User-Centered Intelligent Information Support for Programmers

Daye Nam

CMU-S3D-24-101

May 2024

Software and Societal Systems Department
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213

Thesis Committee:
Bogdan Vasilescu, Co-Chair
Brad Myers, Co-Chair
Vincent Hellendoorn, Co-Chair
Baishakhi Ray, (Columbia University)
Andrew Macvean, (Google, Inc.)

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Software Engineering.

This work was supported by the National Science Foundation, under NSF grant CCF-2007482, and by Google. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author and do not necessarily reflect those of the National Science Foundation or Google.

Keywords: Information Seeking, Program Understanding, API learning, Developer Tools, Human-Computer Interaction, Software Engineering

To my loving family,
for unwavering support and belief in me.

Abstract

Software engineering is an information-intensive discipline. While building and maintaining software systems, programmers encounter a broad spectrum of questions ranging from implementation specifics to architectural concerns. However, satisfying their information needs is not easy, because the relevant information is often scattered across varying mediums in different formats. It becomes even more challenging when a programmer needs to work with unfamiliar code or libraries, without the necessary knowledge and experience to search for information effectively.

In this thesis, I aim to address the challenges programmers face in seeking information by designing, building, and evaluating intelligent information support tools using user-centered approaches. To provide user-centered intelligent support, this thesis investigates both the users (i.e., programmers), and the intelligent techniques. To motivate the need for user-centered information support, I first present an exploratory, mixed-methods empirical study on documentation page-view logs, revealing discernible documentation page visit patterns. The study shows that programmers use documentation differently, and their contextual factors correlate with their documentation usage, highlighting the need for information support tailored to diverse user contexts instead of "one size fits all" solutions. I then introduce four prototype intelligent information support tools designed to assist developers working with unfamiliar code, concepts, application domains, and APIs. Evaluations of these tools demonstrate the effectiveness of intelligent solutions, compared with traditional baseline approaches. Furthermore, user studies conducted with these prototypes illustrate the benefits of designing information support tools that account for users' current tasks and the broader context in which these tasks are situated. Motivated by these findings, this thesis finally explores the possibility of personalizing information support for programmers, by leveraging the programmers' contextual factors (e.g., familiarity with application domains) to enhance information support.

By investigating various intelligent techniques for different information-seeking scenarios, this thesis illustrates how a thorough understanding of the users not only yields valuable insights for designing more useful and usable tools but also improves the performance of intelligent techniques used in these tasks. The methodologies used for building user-centered information support tools, along with the insights gained from user studies in this thesis, will further our understanding of how developers with varying goals and backgrounds seek and use information. Taken together, the thesis will shed light on the design of future programming tools, especially those that will be built for a new programming paradigm that relies heavily on AI-based support.

Acknowledgments

The past six years have been a profound journey, marked by a series of ups and downs that, while vivid in my memory, also seem to have passed in the blink of an eye. Although I often still feel like a wide-eyed kid, amazed by the rapid changes unfolding in the world, I have also grown significantly since 2018 when I first started my PhD. Reflecting on this period, I believe the PhD program was designed to challenge me thoroughly, both academically and personally, yet it provided a nurturing environment surrounded by wonderful people who earnestly supported my success.

First, I owe immense gratitude to my three co-advisors: Bogdan Vasilescu, Brad Myers, and Vincent Hellendoorn. I am profoundly grateful for their mentorship which was not just academic but often personal, helping me navigate the complex landscape of research and self-discovery. Their diverse expertise fueled my interdisciplinary journey, shaping my aspirations and molding me into the researcher I am today. What I received from my advisors extends beyond the confines of a single paragraph, but I believe the most crucial factor was their belief in my potential, even when I doubted myself. Although I will miss the four-person weekly meeting a lot, I also look forward to all the collaborations we can do in the future.

My appreciation extends to my thesis committee members, Baishakhi Ray and Andrew Macvean. Their insightful advice, probing questions, and thoughtful comments have significantly enriched my thesis work. Their mentorship throughout my thesis work and the collaboration has also shaped my resilience and perspective, preparing me for challenges both within and outside the academic world.

