API Usability at Scale

Andrew Macvean
Google Inc
amacvean@google.com

Luke Church
Google Inc
lukechurch@google.com

John Daughtry
Google Inc
daughtry@google.com

Craig Citro
Google Inc
craigcitro@google.com

Abstract

Designing and maintaining useful and usable APIs remains challenging. At Google, we manage hundreds of externally visible web APIs. Here, we report on our experiences and describe six on-going challenges: resource allocation, empirically-grounded guidelines, communicating issues, supporting API evolution over time, usable auth, and usable client libraries at scale.

Introduction

Application Program Interfaces (APIs) are the way in which programmers access functionality created by others. Every substantive program written in every language today makes extensive use of APIs. It's no surprise that their usability often dominates the experience of programmers.

APIs have been used for decades to connect software modules on a single machine. With the rise of web services and microservice architectures, web APIs have become the primary user interface for many types of services being offered online. While the paper is not centered only on web APIs, we do focus on them (as opposed to standard libraries) a few times. When we do, we are generally referring to representational state transfer (REST) APIs over HTTP. However, the structure of the API (REST vs., say, Simple Object Access Protocol [SOAP]) and the communication protocol (e.g., HTTP vs. local connection) are orthogonal for the discussion.

Despite their importance, there is comparatively little work on understanding API usability. A special interest group was held at the 2009 ACM CHI conference to discuss this problem, the result of which was the founding of apiusability.org (Daughtry, Farooq, Myers, & Stylos, 2009). As of the time of writing, some 50 publications have been listed on this site. A recent paper by Myers and Stylos (2016) provides a summary of the existing work and outlines the various techniques in use for understanding and improving API usability.

In this experience report, we describe some of the major challenges that we have faced building usable APIs at Google and discuss some of our approaches to those challenges. We aim to show the kinds of problems that Google faces as a major API provider and to start a broader conversation as to how we might go about addressing these challenges.

1. Challenge 1: UX Resource Allocation

Traditional usability evaluation techniques can be applied to APIs in order to measure, understand, and improve usability (Clarke, 2004), (Stylos et al., 2008). One of the most cited techniques is running API usability lab studies. This is effective, but not scalable, as participants with professional software experience and researchers with a blend of human-computer interaction, domain and technical knowledge are limited resources (Farooq & Zirkler, 2010). Replacing lab studies with design reviews has been explored by both Microsoft (Farooq & Zirkler, 2010) and Google (Macvean, Maly, & Daughtry, 2016); our experience so far suggests this can augment lab studies, but not replace them.

We are beginning to explore having software engineers run their own studies for APIs, which Google has employed successfully in the past for other domains (Baki et al., 2013). This reduces the need to find qualified researchers but doesn't address the time commitment or the burden of recruiting professional programmers as participants.

Stylos and Myers (2007) described heuristics for how to prioritize research of API design decisions;

Metric	Data for a single API
Opened API Explorer	6,067 sessions
Clicked Execute	1,687 sessions
Experienced 4XX error	47% (795 sessions)
Achieved 2XX response	65% (1,099 sessions)

Table 1 – Explorer usage over one week of data for a single Google API.

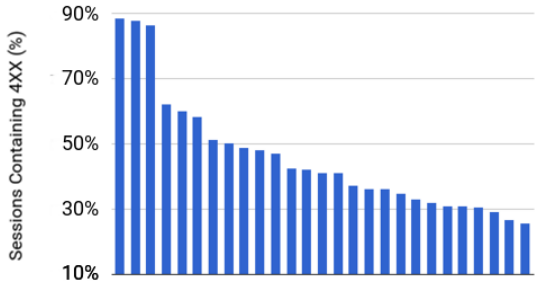

Figure 1 – Percentage of Explorer sessions that encountered a 4XX error across 26 Google APIs.

namely, prioritizing for decisions that developers make often and that severely affect API use. We are using two sources to find the APIs which create the most friction for our users: (1) API exploration usage data, and (2) API Customer Satisfaction surveys.

We can identify invalid API requests by looking for responses with a 4XX status code (i.e. an error in the client request, as opposed to, say, an issue caused by the server of the API. For example, a 404 or 429 HTTP response code), and we can compute the percentage of these responses across APIs. The problem with this approach is that API log data doesn't distinguish between calls that are placed when the developer is learning the API and calls that are run in production, making it difficult to distinguish between a hard-to-use API and existing buggy code. However, nearly all modern web APIs have a 'try it out' feature available online, such as the Google API Explorer[1] or Swagger UI[2]. When we examine log data coming from such a tool, we know that a human is on the other side.

