This section elaborated on eight common needs of API designers. Table 3 shows these information needs and their frequencies.

Table 3. Unmet or Partially Met Information Needs of API Designers

Information needs	Num
A holistic view of real use cases	15
A rich description about user's mental models	12
Common mistakes and workarounds	11
API call frequency	9
Behavioral metrics, e.g., runtime states, performance	9
Backward compatibility	7
Comparing and assessing similar libraries	7
Design inconsistencies	6

A holistic view of real use cases. The majority of participants (15/23), mostly *user-driven designers* and *visionary designers*, expressed a strong desire to get a holistic view of real use cases. Participants were curious about the recurring patterns among these use cases. Because recurring patterns indicated refactoring opportunities to simplify their API design. For instance, P13 mentioned that many data scientists used APIs in another library called `NumPy` together with APIs in `MXNet` for data processing, which required extra code to convert data structures between `NumPy` and `MXNet`. Thus the participant planned to provide new APIs that seamlessly integrated with `NumPy` without requiring users to write additional code. Participants were also curious about those corner cases that they did not anticipate in the initial design. While previous mining techniques only focus on mining common API use cases, identifying corner cases was also important to inform API design.

"*I'd like to get is looking at their code and see if they write code in the ideal way we want them to write. Because every API has its own purpose when they're being designed, and we have some use case in mind that this APIs should be used in this way. So I would like to look into their code and see if they use the APIs in our way. If not, we'll probably think if this API is designed in a proper way or we should create some more obvious APIs to deal with this case.*" [P13]

A rich description of users' mental models. Twelve participants wished to gather several major clues to understand users' mental models, including the intent of using an API, how the API was discovered, what steps it took to learn the API, how long it took, where users were stuck, and where users gave up. By understanding users' mental models, API designers were able to identify inconsistencies between API design and users' expectations and adjust their API design to reduce confusions and learning barriers.

"*I want to get what people are thinking, what's going under head, not just whether they are able to use this feature, like that's what you would get from like mining GitHub repos. But what you are not getting is like this person still spent three hours googling how to use this feature, and he spent an hour on Stack Overflow trying to figure out what was going on, in the end, he got that, but it was a frustrating attempt.*" [P6]

P9 said teaching his R packages in classrooms and watching students over their shoulders made him quickly understand what students tried, what worked, and what failed. Yet not every API designer could easily get a room of users to learn their APIs. Several participants mentioned that they could get a sense of where users were stuck by reading questions users asked in emails and online forums. However, there was no easy way to comprehensively understand the whole picture of users' mental models at a low cost.

Common mistakes and workarounds. Compared with correct API usage, eleven participants mentioned that they were more curious about what mistakes users made before figuring out the correct usage. Participants wanted to find out whether a user ran into the same error repeatedly, whether different users made the same mistake, how easy it was for a user to make a mistake, and what workarounds users made.

Participants mentioned that compilation errors and runtime exceptions were good indicators of confusing API design. Even though those errors were prevented from creeping into production code, bad API design was never solved in the first place. As a result, novice users may repetitively make the same mistake. Participants in the Chisel team mentioned that they implemented a fairly strict type system as well as many runtime checks to fail potential errors early. They wondered which runtime checks were triggered more often and whether a user encountered the same error repeatedly. Because if a user repeatedly made the same mistake, it may indicate the API design was so counter-intuitive that users cannot easily adjust their mental models to the correct usage. Instead of mining all recurring patterns, P14 suggested to identify patterns that were likely to lead to program crashes and errors. Because error-inducing patterns were more actionable than showing all common patterns.

Though some issues were submitted as bug reports or asked on online forums, participants suspected a large portion of user mistakes were not reported at all. However, such information was barely committed and thus cannot be easily discovered from code repositories. As a result, this calls for a new mechanism to systematically gathering API-related errors and inferring error-inducing patterns.

