AI-Enhanced API Design: A New Paradigm in Usability and Efficiency

Mak Ahmad*
shahmad@ucdavis.edu
UC Davis
Davis, CA, USA

Andrew Macvean
Google
Seattle, USA
amacvean@google.com

David Karger
MIT
Cambridge, MA, USA
karger@mit.edu

Kwan-Liu Ma
UC Davis
Davis, CA, USA
klma@ucdavis.edu

ABSTRACT

This study uses mixed methods to evaluate API design methods, focusing on design and consumption phases. Our goal was to understand the impact of API governance approaches on productivity and usability. A controlled developer experiment (n=34) demonstrated a 10% increased requirement fulfillment using API Improvement Proposals (AIPs) and linter versus no protocols. Meanwhile, 73% of 33 surveyed API consumers preferred AIP-aligned designs for enhanced usability and comprehensibility. Complementing this, a custom large language model called the API Architect received average expert ratings of just 5/10 for specification quality, revealing gaps versus manual design. The quantitative performance metrics combined with qualitative user feedback provide evidence from multiple angles that strategically integrating industry best practices with maturing AI capabilities can meaningfully improve API design outcomes. This research offers empirical insights from developer and consumer perspectives to advance scholarly discourse and industry practice regarding optimal API design workflows.

CCS CONCEPTS

• **Software and its engineering** → **Software development techniques**; • **Human-centered computing** → **Interactive systems and tools**; • **Computing methodologies** → *Artificial intelligence*.

KEYWORDS

API Design, API Usability, Design Reviews, LLM

ACM Reference Format:
Mak Ahmad, Andrew Macvean, David Karger, and Kwan-Liu Ma. 2024. AI-Enhanced API Design: A New Paradigm in Usability and Efficiency. In *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems (CHI EA '24), May 11–16, 2024, Honolulu, HI, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3613905.3650803

*Corresponding Author

1 INTRODUCTION

Application Programming Interfaces (APIs) are essential for interactions between various systems [15]. Their complex nature [11] has driven the adoption of governance protocols like API Improvement Proposals (AIPs). AIPs comprise a set of guidelines and best practices for API design at our organization. They codify recommendations on areas like API structure, naming conventions, resource modeling, and security. The API linter is an automated static analysis tool that checks API specifications for adherence to the AIP standards, as well as common anti-patterns. It flags issues like inconsistent capitalization of names, missing or incorrect HTTP verbs for routes, and potential security misconfigurations. The linter errors provide actionable feedback for developers to refine their APIs. These protocols aim to strike a balance between flexibility, security, and usability [9]. Despite their widespread implementation, the effectiveness of such standardization methods remains largely unquantified.

The development of effective APIs is challenging [12], necessitating rigorous documentation and strategic version management [11]. Meeting the needs of both providers and consumers is crucial, with usability as a key yet often overlooked factor [21]. Given the complexity involved in building usable APIs, many organization have established governance groups and set design standards [8][18][17][9]. As formalization of API standards becomes more prevalent [10], its critical to truly understand their impact.

This paper conducts a thorough evaluation of API Governance procedures, examining the process of API development and usage with and without the implementation of AIPs and an API Linter. A novel aspect of this research is the utilization of a custom large language model (LLM), named the *API Architect*, enhancing the depth of our evaluation. To benchmark its capabilities, we had industry experts with over 20 years of experience review the Architect's outputs based on criteria like overall quality and completeness. This expert assessment provided a revealing look at the current state of AI in replicating human API design expertise. While the Architect demonstrated promising velocity, producing specifications in minutes, the reviewers identified substantive gaps in technical proficiency and best practice compliance compared to manual approaches. By illuminating the strengths and weaknesses of this emergent generative design tool, we aim to guide future development and integration of AI capabilities in the API lifecycle.

2 RELATED WORK

The landscape of API governance and usability has seen significant advancements and challenges, with a growing emphasis on standardized protocols and tools to enhance API design and development efficiency. A multifaceted API Governance process at Google, involving API Improvement Proposals (AIPs), API Linter, and API Readability programs, has been instrumental in improving API quality through consistency in design rules and processes [2]. This approach reflects an industry-wide trend towards formalized governance to balance the often conflicting needs of API flexibility, security, and usability.