Our hypothesis is that by exposing the Explorer data as usability metrics to the producers of those APIs, we can shine a light on problems and elicit positive change. As a first step, we explored a set of metrics for a small set of APIs. We can, for example, look at a success funnel on a per API basis, see Table 1.

In isolation, these metrics are not actionable. While we want success rate to be higher, what constitutes a good success rate? As a next step, we took a look at the 26 most used APIs in Explorer in a one week period, see Figures 1, 2 and 3.

We now have actionable information. Each graph exposes APIs that are outliers and need significant attention, while also showing us which APIs have further room for improvement. In Figure 1, we see that there are three APIs for whom almost everyone experiences a 4XX error at least once. We now know which APIs to focus on improving and can set a realistic numeric goal based on the less error-prone APIs (e.g., 40%). When we look at Figure 2, we see that many of our APIs average two 4XX errors per session. If you make an error, then the response should include enough information for you to fix the mistake. So, we know that we should look at the APIs in the > 3 range and try to bring them down close to 2. Figure 3 shows us the percentage of sessions that experienced success, where success is defined as achieving a syntactically correct (2XX) request. Again, we see a few APIs that are underperforming and room for improvement on many others.

There are three downsides to these metrics, which must be remembered during interpretation. First,

[1] https://developers.google.com/apis-explorer/
[2] http://swagger.io/swagger-ui/

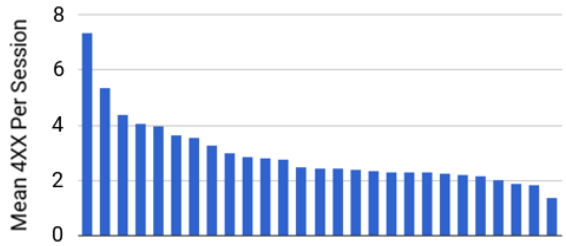

Figure 2 – Mean number of 4XX errors encountered per Explorer session over 26 Google APIs.

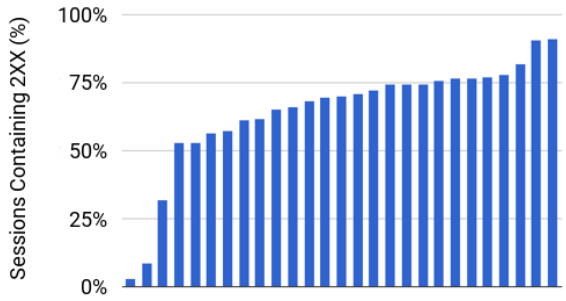

Figure 3 – Percentage of sessions that experienced a 2XX response over 26 Google APIs.

while we know errors are a signal for usability problems, we don't know exactly what usability issue(s) led to the erroneous request. Second, the problems they are experiencing could be usability problems with Explorer itself (the tool) as opposed to API usability issues. Third, we know from other research we have conducted that not every use of Explorer has a goal of achieving a 2XX response; for example, you may be trying to intentionally replicate a bug, or exploring the error messages that occur when you include invalid parameters. In practice, these three problems do not detract from the utility of the method. As long as the metrics guide follow-up evaluation, human investigation can eliminate false-positives while also better understanding the specific usability problems.

While we don't include representative data here, we can also change the granularity of our metrics to expose problems on a per-method basis as opposed to a per-API basis. This can expose potentially confusing methods within a single API as well as providing UX researchers and software engineers more specific targets for lab studies or design reviews.

Our next step is to broaden the scope of the data we include in the analysis to include all of the Google APIs exposed via Explorer and to expose the metrics to API producers. Having access to so many APIs will allow us to set benchmarks that the API producers can compare their API against (e.g., more than one standard deviation above or below the mean). We are also conducting exploratory analysis to establish whether we can reliably distinguish between the different types of usage we see in Explorer. Splitting off the data from those, for example, intentionally getting errors to replicate a bug, would help with the validity of the data, as well as opening up avenues for other interesting analysis.

In parallel to our API log-file analysis, we are also exploring the application of customer satisfaction surveys to our APIs. Using a simple Likert-scale question (Likert, 1932), which asks users to rate their overall satisfaction with an API on a 5-point scale between 'Very dissatisfied' and 'Very satisfied', we can begin to quantify user sentiment towards a particular API. By distributing surveys across our suite of APIs, we also begin to get a picture of how users perceive our services, and, as with the Explorer data, the ways in which one API may stack up against others in order to set benchmarks for improvement. Furthermore, this lightweight approach to surveying allows tracking against time, allowing us to, for example, gauge the impact of a new version of an API with the old.