"*I think the people that use the software with no prior experience perhaps hit the wall quickly. Every now and then you do see those issues of someone, like just never use R packages before. That's the real issue. But for the most part, I suspect that I missed most of the easy problems going on there because they don't convey that first barrier with their remote communication.*" [P9]

API call frequency. Nine participants wanted to understand API call frequency for different reasons, e.g., estimating the impact of deprecating an API, prioritizing efforts, allocating computation resources for web APIs, etc. However, two participants warned about recency bias when interpreting API call frequency. Because new APIs may not have accumulated as many use cases as existing APIs.

"*I think even you can just get users' code and see which API is more popular. There are still some concerns. For example, in our case, an alternative API is relatively new. So if we just look at code, I think it's hard to judge which API is better. Maybe people just don't know there is another way to implement that. So I think making a poll makes more sense because now people know we have these two options and which one they prefer.*" [P13]

Web API designers and *closed-world designers* can easily calculate API call frequency by monitoring web API traffic or searching over the internal code base in a company. However, for public library APIs, it was hard to estimate the number of call sites in the wild. P3 said, "*I used to search on GitHub more often, but I just found a lot of false positives. If a function name is like a common word, then it's sort of a huge pile, and it's kind of finding a needle in a haystack.*" Thus it was necessary to resolve function names precisely, which was challenging for loosely typed languages such as R and JavaScript. In addition, five participants also suggested to include context information such as exception handling logic and surrounding API calls beyond just API call frequency.

Behavioral metrics beyond static usage. Instead of just statically inspecting use cases, nine participants wanted to gather more behavioral metrics. For instance, they wanted to know the values of function arguments or data fields in an API request, whether these values were always constant, and whether these values were in a valid range. Several participants also mentioned that performance was a major factor that affected their API usability, e.g., latency for web APIs, slow training in deep learning. Therefore, they would like to collect performance metrics from users' code.

Backward compatibility. Backward compatibility was the top concern of *user-driven designers*. Participants were not only concerned about compilation errors caused by interface changes, but also behavior and performance inconsistencies caused by the implementation changes of an API. Participants said they were more okay if an API change broke client code in a *noisy* way, e.g., throwing compilation errors, runtime exceptions, or test failures. Because at least users knew that something went wrong and should be fixed. However, participants were more concerned about silent changes in behavior caused by API updates.

In practice, six participants said they curated a small set of client applications or historical API requests to detect potential code breakage. However, participants found it difficult to understand backward compatibility at a large scale. Participants wished to know how much code breakage users were willing to tolerate and how easy it was to fix broken code. Because many code-breaking API changes were beneficial in a long term, though they may lead to code breakage at the moment.

Comparing and assessing similar libraries. Seven participants would like to compare their own libraries with similar libraries in the market. There were two important perspectives of comparing and assessing similar libraries. First, API designers would like to identify features that were well supported by alternative libraries but not by their own. Participants working on deep learning frameworks such as TensorFlow and MXNet were interested in finding out new neural network models that were integrated into their competitors. Second, API designers were curious about how easy a feature was supported by their own libraries in comparison to other libraries. For example, P6 was curious about whether a visualization was easy to express in D3 or Tableau but was cumbersome to construct in Vega.

Design inconsistencies. Six participants wished to identify design inconsistencies to the design choices they made previously or common choices made by other API designers. In fact, it was difficult to always conform to the same design style consistently. For example, P9 mentioned that, when creating functions in his R packages, he followed the tidyverse style guide but sometimes failed.[3] Such design inconsistencies were often caught by his students when he taught his R packages in classrooms. P12 mentioned that he had to manually search for other APIs in the company and followed their design to ensure the design consistency. Therefore, when there are no explicit design guidelines, it may be beneficial to expose design decisions and trade-offs made by other API designers.

OPPORTUNITIES FOR TOOL SUPPORT

The unmet needs of API designers imply many opportunities for building interactive systems that help developers gather, interpret, and consume community usage data to inform better API design. Some unmet needs require extending existing techniques to discover more information cues from a broader scope of community usage data beyond code, while some require building new mechanisms to collect new data that do not exist. We highlight several design implications below.