More generally, organizations have explored various techniques for enhancing API Design, such as the Reviews described by Farooq et al [4] and Macvean et al. [9]. These reviews complement traditional API usability tests by focusing on the design rationale and conceptual flaws in APIs, thereby addressing usability issues at an earlier stage in the development process. Unlike API usability tests that yield in-depth feedback from real user interactions, API peer reviews involve a review of the public surface of the API by end-user developers, which offers different insights. Techniques like heuristic evaluations, lab studies, and structural metrics have been employed to quantify API usability, providing a comprehensive understanding of the challenges and best practices in API design [8]. Research by Myers [13] and Piccioni [14] provides key perspectives into API usability by aligning designs with developer cognition. They demonstrate the value of minimizing cognitive load and bolstering intuitive interactions.

Challenges in API usability, as highlighted by Scaffidi [18] and [16], include inadequate documentation, insufficient orthogonality, and mismatches between API abstractions and application requirements. These challenges emphasize the need for meticulous documentation and thoughtful versioning strategies in API design. Research has also focused on the evolution of APIs, with studies by Church et al. [3] and Agarwal and Chun [1] examining version resilience and deprecation timeframes, respectively. The works of Grill, Scheller, and Rama offer vital insights into API usability and governance. Grill et al. emphasize the importance of structured API governance in software development, highlighting its role in enhancing product quality and mitigating risks in fast-paced development environments [6]. Scheller focuses on improving the developer experience through intuitive API design, enhancing productivity and satisfaction [19]. Collectively, their research provides a multifaceted view on optimizing the API lifecycle for developers and end-users alike. However, the efficacy of such standardization techniques has not been comprehensively quantified. This research aims to shed light by exploring API design outcomes with and without AIPs. Additionally, we highlight the benefits and potential drawbacks of integrating advanced AI capabilities in this field. Specifically, we investigate the following questions:

- **RQ1:** How do the implementation of API Improvement Proposals (AIPs) and the utilization of tools like the API Linter and AI-based API designers influence the overall efficiency and quality of API production? This question seeks to understand the extent to which these tools and protocols impact the speed, accuracy, and satisfaction in the API development process.
- **RQ2:** In what ways does the adoption of AIPs and AI-based tools affect the user experience of both API producers and consumers? This includes examining the measurable impacts on development velocity, requirement fulfillment, and the overall user experience in interacting with APIs.

3 METHODOLOGY

The methodology involved analyzing API production and consumption among developers and evaluating AI-generated specifications. For the first study, engineers were tasked with designing APIs under varying conditions with and without AIPs and linter. To supplement human expert evaluation, an AI assistant termed the *API Architect* was developed using a custom GPT-4 on OpenAI. The GPT-4 model ingested labeled data covering all AIP specifications to allow it to generate API designs informed by industry best practices. After fine-tuning, the *API Architect* could receive API requirements just like a human designer, and respond by generating full API specifications along with additional analysis. We then had 6 industry experts with over 20 years of API design experience evaluate the Architect's outputs based on criteria including overall quality, evolvability, completeness and AIP adherence. This two-pronged approach enabled a comprehensive empirical examination of both human and AI-based API design processes and outcomes.

Our research methodology comprised three core evaluation components. First a controlled developer experiment comparing API design processes and outcomes across groups. Second, a qualitative interviews assessing API consumer preferences through side-by-side specification comparisons. Lastly, an expert review of AI-generated API specs using structured ratings on scales from 0 to 10. The controlled experiment and consumer interviews involved direct comparisons between options. However, for expert assessment of the AI system, numerical ratings enabled a more nuanced evaluation of multiple facets like completeness and evolvability. This multifaceted approach enabled comprehensive benchmarking of both human and AI-based API design capabilities from the perspectives of producers, consumers, and experts.