Surveying API consumers is difficult, given the fact that much of their interaction with the API will

be spent within their IDE or text editor, writing code. We are exploring distribution of our surveys via an unobtrusive intercept survey method (Müller & Sedley, 2014), letting us target API consumers reading documentation. This approach may introduce a selection bias, as using the documentation is not a requirement for using an API, and users viewing the documentation may not have any experience with the API. None the less, when distributed equally and consistently across all of our APIs, we can uncover interesting trends.

Much like with our API Explorer data, comparing sentiment across APIs provides a reference point for improvement. It allows us to show a team managing a given API where their API(s) score in relation to the set of all Google APIs, a set of comparable APIs, or the set of APIs within a given product suite. By exposing outliers, we also know which set of APIs to study to identify successful patterns and which set need to be revised.

Customer satisfaction surveys also overcome one of our problems with the Explorer data; they give us a chance to ask for follow-up qualitative data that helps us interpret the quantitative data. When a respondent indicates that they are not satisfied with the API, we use an open-ended form asking for more details, allowing us to uncover pain points, misconceptions, and feature requests. In aggregate, this exposes cross-cutting issues within a set of APIs. Authentication emerges as a consistent friction point for our users, and is discussed in Section 5.

In this section, we've summarized two approaches we are using to uncover and understand API usability problems. This allows us to direct limited UX resources to more expensive lab studies where they are most needed. Our next step is to see what happens when we expose these metrics to the teams responsible for each API. We hope that providing them with a clearer picture of where they stand with respect to their peers will elicit self-correction and self-management over time. In the next section, we explore methods for discovering these issues before an API is released.

2. Challenge 2: Empirically-Grounded Guidelines

We aim to increase success and satisfaction among developers using our APIs by updating the APIs themselves. Any metrics we can define based on the API surface itself allow us to iterate without expensive lab studies in the process. This approach has been explored by Scheller and Kuhn (2015), who developed a framework for evaluation of an API, identifying measurable factors that impact an API's usability. Similarly, Rama and Kak (2015) established a set of structural issues affecting an API definition and mapped them to a set of measurable metrics. While these methods are grounded in literature, large scale validation remains to be done. Validation often suffers from a lack of empirical data, a problem well understood in the software engineering community (Kaner & Bond, 2004).

We have started a program of work on API analytics, with the goal of empirically validating structural measures of API usability. Our primary goal is to get this information to API producers during the API design and review process, when it's easiest to make changes.

As usability itself is something that is difficult to objectively measure, we chose to use API error rates as a proxy for usability during our preliminary analysis. Thus, the usability of an API is captured as the ratio of client side erroneous requests (4XX) / total requests to the API. In other words, we posit that APIs which result in more client side errors have a poorer usability.

For this analysis, we chose to use API Explorer usage data. This has the obvious shortcoming that users are not fully crafting their own API requests, as they are supported by a GUI based form rather than writing the code themselves. However, this does have the added benefit that we know a human is behind the call, as discussed in the preceding section. We chose to conduct our analysis at an individual method level, rather than, say, evaluating an API as a whole. This is due to the fact that individual methods differ so drastically in structure, that collating across an APIs entire set of methods would likely result in misleading results. For our analysis, we chose the following structural factors:

1. Number of parameters in the request (required and optional).

2. Number of required parameters in the request.
3. Number of different types of parameters (integer, string, float, etc).
4. Number of methods in the overall API.
5. Whether the API required a request body (True|False).
6. HTTP method for the API request (GET, POST, PATCH, etc).
7. Whether the request mutated server state (True|False).

Using a corpus of one year's worth of API Explorer data, we built a linear regression model to assess how much of an API method's 4XX response (client side error) rate could be predicted based on the aforementioned structural factors. We excluded methods which had less than 500 requests in the year, due to the low frequency of traffic. This left us with data on 724 different API methods.

In building our model, we got a coefficient of determination (R^2) score of 0.256. While in some respect this may be interpreted as a poorly fitted model, given the many potential factors that can impact an APIs usability (both structural and otherwise), we are encouraged by the ability to predict as much as 25% of the variance.

While these result are early in nature, and further work (discussed below) is required before we begin to draw strong conclusions, this data leads to two preliminary conclusions. Firstly, objectively measured structural factors do indeed have a measurable impact on the error rate of API requests, a key determinant of an API's usability. This helps to validate both the explorations presented within this paper, and the prior work in the field. Secondly, this technique, coupled with other algorithms such as outlier detection, can focus the attention of API usability practitioners. Those methods which stand apart, both positively or negatively, give cause for further investigation. This can help focus resource allocation, much like the techniques discussed in Section 1.