Mining and visualizing API usage beyond syntactic features and frequencies. There is a lot of literature on mining common patterns from code repositories [62, 35, 59, 41, 60]. However, prior work mostly focuses on finding syntactic patterns of API usage. The unmet needs of API designers in the previous section reveal several desired information cues beyond syntactic API usage. We synthesize a list of design principles for surfacing such information cues.

- API designers consider error-inducing patterns more actionable than common, correct API usage.
- In addition to syntactic usage, it is also important to show behavioral metrics to help API designers understand the semantics of use cases, e.g., runtime parameter values, call stacks, program states, performance, etc.
- It is helpful to include the program context of a code example such as preceding and post API calls, so API designers can better understand what users were trying to do and why their APIs were used in a specific usage scenario.
- Going through a pile of lengthy code examples is overwhelming, so it is important to keep unrelated code folded.
- Recency bias should be taken into account when interpreting API usage frequencies.

Furthermore, it is beneficial to analyze the proliferation of other kinds of community data besides code repositories, such as online discussions, tutorials, bug reports, and emails. In fact, participants mentioned that they primarily looked through online discussions and bug reports to understand what users are confused about and what mistakes users have made. For example, questions in Stack Overflow often ask about exceptional behaviors and boundary conditions that cause program failures when using an API. Therefore, we could extend existing mining infrastructures to analyze non-code data using natural language processing techniques.

New elicitation mechanisms to understand users' mental models. One major unmet need of API designers is to get a rich description about users' mental models. In fact, what users wrote and committed to their repositories does not reflect how much they suffered when learning and using an API. Existing approaches either elicit users' mental models directly via interviews [57, 26] or indirectly through questionnaires [28, 50], which is deemed to be expensive and unscalable given the large number of APIs and use cases. Instead, we recommend leveraging software instrumentation to gather fine-grained telemetry data at scale under users' permission. For instance, we can instrument browsers and IDEs to record visited learning resources, error messages, and user actions. Such instrumentation methods can also be further augmented with prompting questions to directly solicit explanations on API usage, though that would require further investigation of interruption styles [46] to efficiently solicit explanations without causing harm to users' productivity.

It may also be possible to infer users' mental models from low-level telemetry data. For instance, if a user never encountered an error again after resolving the error, it may indicate that she could quickly grasp the correct API usage after seeing the error message. On the other hand, if a user encountered the same error repeatedly, it may indicate a big gap between the API design and the user's expectations such that the user cannot easily adjust her mental model. One possible approach is to build a probabilistic model to approximate users' mental states conditioned on a sequence of low-level events.

Interactively analyzing population-level API usage. *Visionary designers* expressed a strong desire for validating their own hypotheses of API usage with real-world use cases of their APIs. Four participants suggested going beyond inspecting the common patterns identified by a mining infrastructure, since there could be many patterns. They wanted to be able to query the corpus of real-world use cases to evaluate their hypotheses of specific patterns, e.g., how often two APIs were used together, how often an API argument was set to a specific value. Text-based query interfaces that use keywords or regular expressions cannot easily express structural and semantic properties in a code pattern. EXAMPLORE [22] consolidates unique API usage features extracted from a large collection of code examples into a single API skeleton and allows developers to quickly filter the code corpus by selecting desired features in the skeleton. However, its expressiveness falls short in several

[3] Tidyverse Style Guide, https://style.tidyverse.org/

aspects. First, it cannot model structural properties such as "a method call must occur in a loop." Second, designers cannot specify alternative feature options in a pattern, which is useful to reason about alternative API usage. Third, designers cannot reason about the absence of a feature in a pattern, e.g., "how many use cases only call `lock` but forget to call `unlock`?" Therefore, it is beneficial to build new interactive mining techniques that (1) enable users to specify the structural properties of interesting API usage (i.e., *structural patterns*), (2) support disjunction of alternative API feature options in a pattern (i.e., *alternative patterns*), and (3) support negation of a feature option in a pattern (i.e., *negative patterns*).