3.1 API Production Experience

A survey was sent to 4,000 software engineers at a Google API interest group. The group is a mailing list consisting of members interested in any aspect of APIs (e.g., API Usability, API Security, Designing APIs and Frameworks). In the email, we requested responses from only those with at least 1 year of experience designing API specs. In total, we received 234 responses to the survey. Among the 234, 100 agreed to be part of this study, where they were tasked to design APIs based on sample product specifications that we provided. The goal of the product was to build a Social Network App for kids to share ideas and information with their friends. Requirements included things such as "Parent creates an account and child account" or "Feed displays relevant posts based on child's followers and interactions (likes and comments). Participants were asked to spend 2 hours recording their screen while designing the API. After completing the design, they were also required to submit a survey to reflect on the experience. For their time, participants were rewarded $80. We balanced the 100 participants into 3 groups using the Gower distance [5]:

- **Group I**: Must use AIPs and the API Linter
- **Group II**: Must use AIPs but not the API Linter
- **Group III**: Not allowed to use AIPs nor the API Linter but can use any other resources

The Gower Distance To create balanced experimental groups, we leveraged advanced statistical techniques including Gower Distance [5] for measuring dissimilarities and Partitioning Around Medoids (PAM) [20] for clustering. This allowed us to split the 100 participants into 3 groups with minimal differences in key attributes like experience level. We further employed propensity score matching, assessing distribution of categorical variables and numerical means, to mitigate selection bias. This rigorous approach enabled comparable groups, enhancing experimental validity. Out of the 100 participants, we received responses from 34, which we divided into three groups: 11 in **Group I**, 11 in **Group II**, and 12 in **Group III**.

3.2 API Consumption Experience

We recruited 33 professionals with an average of 12 years of software engineering experience through LinkedIn and API discussion channels. Approximately 71% were familiar with AIPs, and 56% had prior experience using AIPs for API design. The API specifications evaluated in the interviews were randomly selected from those developed by engineers in the API production phase of our study. Each 30-minute video interview consisted of both open-ended questions and a structured survey. After gauging participants' AIP familiarity and design criteria, we administered surveys with 5 side-by-side API specification comparisons randomly drawn from the 3 pipeline groups.

To enable comprehensive comparisons, we manually matched elements across the groups' API specs. For example, when comparing "create account" flows, we systematically paired the Group 1 design against relevant sections in Group 2 and Group 3. This matching process ensured an unbiased evaluation of all specifications. The process included 4 steps. `Survey Creation`: Three distinct surveys were developed, each with five questions to enable random pairwise comparisons between API specifications across the groups. `API Documentation Generation`: Using buf.build, we generated detailed API documentation for each specification, forming the basis of our analysis and comparison. For further details, if needed, see the screenshot/figure 1 included in the Appendix. `Code Snippet Creation`: Alongside the documentation, we produced code snippets illustrating the APIs' functionality, providing a practical context for survey comparisons. For further details, if needed, see the screenshot/figure 2 included in the Appendix. `Pairwise Analysis`: During our surveys, participants undertook informed pairwise comparisons of API specifications across three distinct buckets. This approach ensured a thorough comparison, with each API element from one bucket being systematically paired against corresponding elements from the other two buckets. For instance, in scenarios like "create an account", "send a post", and "retrieve a list of posts with paging", participants selected their preferred API version from two choices presented in each scenario. This structured approach allowed us to draw meaningful conclusions regarding the strengths and weaknesses of the APIs, ultimately contributing to a more comprehensive understanding of their respective documentation and usability.

The *API Architect* evaluated the API designs from the three pipeline groups using several criteria. These included a **Usability score**, a quantitative assessment of the API specification's usability on a scale from 0 to 100; an **Evolvability score**, measuring the ease of modifying or expanding the API specification in the future, also on a 0 to 100 scale; and an **AIP adherence score**, quantifying the extent to which the specification follows AIP conventions, again rated between 0 and 100.

4 RESULTS

4.1 API Production Experience

Comparing the perception of engineers with the actual design assessment, 94% of the engineers believed they completed more than 32% of the requirements specification in the 2-hour period. However, the analysis shows just about 58% of the engineers actually did so. 44% believed we could have allotted more than 2 hours period for the API design.

A majority of engineers (65%) reported that the API design process felt easy. Of this group, 29% used both AIPs and linter, 38% used just AIPs, and 33% used neither protocols. For the 35% of engineers who found API design difficult, first-time AIP users in particular noted challenges toggling between documentation and coding environments.

Despite these learning curves, for those leveraging AIPs, 91% agreed that AIPs were helpful for design. Furthermore, among those also using the linter tool, 75% found it beneficial when combined with AIPs. While adapting to new protocols poses initial difficulties, especially for novice users, a significant majority of engineers found value in AIPs for enhancing API design. Integrating linter alongside AIPs provided additional gains in perceived productivity.