While these results are positive in nature, we acknowledge that there are limitations to this work. First, we wish to replicate this with our live API logs, thus, evaluating its applicability to 'real' API request data. Additionally, we used a small subset of the possible structural information. We will continue to expand our measures, building upon the literature on API design, software engineering metrics, and our own first hand experience working on API usability. Importantly, future analysis should evaluate the predictive value of each of the individual structural factors, and their role in the overall model. Furthermore, we acknowledge that while API request error rate is an important factor of API usability, it is certainly not the only metric to consider. We wish to extend our analysis to other metrics, including, but not limited to, API customer satisfaction as measured by API surveys, and 'errors on the path to success', a measure of how many erroneous requests an individual makes before achieving a successful call. Finally, linear regression is just one such model we can explore, with its own shortcomings as a way to predict variance. Future work will continue to explore other approaches to machine learning.

3. Challenge 3: Communicating Issues

The techniques above produce a considerable volume of data about API usability. Just as there are substantial resource constraints on the researchers studying the usability of the APIs, the engineers who design and implement the APIs are also busy. Making the information about API usability available to them in a way such that they can easily comprehend and use it is crucial to building better APIs.

Clarke (2004) describes one approach for doing this, using a Cognitive Dimensions profile and showing the difference between the ideal properties of an API and the 'as designed' properties. We have used a similar tool for summarizing dimensional profiles, however we have found that whilst they provide a good synopsis, challenges remain in communication.

Problems with API usability are often made of many small 'papercuts'. Each individual problem by itself may not be too severe, however the overall result over many interactions is that the user makes relatively

little progress. To address this we use friction diagrams, such as Figure 4, which are generated based on observation of a user working with an API in a usability study. These diagrams show the progression of time from left to right. The timelines can be duplicated to highlight specific issues, as they are here. The bottom row within each timeline indicates 'productive working time', the middle row indicates 'experiencing a problem' and the top row indicates 'not making progress or distracted'. For example, in this diagram, we were explaining two issues, one was that there are infrastructure problems with the tooling at that point in time, and the second is that the design pattern in use resulted in developers being repeatedly confused about the handling of state in the API. Whilst each individual interaction wasn't too bad, the overall effect is considerable.

The second technique that we have adopted is to use the outcome of a simplified Cognitive Walkthrough as an explanatory model rather than a predictive one. Figure 5.(a) highlights the simplified version of the Cognitive Walkthrough that we have used for summarizing the interaction that a user might have with an API, derived from (Blackwell, 2008).

This gives us a frame that we can use to explain to engineers the challenges the users are having in interacting with the API. For example, Figure 5.(b) is used to explain a UX problem caused by an apparently missing method in an API (the delete method on the File class). The user has observed the list of available methods, but can't find what they were looking for. Figure 5.(c) shows UX problem caused by the user needing to perform many subgoals into order to achieve the task (to write a line to a file).

These forms of communication have proven reasonably effective at helping engineers understand UX issues, and work on redesigning their APIs. This then leads to another challenge; once engineers have decided how they wish to adjust the APIs, how can they do that in a way that assists users?

4. Challenge 4: Supporting Change

In general it is difficult to change the APIs of a service that is used in the wild. APIs are primarily consumed by computers acting on behalf of their users. Even a small change to the API can result in outages of these dependent systems. APIs, then, have two groups of users: programmers who write the code against APIs, and the computer programs that run this code on behalf of their users. The combination makes safely evolving an API a substantial sociotechnical challenge.

Within Google the source code is largely stored in a single system (Potvin & Levenberg, 2016) which allows every use of an API to be located. Whilst it is still a considerable effort, this makes it possible to determine that all usages of an API have been upgraded before the previous version is disabled.

Externally, things are more challenging, as even APIs that have been marked as obsolete for a number of years show considerable usage. Google's OAuth 1.0 API sent out a deprecation announcement in April of 2012 (Feldman, 2012). The primary use case wasn't turned down until May 2015, and even then many users had not migrated to the new API (Denniss, 2015). The rest of the API was kept alive even longer, and will be shut down in October 2016 (Agarwal & Chun, 2016), over four years after the original deprecation notice. Here again we see a core difference between the defaults for a human and computer – once a process using an API is automated, it will continue using that API, and it requires a developer to spend additional effort to stop it.