Exploring the design space of similar APIs. Two particular exploration tasks emerged from the interview study of 23 API designers. First, participants would like to know concrete design choices, e.g., naming conventions, the number of parameters, the adopted design patterns. Second, participants would like to identify those features that are well supported by alternative APIs but are missing or not well supported by their own. Supporting the first task could be relatively straightforward by extracting individual dimensions from interface definitions and architectures. However, the second task is more challenging since it requires a meta abstraction to make concrete use cases of different libraries or DSLs comparable. For instance, to compare features between data visualization libraries such as D3 and Vega, we may want to abstract a visualization script as input-output pairs (i.e., input csv files and output graphs) and completely ignore the implementation details in the script. In this way, we will be able to compare D3 and Vega scripts despite their syntactic differences, and find out what types of graphs are visualized more often using one library but not the other.

Live API documentation. Writing API documentation and keeping it up-to-date is challenging [12, 13]. Online discussions, tutorials, bug reports, and emails could serve as insightful crowdsourced documentation. Six participants said if they had access to real use cases, they would like to use those examples to augment their API documentation. Therefore, it may be beneficial to build new techniques that systematically glean real-world use cases and discussions related to an API and constantly update its documentation. For example, if a new trending usage scenario of an API emerges in the community, the documentation should be updated to reflect the trend. On the other hand, only keeping a pile of code examples and related discussions may not be useful, since users may not have time to read all of them. It is important to distill representative families of examples and insightful comments.

LIMITATION

In this paper, we shared observations and themes that emerged in interviews with 23 API designers. Because experts in qualitative studies specifically warned about the danger of quantifying inherently qualitative data [14], we did not make any quantitative statements about how frequently the themes may occur in a broader population. We followed this guideline to focus on providing insights that contextualize our empirical findings, especially when they can serve as the basis for further studies, such as surveys.

We strove to include API designers from different fields that work on different kinds of APIs and projects. Therefore, we believe the nature of identifying information needs in this context is meaningful. As future work, we plan to encode those findings in a questionnaire and conduct a large-scale survey study of API designers. The survey will ask participants to rate the importance of each finding, including usability evaluation practices, information needs, and desired tool support in 7-point likert scale. We will also solicit more insights through open-ended questions in the survey.

Note that we do not argue that usability concerns should always be placed first when designing an API. As participants mentioned, there is always a trade-off between "making hard things possible" and "making simple things easy." If API designers focus too much on making simple things easy and addressing all corner cases, it is possible to lock themselves into a corner where they could never get hard things done. API design is an iterative process. Many API designers want to get the core features done first without concerning themselves much about API usability. As they get more users and more time, they will start looking at their APIs more closely and respond to usability issues reported by their users. It is beneficial to establish a comprehensive understanding of real use cases and potential gaps between API designers' and users' expectations during the iterative design process. We argue that it is important to keep usability in mind and make implicit design decisions explicit, so API designers do not inadvertently create an unusable API, and if they have to do so, do it knowingly with minimal cost.

CONCLUSION

This paper presents an in-depth analysis of unmet information needs of, and desired tool support for, API designers in terms of gathering and interpreting user feedback at scale. Given the diversified and distributed nature of modern API communities, it is generally challenging to establish a comprehensive view of real use cases, which the majority of participants desired to have. Due to a lack of tool support for collecting community usage data at scale, participants often gathered use cases and user feedback in an informal way, primarily through bug reports and online discussions. Participants also expressed a strong desire to understand users' mental models, while there is no systematic elicitation method to do so at scale without human intervention. This study provides empirical insights and design implications on how to build interactive systems to enable API designers to make more data-driven decisions based on real use cases and user feedback at scale.