To evaluate completion rate and errors, each API design response was manually checked against the 29 product requirements. Each requirement was scored as either 1 (satisfied) or 0 (unsatisfied). The percentage of requirements satisfied determined the completion rate for each response. Additionally, the number of errors flagged by the API Linter was recorded where applicable. The analysis revealed the following completion rates and error counts across groups:

- **Group I** (API Linter and AIPs): Average completion rate of 71.11%. Average of 13.9 linter errors.
- **Group II** (AIPs but No API Linter): Average completion rate of 62.22%. Average of 19.2 linter errors.
- **Group III** (No API Linter and No AIPs): Average completion rate of 62.92%. Average of 21.5 linter errors.

To investigate the practical significance of the observed differences in completion rates among the three groups, we computed Cohen's *d*, a widely used measure of effect size. In addition to calculating effect sizes, we examined the precision and certainty of our estimates using confidence intervals and standard errors. The effect size analysis revealed moderate effect sizes of 0.4 and 0.6 respectively when Group I is compared with Groups II and III, while the comparison between Groups II and III revealed a small effect size. Standard errors are relatively small. However, the wide confidence intervals indicate that our estimates are associated with considerable uncertainty due to the small sample size.

This results of the API production experience provided insights into the impacts of governance protocols. Despite initial learning curves, especially for new AIP users, a significant majority found AIPs valuable for enhancing design. The controlled experiment also quantified productivity gains, with the group leveraging both AIPs and linter achieving a 10 percentage point higher completion rate compared to the other groups.

4.2 API Consumption Experience

Our interviews revealed that most participants (56%) had less than 7 years of experience in the industry, and a significant portion (44%) had more than 7 years of experience, suggesting a diverse range of experience levels among API consumers. There were two main groups of API consumers, based on their evaluation criteria. The first group (33%) prioritized technical proficiency and dexterity, and they valued APIs that were readable and easy to use. The second group (30%) prioritized broader business goals and adaptability, and valued APIs that were agile and end-user-acceptable. Only 4% of API consumers prioritized security-related aspects. The survey found that 73% of the total participants preferred API designs that incorporated AIPs, and 76% of that percentage preferred designs that included the use of both AIPs and API Linter which contributed to the improved structure, standardization, and documentation of the APIs. However, 38% expressed a preference for alternative designs without AIPs or linter, citing completeness as a concern.

When comparing Group I and Group III API specifications, out of 120 pairwise matchups, Group I was preferred 73 times compared to 47 times for Group III. A two-proportion z-test showed this distribution has a z-score of 3.156 and a p-value of 0.008, indicating a statistically significant preference for the Group I designs at the $p < 0.05$ level. However, the comparison between Group II and Group III did not reveal a significant difference in preferences. Out of 150 matchups between these groups, Group II was preferred 79 times versus 71 times for Group III. The z-test resulted in a z-score of 1.05 and a p-value of 0.147, meaning no substantial preference difference between Group II and III. Users exhibited a definitive preference for Group I APIs over Group III, but Group II vs. Group III did not show a statistically significant distinction. This suggests integration of both AIPs and linter notably enhances API usability over no protocols, while AIPs alone do not confer a similarly sizable advantage.

On average, the *API Architect* scored the Group I specifications highest across all criteria (usability = 74.4, evolvability = 69.6, AIP adherence = 73.6). This aligns with the 73% rate of human experts preferring Group I designs, demonstrating strong agreement between human evaluator preferences and the Architect assessments. To test if Architect-measured differences between pipelines were statistically significant, two-proportion z-tests were conducted. Comparing Group I and Group III AIP adherence scores (where p_1 = mean Group 1 score = 73.6, p_2 = mean Group 3 score = 72.8) yielded $p = 0.04818 < 0.05$.

The respondents with less than 7 years preferred APIs that were "readable" and "easy to use." These considerations underscored their emphasis on technical proficiency and dexterity, as they expressed their preference for APIs that are not only functional but also comprehensible. Specifically, phrases such as *"Readability: I want to understand what it's doing if I read the code in order"* encapsulated their criteria for assessing API quality. Conversely, respondents with more than 7 years of experience leaned toward facets such as "end-user acceptability" and "agility" when scrutinizing APIs. These preferences reflected their wealth of leadership and product-oriented experience, demonstrating a nuanced perspective that prioritizes broader business goals and adaptability. Phrases like *"Does the API deliver business value?" and "Does it achieve a goal? How easy does it solve the problem?"* captured the essence of their evaluative framework.