The challenge of evolving APIs impacts more than deprecation schedules. At the API special interest group at CHI in 2009, many participants reported that they had seen public APIs withheld from users far longer than they should have been, due solely to the fact that evolving clients is difficult (Daughtry et al., 2009). Thus, designers are encouraged to make the API 'perfect' before release.

The use of typed APIs is a considerable benefit here. For example, a gRPC[3] backed system allows the tooling to warn, or prevent compilation, when an API has been changed in a manner that is incompatible. In some cases it can even provide sufficient information that the tooling can rewrite the old APIs into a

[3] http://www.grpc.io/

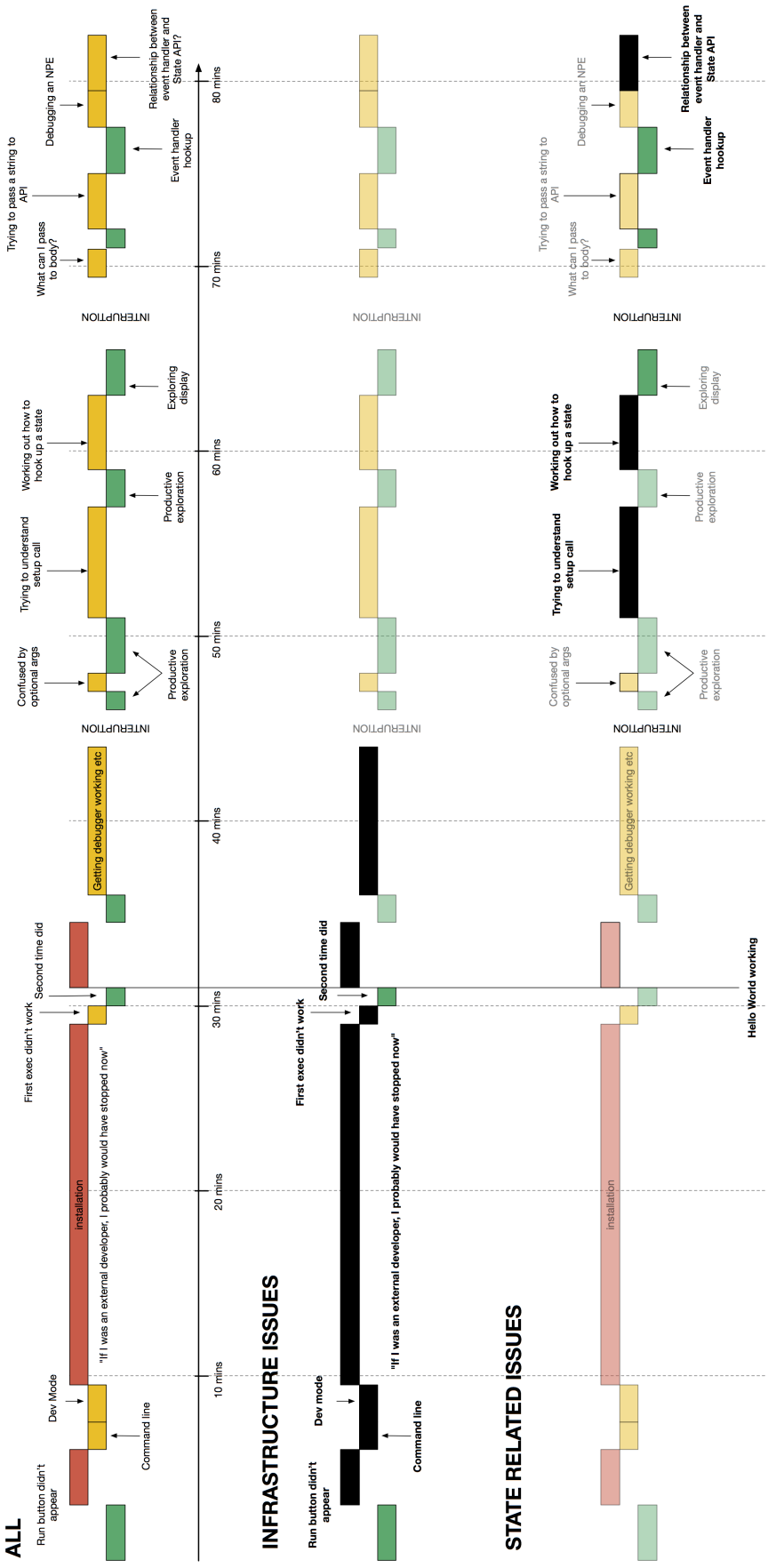

Figure 4 – An example friction diagram, highlighting where and when a user experienced friction during a period of work with a Google API. The bottom row within each timeline indicates 'productive working time', the middle row indicates 'experiencing a problem' and the red row indicates 'not making progress or distracted'.