ACKNOWLEDGMENTS

We would like to thank anonymous participants for the interview study and anonymous reviewers for their valuable feedback. This work is in part supported by NSF grants CCF-1764077, CCF-1527923, CCF-1723773, ONR grant N00014-18-1-2037, Intel CAPA grant, and Samsung grant.

REFERENCES

[1] 2017. Google API design guide. (2017). `https://cloud.google.com/apis/design/` Accessed: 2019-08-29.

[2] 2019. Apigee: A Cross-Cloud API Management Platform. (2019). https://cloud.google.com/apigee/ Accessed: 2019-08-29.

[3] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2016. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989* (2016).

[4] Christopher Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. 2016. How to break an API: cost negotiation and community values in three software ecosystems. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 109–120.

[5] Virginia Braun and Victoria Clarke. 2006. Using thematic analysis in psychology. *Qualitative research in psychology* 3, 2 (2006), 77–101.

[6] Casey Casalnuovo, Bogdan Vasilescu, Premkumar Devanbu, and Vladimir Filkov. 2015. Developer onboarding in GitHub: the role of prior social links and language experience. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering*. ACM, 817–828.

[7] Chunyang Chen, Zhenchang Xing, and Yang Liu. 2019a. WhatâĂŹs spainâĂŹs paris? mining analogical libraries from q&a discussions. *Empirical Software Engineering* 24, 3 (2019), 1155–1194.

[8] Chunyang Chen, Zhenchang Xing, Yang Liu, and Kent Long Xiong Ong. 2019b. Mining likely analogical apis across third-party libraries via large-scale unsupervised api semantics embedding. *IEEE Transactions on Software Engineering* (2019).

[9] Fuxiang Chen and Sunghun Kim. 2015. Crowd debugging. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 320–332.

[10] Steven Clarke. 2005. Describing and measuring API usability with the cognitive dimensions. In *Cognitive Dimensions of Notations 10th Anniversary Workshop*. Citeseer, 131.

[11] Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. 2012. Social coding in GitHub: transparency and collaboration in an open software repository. In *Proceedings of the ACM 2012 conference on computer supported cooperative work*. ACM, 1277–1286.

[12] Barthélémy Dagenais and Martin P Robillard. 2010. Creating and evolving developer documentation: understanding the decisions of open source contributors. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 127–136.

[13] Barthélémy Dagenais and Martin P Robillard. 2012. Recovering traceability links between an API and its learning resources. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 47–57.

[14] Norman K Denzin and Yvonna S Lincoln. 2011. *The Sage handbook of qualitative research*. Sage.

[15] Ekwa Duala-Ekoko and Martin P Robillard. 2012. Asking and answering questions about unfamiliar APIs: An exploratory study. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 266–276.

[16] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. 2013. An empirical study of cryptographic misuse in android applications. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 73–84.

[17] Brian Ellis, Jeffrey Stylos, and Brad Myers. 2007. The factory pattern in API design: A usability evaluation. In *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 302–312.

[18] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. 2012. Why Eve and Mallory love Android: An analysis of Android SSL (in) security. In *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 50–61.

[19] Umer Farooq, Leon Welicki, and Dieter Zirkler. 2010. API usability peer reviews: a method for evaluating the usability of application programming interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2327–2336.

[20] Qing Gao, Hansheng Zhang, Jie Wang, Yingfei Xiong, Lu Zhang, and Hong Mei. 2015. Fixing recurring crash bugs via analyzing q&a sites (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 307–318.

[21] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. 2012. The most dangerous code in the world: validating SSL certificates in non-browser software. In *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 38–49.

[22] Elena L Glassman, Tianyi Zhang, Björn Hartmann, and Miryung Kim. 2018. Visualizing API usage examples at scale. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. ACM, 580.

[23] Thomas R. G. Green and Marian Petre. 1996. Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Journal of Visual Languages & Computing* 7, 2 (1996), 131–174.