4.3 API Architect Evaluation

To benchmark the human-generated API designs, the AI-based *API Architect* developed specifications for the same product requirements within 3 minutes. Six industry experts with an average of 20 years of API design experience evaluated the Architect's APIs based on overall quality, evolvability, completeness, and AIP adherence. The average overall quality score was 5 out of 10, highlighting some deficiencies compared to human-level design proficiency. The evolvability scored highest at 7 out of 10, as the Architect's systematic approach enables future extensibility. However, completeness was rated lower at 5.8 out of 10, indicating key functionality gaps in the generated designs. Finally, AIP adherence received a 4.5 out of 10 score, reflecting deviations from best practice conventions.

Qualitative feedback emphasized issues like the API Architect's divergence from resource-oriented principles and lack of HTTP bindings. As one expert noted, "It doesn't seem resource-oriented at all...no definition of User or Post, just duplicate fields in different requests/responses." Another stated the designs don't follow "the most common AIPs (Pagination, field names on resources, etc)." Several responses highlighted missing functionality, with one commenting "It doesn't look like the parental controls are there at all." These example quotes align with the quantitative analysis pointing to limited completeness and AIP compliance.

5 DISCUSSION

5.1 API Production Experience

The analysis of API production experience provides valuable insights into RQ1 on how AIPs and tools like the API Linter influence development velocity and requirement fulfillment. The completion rate data reveals that Group I, leveraging both AIPs and linter, achieved the highest rate of 71.11%, surpassing Groups II and III by nearly 10%. This aligns with RQ1 by quantitatively showing how comprehensive integration of structured protocols boosts productivity.

Furthermore, the effect size analysis indicates a moderate effect when contrasting Group I to Groups II and III, demonstrating the notable influence of the AIPs and linter tandem on productivity. Given the substantial Cohen's d values, these tools tangibly improve engineers' capacity to build functional, production-ready APIs within constrained timeframes. While the confidence intervals are wide owing to small sample sizes, the consistently higher completion rates and API quality (lower error counts) underscore the benefits of AIP adoption. However, the perceived difficulty variance across groups highlights an avenue for refinement. The additional rigor imposed by strictly adhering to AIPs and resolving linter errors was seen as challenging, especially by first-time users. This indicates that more work is required towards tightening the integration within existing developer workflows. As API design practices evolve across organizations, maintaining an emphasis on minimizing disruption and flattening learning curves will be key. Nonetheless, our findings strongly validate that leveraging AIPs and linter collectively enhances the API development experience along critical factors of velocity and requirement fulfillment. In this context, the importance of effective API governance systems, as discussed comprehensively by Krintz et al. [7], becomes evident, emphasizing the critical role of APIs in digital IT infrastructure and the need for structured management and governance systems.

5.2 API Architect

Participants typically invested 2-3 hours to design APIs, while the *API Architect* leveraged GPT-4 to generate specifications in just minutes. However, expert assessment revealed the Architect's outputs were rated average 5 out of 10 overall, highlighting deficiencies compared to manual approaches. While the *API Architect* demonstrated promising velocity, its technical proficiency and best practice compliance requires substantial improvement to match human designers. This tempers notions of AI achieving comparable API design quality and prompts further evaluation of how to best integrate AI assistance without compromising end results. The time savings warrant exploration of AI acceleration of rote tasks, provided the overall workflow incorporates sufficient human oversight.

The expert assessment of the *API Architect* provides valuable insights into the current capabilities of AI for API design. While the Architect demonstrated promising velocity by drafting specifications in minutes, the modest scores it received highlight meaningful gaps compared to human expertise. With average ratings between 4.5 and 7 out of 10 on overall quality, completeness, and AIP adherence, the results reveal clear room for improvement. The greatest strength lies in the Architect's systematic approach, reflected in the higher evolvability score. However, reviewers cited critical weaknesses in resource modeling, protocol compliance, and missing functionality. This tempers RQ1 by indicating substantial room for improving technical proficiency of generative AI tools. However, the Architect shows potential for accelerating rote specification generation to augment designers through expert fine-tuning.