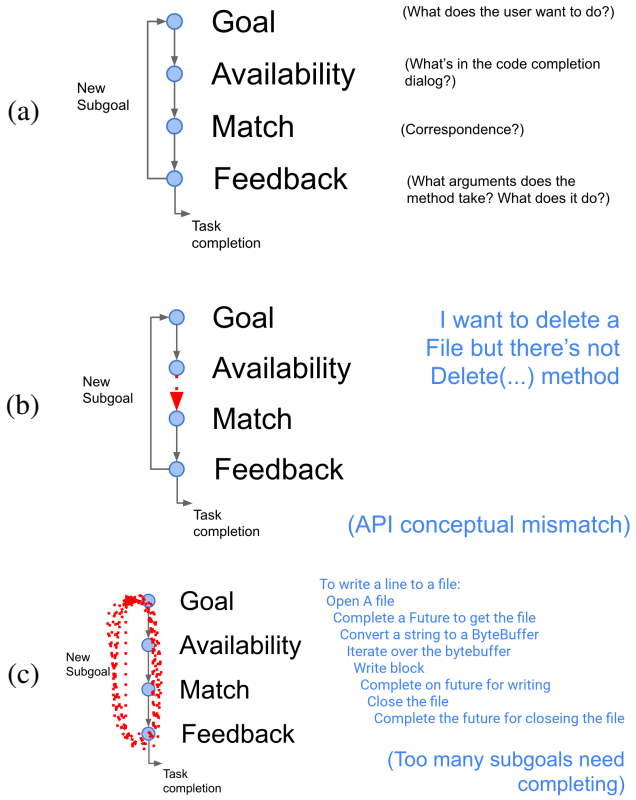

Figure 5 – Explanatory models for effectively communicating API usability issues.

new format, for example Gofix[4].

It may be possible to extend the potential scope of the types of changes that can be addressed this way using the kind of techniques outlined by Church et al. (2016). However, despite these tools, designing systems that support effective dissemination of changes across multiple organizations with legacy code remains a serious challenge and one that limits our ability to act on broad empirical evidence about API usability.

Now we have looked at the general challenges that we have faced in this area, let's look at two specific questions - usable authentication and client library generation.

5. Challenge 5: Usable Auth

Time to Hello World (TTHW) is one metric regularly cited within industry as a key determinant of success for an API. The goal is to reduce friction during the getting started phase such that a new developer can quickly and seamlessly get to a successful Hello World state (or with web APIs, a 2XX response to their request). However, time is only one factor, and one must be careful not to optimize too heavily on a quick 2XX call, at, say, the detriment of long term success. Thus, within Google, we have been advocating that teams instead focus on Pain to Hello World (PTHW) and have been working with teams to understand how they can apply our study design to expose a more holistic picture of the getting started experience.

During task-based API usability studies, we carefully evaluate the user's journeys, tracking the resources used, context switches, misconceptions, difficulties, mistakes made, feature requests, and task time. Highlighted in Figure 6, is one user's unsuccessful attempt to make a 2XX call with a Google API. Each box represents a different context (code editor, documentation, API tooling, etc), and each line represents

[4]https://blog.golang.org/introducing-gofix

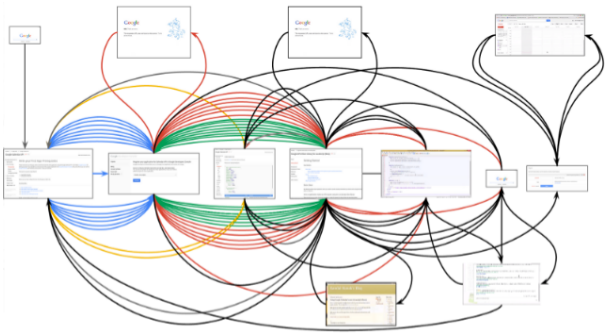

Figure 6 – The journey experienced by one user when trying to get started with a single API.

a context switch. Lines are color coded to represent the phase of the journey. During the red and green phases on the journey, which saw many frequent cycles between documentation and our API management tooling, this user struggled to understand which type of auth (authentication or authorization) was required for the API they were using. Through a number of these evaluations, spanning multiple Google APIs, and approximately 60 participants, we have seen auth emerge as one of the largest painpoints new and experienced developers face alike, somewhat confirming the results of (Biddle, 2014).