[24] Natalie Gruska, Andrzej Wasylkowski, and Andreas Zeller. 2010. Learning from 6,000 projects: lightweight cross-project anomaly detection. In *Proceedings of the 19th international symposium on Software testing and analysis*. ACM, 119–130.

[25] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 631–642.

[26] Amber Horvath, Mariann Nagy, Finn Voichick, Mary Beth Kery, and Brad A Myers. 2019. Methods for Investigating Mental Models for Learners of APIs. In *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems*. ACM, LBW0158.

[27] Andrew J Ko, Brad A Myers, and Htet Htet Aung. 2004. Six learning barriers in end-user programming systems. In *Visual Languages and Human Centric Computing, 2004 IEEE Symposium on*. IEEE, 199–206.

[28] Todd Kulesza, Simone Stumpf, Margaret Burnett, and Irwin Kwan. 2012. Tell me more?: the effects of mental model soundness on personalizing an intelligent agent. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 1–10.

[29] Hongwei Li, Sirui Li, Jiamou Sun, Zhenchang Xing, Xin Peng, Mingwei Liu, and Xuejiao Zhao. 2018. Improving api caveats accessibility by mining api caveats knowledge graph. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 183–193.

[30] Robyn Longhurst. 2003. Semi-structured interviews and focus groups. *Key methods in geography* 3 (2003), 143–156.

[31] Andrew Macvean, Luke Church, John Daughtry, and Craig Citro. 2016a. API Usability at Scale.. In *Psychology of Programming Interest Group*. 26.

[32] Andrew Macvean, Martin Maly, and John Daughtry. 2016b. API design reviews at scale. In *Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems*. ACM, 849–858.

[33] Jennifer Marlow, Laura Dabbish, and Jim Herbsleb. 2013. Impression formation in online peer production: activity traces and personal profiles in github. In *Proceedings of the 2013 conference on Computer supported cooperative work*. ACM, 117–128.

[34] Phil McMinn, Muzammil Shahbaz, and Mark Stevenson. 2012. Search-based test input generation for string data types using the results of web queries. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, 141–150.

[35] Evan Moritz, Mario Linares-Vásquez, Denys Poshyvanyk, Mark Grechanik, Collin McMillan, and Malcom Gethers. 2013. Export: Detecting and visualizing api usages in large source code repositories. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 646–651.

[36] Manish Motwani and Yuriy Brun. 2019. Automatically generating precise Oracles from structured natural language specifications. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 188–199.

[37] Lauren Murphy, Mary Beth Kery, Oluwatosin Alliyu, Andrew Macvean, and Brad A Myers. 2018. API Designers in the Field: Design Practices and Challenges for Creating Usable APIs. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 249–258.

[38] Emerson Murphy-Hill, Caitlin Sadowski, Andrew Head, John Daughtry, Andrew Macvean, Ciera Jaspan, and Collin Winter. 2018. Discovering API usability problems at scale. In *Proceedings of the 2nd International Workshop on API Usage and Evolution*. ACM, 14–17.

[39] Brad A. Myers. 2017. Human-centered Methods for Improving API Usability. In *Proceedings of the 1st International Workshop on API Usage and Evolution (WAPI '17)*. IEEE Press, Piscataway, NJ, USA, 2–2. DOI:http://dx.doi.org/10.1109/WAPI.2017..2

[40] Brad A Myers and Jeffrey Stylos. 2016. Improving API usability. *Commun. ACM* 59, 6 (2016), 62–69.

[41] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. 2009. Graph-based Mining of Multiple Object Usage Patterns. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE '09)*. ACM, New York, NY, USA, 383–392.

[42] Raphael Pham, Leif Singer, Olga Liskin, Fernando Figueira Filho, and Kurt Schneider. 2013. Creating a shared understanding of testing culture on a social coding site. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 112–121.