5.3 API Consumption Experience

The API consumption research analyses provide compelling insights into RQ2 regarding the impact of AIPs and linter on the user experience. Across all participants, including those possessing prior AIP proficiency, the study revealed a resounding 73% preference towards Group I designs harnessing AIPs and linter in tandem. This directly addresses the core research question regarding measurable improvements in efficiency and user satisfaction.

The statistically significant edge of Group I over Group III validates that Purposeful API design standards substantively enhance comprehension and utility for consumers. Moreover, the alignment between human evaluator preferences and *API Architect* scoring demonstrates the objectively higher quality of Group I specifications. Notably, while familiarity with AIPs contributes to an informed perspective, it does not universally translate to favorability, as 38% of participants opted for alternatives lacking AIPs or linter. Additionally, differentiation emerged across experience levels, with less seasoned engineers prioritizing readable and usable designs, while more seasoned ones focused on business value and problem-solving efficacy. This bifurcation spotlights the need for versatility in API design thinking, catering to multifaceted consumer viewpoints.

The consumption analysis provides unambiguous quantitative and qualitative proof that integrating AIPs and linter into API design workflows yields superior outcomes for end users. The areas warranting further attention involve enhancing intuitive integration for producers and accommodating the diversity of consumer preferences. Overall, the research strongly confirms that adherence to API development best practices translates to substantially improved experience for API consumers.

6 LIMITATIONS AND FUTURE DIRECTIONS

While this research provides compelling insights, some limitations warrant acknowledgement. The production experience study's small sample size constrains generalizability of the performance gains measured. Additionally, the limited API consumption interview sample merits expanded investigation. The *API Architect* evaluation also faced constraints, including a small expert reviewer pool and reliance solely on specifications rather than implementations. More objective rubrics could improve precision given variance in scores. Nonetheless, the quantitative and qualitative techniques established convincing evidence of gains from standardized API design. The limitations present clear pathways for enhancing the research's rigor and real-world value through three primary directions. First, larger participant samples in both the production and consumption studies, coupled with longitudinal tracking of API iterations, can strengthen insights. Second, iterative expert refinement of AI tools like the Architect through cycles of feedback is a promising direction. Finally, end-to-end studies monitoring the Architect from design through deployment could reveal new insights. As API governance matures, purposefully benchmarking human creativity against structured protocols and generative AI will shape the next evolution of API design innovation. While continuous enhancement is warranted, this research affirms integrating industry best practices with maturing AI can significantly enhance API development outcomes.

7 CONCLUSION

This study offers strong evidence for enhancing API development with standardized protocols and AI. Results show productivity gains and a 10% higher completion rate using AIPs and linters. Additionally, 73% of API consumers favored AIP and linter-developed specifications, highlighting their benefits. Although AI tools like the *API Architect* need refinement for full automation, their potential in routine tasks is evident. The research suggests a collaborative approach, integrating human creativity, structured protocols, and AI, advancing API design and benefiting creators and consumers alike.

ACKNOWLEDGMENTS

Thank you Richard Frankel, JJ Geewax, Alfred Fuller, Mike Kistler, Dan Hudlow, Yusuke Tsutsumi, Joaquin Carretero and the many Google engineers for your assistance in this research.

REFERENCES

[1] Vartika Agarwal and Wesley Chun. [n. d.]. Saying goodbye to oauth 1.0 (2LO). https://developers.googleblog.com/2016/04/saying-goodbye-to-oauth-10-2lo.html

[2] Anonymous. [n. d.].

[3] Luke Church, Emma Söderberg, Gilad Bracha, and Steven Tanimoto. 2016. Liveness becomes Entelechy-a scheme for L6. In *The second international conference on live coding*.

[4] 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*. 2327–2336.

[5] John C Gower. 1971. A general coefficient of similarity and some of its properties. *Biometrics* (1971), 857–871.

[6] Thomas Grill, Ondrej Polacek, and Manfred Tscheligi. 2012. Methods towards API usability: A structural analysis of usability problem categories. In *Human-Centered Software Engineering: 4th International Conference, HCSE 2012, Toulouse, France, October 29-31, 2012. Proceedings 4*. Springer, 164–180.