To help remediate these problems, we designed and launched a credentials wizard flow within the Google API Console that provides you with the correct credential given the answer to a few questions, namely, which API you are using, in which type of context you intend to use it (e.g. an Android application vs. as web-application), and what type of data your API will be accessing (e.g. user vs. application data). This has helped with the problem, guiding users to the correct auth mechanism while educating them on key decision points which impact auth type. However, it does so by papering over a larger issue; namely, API auth is hard. Even if you manage to get it set up, it is unlikely that you understand how it works, and there is a good chance you are still doing something insecure (e.g., exposing a sensitive credential on GitHub).

When dealing with APIs, you may have to deal with API keys, OAuth clients, and service accounts, with each broad category including sub-items meant to improve security. Some APIs require working with multiple types. The complexity has arisen for good reasons and has helped to improve API security, however it has come at the detriment of developer experience. Most of the attempts to improve API auth usability that we have seen tackle only a subset of the design space. Such local optimization has led to more proliferation of credential types, documentation, restrictions, and failure paths.

We believe that the API authentication design space is ripe for improvement. While there may be more local optimizations that helps, we believe a holistic design would have impact that is orders-of-magnitude greater over time.

6. Challenge 6: Usable Client Libraries at Scale

An attractive feature of web APIs is that they're accessible in any language with an HTTP library – from an AWS Java SDK to a Github client in Emacs Lisp. However, this flexibility comes at a cost: it's hard to design an API that feels equally usable across all potential client languages. In particular, we're faced with several core design decisions: Do we want a hand-written client library for each language and API, or are generated client libraries sufficient? If we hand-write our client libraries, how can we ensure that we're consistent as we move from language to language or API to API? Generating client libraries makes it easy to apply a fix across all APIs, but makes it equally easy to invalidate user code or samples across all those APIs. How do we balance the two?

If developer time were free, hand-writing client libraries for each API and language has significant advantages because we would have control over what was optimized for each language/API pair. In fact, one could even imagine multiple libraries for a given language based on disparate idioms (eg callback-

based and promise-based libraries for Javascript). This also affords the possibility of matching particular API features to language constructs, for example using Python's `with` statement to model database-like transactions.

Providing libraries in multiple languages across hundreds of APIs simply doesn't scale. For instance, Google currently supports client libraries in 10 languages across many of our APIs. The problem isn't just developer time or the number of APIs that Google supports, even a small team building a single API backend can't be expected to understand common idioms, conventions, and packaging across all languages.

While hand-written code is usually idiomatic, it can also be idiosyncratic. Surveys suggest (StackOverflow, 2016) that many developers end up working across at least two languages; having inconsistencies when using two different languages to talk to the same API can be frustrating and error-prone. A common example in this category is date/time-handling libraries, dealing with the inconsistencies of date libraries from language to language is a common pain point for developers. Generating code makes it possible to have consistency along two axes: talking to multiple APIs from a single language, and using the same API from multiple languages.

At Google, we have examples of both auto-generated and hand-written client libraries, and we've performed user studies on PTHW for both. In one example, we compared and contrasted the experience of developers working with the gcloud-node (hand-written) and google-api-nodejs-client (auto-generated) client libraries within the context of two Google Cloud Platform APIs. Note that the hand-written libraries also have documentation and samples written specifically for that library, whereas the auto-generated libraries refer to language-agnostic documentation. In our studies, participants were asked to implement some canonical functionality with either the auto-generated or hand-written library. We then compared and contrasted the experience, looking not only at the success and productivity of the study participants (i.e., how successfully they could complete the study tasks), but also any misunderstandings or errors that occurred during the session, the number of resources (such as documentation) required, and the distribution of time during the session (e.g., time spent reading documentation vs. debugging code, etc).

Although at a small scale, we saw interesting differences in the PTHW journey of the participants. Those working with the hand-written libraries were more successful in completing the development tasks, made less erroneous assumptions about the surface of the API, and required fewer resources in order to successfully complete the tasks.

The contrasting experience highlighted the merits of hand-written libraries, which can be carefully designed to ensure language idiomaticity and nuanced functionality specific to the APIs. In this case, gcloud-node, our hand-written library, is optimized for a specific subset of Google Cloud Platform APIs, while the auto-generated node library can be used with any Google API. We believe that the need for autogeneration of client libraries is clear, but we still have some work to do before the overall developer experience is up to par for that of hand-written libraries.