[43] Huilian Sophie Qiu, Yucen Lily Li, Susmita Padala, Anita Sarma, and Bogdan Vasilescu. 2019a. The Signals That Potential Contributors Look for When Choosing Open-Source Projects. *Proc. ACM Hum.-Comput. Interact.* 3, CSCW, Article 122 (Nov. 2019), 29 pages. DOI:http://dx.doi.org/10.1145/3359224

[44] Huilian Sophie Qiu, Alexander Nolte, Anita Brown, Alexander Serebrenik, and Bogdan Vasilescu. 2019b. Going farther together: The impact of social capital on sustained participation in open source. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 688–699.

[45] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *Acm Sigplan Notices*, Vol. 49. ACM, 419–428.

[46] TJ Robertson, Shrinu Prabhakararao, Margaret Burnett, Curtis Cook, Joseph R Ruthruff, Laura Beckwith, and Amit Phalgune. 2004. Impact of interruption style on end-user debugging. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 287–294.

[47] Martin P Robillard. 2009. What makes APIs hard to learn? Answers from developers. *IEEE software* 26, 6 (2009).

[48] Martin P Robillard and Robert Deline. 2011. A field study of API learning obstacles. *Empirical Software Engineering* 16, 6 (2011), 703–732.

[49] Caitlin Sadowski, Kathryn T Stolee, and Sebastian Elbaum. 2015. How developers search for code: a case study. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 191–201.

[50] Simone Stumpf, Vidya Rajaram, Lida Li, Margaret Burnett, Thomas Dietterich, Erin Sullivan, Russell Drummond, and Jonathan Herlocker. 2007. Toward harnessing user feedback for machine learning. In *Proceedings of the 12th international conference on Intelligent user interfaces*. ACM, 82–91.

[51] Jeffrey Stylos and Steven Clarke. 2007. Usability implications of requiring parameters in objects' constructors. In *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 529–539.

[52] Jeffrey Stylos, Benjamin Graf, Daniela K Busse, Carsten Ziegler, Ralf Ehret, and Jan Karstens. 2008. A case study of API redesign for improved usability. In *2008 IEEE Symposium on Visual Languages and Human-Centric Computing*. IEEE, 189–192.

[53] Jeffrey Stylos and Brad Myers. 2007. Mapping the space of API design decisions. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2007)*. IEEE, 50–60.

[54] Jeffrey Stylos and Brad A Myers. 2008. The implications of method placement on API learnability. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. ACM, 105–112.

[55] Christoph Treude and Martin P Robillard. 2016. Augmenting api documentation with insights from stack overflow. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 392–403.

[56] Asher Trockman, Shurui Zhou, Christian Kästner, and Bogdan Vasilescu. 2018. Adding sparkle to social coding: an empirical study of repository badges in the npm ecosystem. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 511–522.

[57] Joe Tullio, Anind K Dey, Jason Chalecki, and James Fogarty. 2007. How it works: a field study of non-technical users interacting with an intelligent system. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 31–40.

[58] Marat Valiev, Bogdan Vasilescu, and James Herbsleb. 2018. Ecosystem-level determinants of sustained activity in open-source projects: A case study of the PyPI ecosystem. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 644–655.

[59] Jue Wang, Yingnong Dang, Hongyu Zhang, Kai Chen, Tao Xie, and Dongmei Zhang. 2013. Mining succinct and high-coverage API usage patterns from source code. In *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, 319–328.

[60] Tianyi Zhang, Ganesha Upadhyaya, Anastasia Reinhardt, Hridesh Rajan, and Miryung Kim. 2018. Are code examples on an online Q&A forum reliable?: a study of API misuse on stack overflow. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 886–896.

[61] Tianyi Zhang, Di Yang, Crista Lopes, and Miryung Kirnt. 2019. Analyzing and supporting adaptation of online code examples. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 316–327.

[62] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. 2009. MAPO: Mining and recommending API usage patterns. In *European Conference on Object-Oriented Programming*. Springer, 318–343.