[7] Chandra Krintz, Hiranya Jayathilaka, Stratos Dimopoulos, Alexander Pucher, Rich Wolski, and Tevfik Bultan. 2013. Developing systems for API governance. *figshare* (2013), 790746.

[8] Andrew Macvean, Luke Church, John Daughtry, and Craig Citro. 2016. API Usability at Scale.. In *PPIG*. 26.

[9] Andrew Macvean, Martin Maly, and John Daughtry. 2016. API Design Reviews at Scale. In *CHI EA '16 Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems*. 849–858. http://dl.acm.org/ft_gateway.cfm?id=2851602&ftid=1716532&dwn=1&CFID=783074184&CFTOKEN=67606185

[10] Lauren Murphy, Tosin Alliyu, Andrew Macvean, Mary Beth Kery, and Brad A Myers. 2017. Preliminary analysis of REST API style guidelines. *Ann Arbor* 1001 (2017), 48109.

[11] 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)*. 249–258. https://doi.org/10.1109/VLHCC.2018.8506523

[12] 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.

[13] Myers and Stylos. 2016. Improving API usability. 59 (May 2016). https://doi.org/10.1145/2896587

[14] Marco Piccioni, Carlo A. Furia, and Bertrand Meyer. 2013. An Empirical Study of API Usability. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*. 5–14. https://doi.org/10.1109/ESEM.2013.14

[15] Yuanbo Qiu. 2016. The openness of Open Application Programming Interfaces. *Taylor & Francis* (2016), 1720 – 1722. http://doi.acm.org/10.1145/1721695.1721705

[16] Girish Maskeri Rama and Avinash Kak. 2015. Some structural measures of API usability. *Software: Practice and Experience* 45, 1 (2015), 75–110.

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

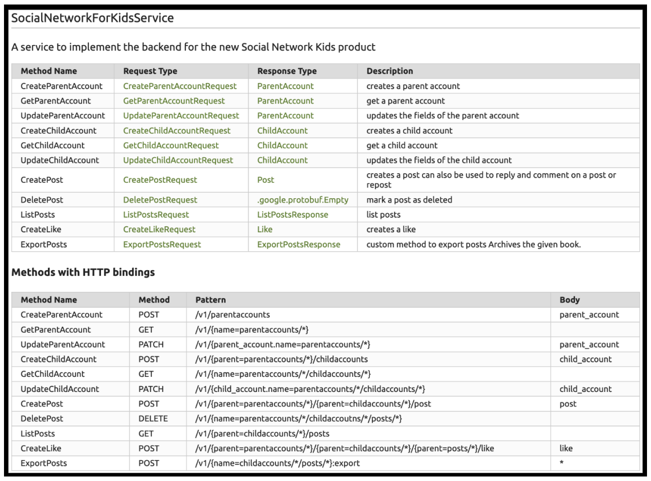
[18] Christopher Scaffidi. 2006. Why are APIs difficult to learn and use? *XRDS: Crossroads, The ACM Magazine for Students* 12, 4 (2006), 4–4.

[19] Thomas Scheller and Eva Kühn. 2015. Automated measurement of API usability: The API concepts framework. *Information and Software Technology* 61 (2015), 145–162.

[20] Mark Van der Laan, Katherine Pollard, and Jennifer Bryan. 2003. A new partitioning around medoids algorithm. *Journal of Statistical Computation and Simulation* 73, 8 (2003), 575–584.

[21] Minhaz F. Zibran, Farjana Z. Eishita, and Chanchal K. Roy. 2011. Useful, But Usable? Factors Affecting the Usability of APIs. In *2011 18th Working Conference on Reverse Engineering*. 151–155. https://doi.org/10.1109/WCRE.2011.26

A RELEVANT SCREENSHOTS



SocialNetworkForKidsService

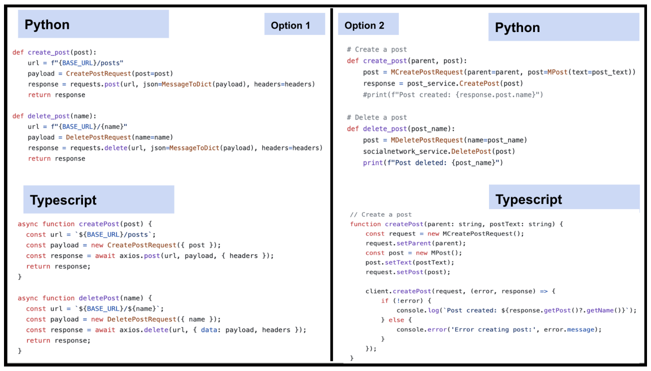
A service to implement the backend for the new Social Network Kids product