7. Conclusion

In this work we have summarized issues that we have faced whilst developing and maintaining production APIs. There are more questions and open areas of research than answers, and we've shared some of our approaches to these.

Researchers in industry are in a good position to explore improving designs via usage data. However there are many other concerns that could benefit from the broader research approach taken in academia. Some examples include: what would empirically justified guidelines of API design based on data look like? How might these guidelines be derived from multiple sources of data? How do they relate to existing design patterns? How should these guidelines and results best be presented to developers? How might we longitudinally validate the effectiveness of guidelines? How should change of APIs best be managed and communicated both to developers and to users?

There is clearly much more to be done in this important space.

8. Acknowledgements

We would like to thank Sanjay Ghemawat and Dan Li for their advice and guidance on our research, as well as for providing feedback on the drafts of this paper; and Tao Dong and Harini Sampath who also gave feedback on our drafts.

9. References

Agarwal, V., & Chun, W. (2016). *Saying goodbye to OAuth 1.0 (2lo).* Retrieved 06/02/2016, from https://developers.googleblog.com/2016/04/saying-goodbye-to-oauth-10-2lo.html

Baki, A., Bowen, P., Brekke, B., Ferral-Nunge, E., Kossinets, G., Riegelsberger, J., ... Mayer, M. (2013). Project pokerface: Building a user-centered culture at scale. In *CHI 2013 extended abstracts* (pp. 2325–2326). ACM.

Biddle, R. (2014). Beyond usable security. In *PPIG 2014 - 25th annual workshop.*

Blackwell, A. F. (2008). *Human computer interaction notes.* Retrieved 06/02/2016, from https://www.cl.cam.ac.uk/teaching/0809/HCI/HCI2008.pdf

Church, L., Söderberg, E., Bracha, G., & Tanimoto. (2016). Liveness becomes entelechy - a scheme for l6. In *International conference on live coding.*

Clarke, S. (2004). *Measuring API usability. dr dobb's journal.* Retrieved 06/02/2016, from http://www.drdobbs.com/windows/measuring-api-usability/184405654

Daughtry, J. M., Farooq, U., Myers, B. A., & Stylos, J. (2009). API usability: report on special interest group at CHI. *SIGSOFT Softw. Eng. Notes*, *34*(4), 27–29.

Denniss, W. (2015). *A final farewell to clientlogin, OAuth 1.0 (3lo), AuthSub, and OpenID 2.0.* Retrieved 06/02/2016, from https://developers.googleblog.com/2015/04/a-final-farewell-to-clientlogin-oauth.html

Farooq, U., & Zirkler, D. (2010). API peer reviews: A method for evaluating usability of application programming interfaces. In *CSCW 2010* (pp. 207–210). ACM.

Feldman, A. (2012). *Changes to deprecation policies and API spring cleaning. https://developers.googleblog.com/2012/04/changes-to-deprecation-policies-and-api.html.*

Kaner, C., & Bond, W. P. (2004). Software engineering metrics: What do they measure and how do we know? In *10th international software metrics symposium.*

Likert, R. (1932). A technique for the measurement of attitudes. *Archives of Psychology*, *140*, 1–55.

Macvean, A., Maly, M., & Daughtry, J. (2016). API design reviews at scale. In *CHI 2016 extended abstracts* (pp. 849–858). ACM.

Müller, H., & Sedley, A. (2014). Large-scale in-product measurement of user attitudes & experiences with happiness tracking surveys. In *OzCHI 2014* (pp. 308–315). ACM.

Myers, B. A., & Stylos, J. (2016). Improving API usability. *Commun. ACM*, *59*(6), 62–69.

Potvin, R., & Levenberg, J. (2016). Why google stores billions of lines of code in a single repository. *Commun. ACM*, *59*(7), 78–87.

Rama, G. M., & Kak, A. (2015). Some structural measures of API usability. *Software: Practice and Experience*, *45*(1), 75–110.

Scheller, T., & Kühn, E. (2015). Automated measurement of API usability: The API concepts framework. *Information and Software Technology*, *61*.

StackOverflow. (2016). Retrieved 06/02/2016, from http://stackoverflow.com/research/developer-survey-2016

Stylos, J., Graf, B., Busse, D. K., Ziegler, C., Ehret, R., & Karstens, J. (2008). A case study of API redesign for improved usability. In *IEEE symposium on visual languages and human-centric computing* (pp. 189–192). IEEE.

Stylos, J., & Myers, B. (2007). Mapping the space of API design decisions. In *IEEE symposium on visual languages and human-centric computing* (pp. 50–60). IEEE.