Method Name	Request Type	Response Type	Description
CreateParentAccount	CreateParentAccountRequest	ParentAccount	creates a parent account
GetParentAccount	GetParentAccountRequest	ParentAccount	get a parent account
UpdateParentAccount	UpdateParentAccountRequest	ParentAccount	updates the fields of the parent account
CreateChildAccount	CreateChildAccountRequest	ChildAccount	creates a child account
GetChildAccount	GetChildAccountRequest	ChildAccount	get a child account
UpdateChildAccount	UpdateChildAccountRequest	ChildAccount	updates the fields of the child account
CreatePost	CreatePostRequest	Post	creates a post can also be used to reply and comment on a post or repost
DeletePost	DeletePostRequest	.google.protobuf.Empty	mark a post as deleted
ListPosts	ListPostsRequest	ListPostsResponse	list posts
CreateLike	CreateLikeRequest	Like	creates a like
ExportPosts	ExportPostsRequest	ExportPostsResponse	custom method to export posts Archives the given book.

Methods with HTTP bindings

Method Name	Method	Pattern	Body
CreateParentAccount	POST	/v1/parentaccounts	parent_account
GetParentAccount	GET	/v1/{name=parentaccounts/*}	
UpdateParentAccount	PATCH	/v1/{parent_account.name=parentaccounts/*}	parent_account
CreateChildAccount	POST	/v1/{parent=parentaccounts/*}/childaccounts	child_account
GetChildAccount	GET	/v1/{name=parentaccounts/*/childaccounts/*}	
UpdateChildAccount	PATCH	/v1/{child_account.name=parentaccounts/*/childaccounts/*}	child_account
CreatePost	POST	/v1/{parent=parentaccounts/*}/{parent=childaccounts/*}/post	post
DeletePost	DELETE	/v1/{name=parentaccounts/*/childaccoutns/*/posts/*}	
ListPosts	GET	/v1/{parent=childaccounts/*}/posts	
CreateLike	POST	/v1/{parent=parentaccounts/*}/{parent=childaccounts/*}/{parent=posts/*}/like	like
ExportPosts	POST	/v1/{name=childaccounts/*/posts/*}:export	*

Figure 1: API documentation automatically generated using buf.



Python

```python
def create_post(post):
    url = f"{BASE_URL}/posts"
    payload = CreatePostRequest(post=post)
    response = requests.post(url, json=MessageToDict(payload), headers=headers)
    return response

def delete_post(name):
    url = f"{BASE_URL}/{name}"
    payload = DeletePostRequest(name=name)
    response = requests.delete(url, json=MessageToDict(payload), headers=headers)
    return response
```

Typescript

```typescript
async function createPost(post) {
  const url = `${BASE_URL}/posts`;
  const payload = new CreatePostRequest({ post });
  const response = await axios.post(url, payload, { headers });
  return response;
}

async function deletePost(name) {
  const url = `${BASE_URL}/${name}`;
  const payload = new DeletePostRequest({ name });
  const response = await axios.delete(url, { data: payload, headers });
  return response;
}
```

Python

```python
# Create a post
def create_post(parent, post):
    post = MCreatePostRequest(parent=parent, post=MPost(text=post_text))
    response = post_service.CreatePost(post)
    #print(f"Post created: {response.post.name}")

# Delete a post
def delete_post(post_name):
    post = MDeletePostRequest(name=post_name)
    socialnetwork_service.DeletePost(post)
    print(f"Post deleted: {post_name}")
```

Typescript

```typescript
// Create a post
function createPost(parent: string, postText: string) {
    const request = new MCreatePostRequest();
    request.setParent(parent);
    const post = new MPost();
    post.setText(postText);
    request.setPost(post);

    client.createPost(request, (error, response) => {
        if (!error) {
            console.log(`Post created: ${response.getPost()?.getName()}`);
        } else {
            console.error('Error creating post:', error.message);
        }
    });
}
```

Figure 2: API code snippet manually created based on the API specification.