I have also been privileged to receive mentorship from many senior researchers at Carnegie Mellon University and within the Software Engineering Community. I cherish my PhD experience, largely due to engaging discussions and learning opportunities from senior researchers. Special thanks to Michael Hilton for shaping my teaching philosophy; Satish Chandra for instilling a sense of real-world impact; Nenad Medvidovic for encouraging my initial steps into software engineering research; and Kelly Blincoe, Claire Le Goues, Jim Herbsleb, Ciera Jaspan, Eunsuk Kang, Miryung Kim, Christian Kästner, Youn Kyu Lee, Rohan Padhye, Christopher Timperley, Westley Weimer, Tianyi Zhang, Yixue Zhao, and many others for their invaluable advice and mentorship.

The unique advantage of having three advisors was having thrice the number of lab mates. I am grateful to Matthew Davis, Luke Dramko, Hongbo Fang, Luis Gomes, Hao He, Amber Horvath, Kush Jain, Sophia Kolak, Jeremy Lacomis, Jenny Liang, Michael Xieyang Liu, Courtney Miller, Manisha Mukherjee, Huilian Sophie Qiu, Daniel Ramos, Nikitha Rao, Marat Valiev, David Widder, Aidan Yang, and others for their camaraderie, shared ideas, and friendship, especially during moments of doubt and rejection.

My heartfelt thanks also go to my Pittsburgh friends: Emily Yewon Byun, Maria Casimiro, Yae Jee Cho, Simon Chu, Samuel Estep, Morgan Evans, Catarina Gamboa, Seonghwan Hong, Jane Hsieh, Kush Jain, Eunji Jo, Seonghan Jo, Daehyuk Kim, Seyun Kim, Chung Hee Kim, Zeeshan Lakhani, Kyle Liang, Kiwan Maeng, Claudia Mamede, Christopher Meiklejohn, So Yeon Tiffany Min, Jimin Mun, Nadia Nahar, Wode 'Nimo' Ni, Chan Young Park, Hyegyeong Park, Seoyeon Park, Melrose Roderick, Paulo Santos, Ryan Shi, Soyong Shin, Jisoo Sohn, Jimin Sun, Vasudev Vikram, Jenna Wise, Jungdam Won, Jiin Woo, Chenyang Yang, Minji Yoon, and many more. Thank you for providing much-needed fun and solace during those metaphorically and literally harsh Pittsburgh winters. I cherish every brunch, dinner, gym session, yoga class, party, picnic, and movie night we spent together. I will especially miss all the ups and downs I have gone through with my old roommate, Minji Yoon, at Apt 1401, and the numerous spontaneous conversations I had with S3D PhD students at TCS 317 on a myriad of topics.

I extend my appreciation to all my study participants whose contributions were crucial to my research, as well as to the National Science Foundation (CCF-2007482) and Google for their generous financial support. I am also thankful for the constructive feedback from numerous reviewers of my papers.

Last but certainly not least, my deepest gratitude goes to my family—Sikwan Nam (남시관), Yunrye Kang (강윤례), and Dabin Jessica Nam (남다빈)—for their unwavering belief in me, their constant support, and their endless love. I honor my grandmother, my greatest supporter, with love and gratitude in her cherished memory. My partner, Hyun Uk Chae, also deserves special recognition for being my steadfast support, sharing in both the struggles and the triumphs during our PhDs. Though I was physically away, the love and support from my family have been the pillars of my resilience and success. This achievement is as much yours as it is mine.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software engineering is an information-intensive discipline. In every step of the software engineering process, when engineers design a software system, write code, make edits, and triage bugs, various information needs exist. Programmers face a broad spectrum of questions ranging from architectural considerations (e.g., Why was this code implemented this way? [118]) to implementation bugs (e.g., How did this runtime state occur? [124]).

However, much programming information is often not clearly documented. Most of the information needed to understand and use existing code is written by developers who build the systems (e.g., reference documentation), or by users of the systems (e.g., Stack Overflow). As it requires a lot of human efforts to document, it is practically impossible and inefficient to document every piece of information possible, and often gets outdated or becomes obsolete as software evolves. The multi-modal nature (natural language text, source code, diagrams, etc.) of software engineering makes it even harder for programmers to collect relevant information in one place, as it can be dispersed across various mediums and formats.

Thus, programmers spend a significant amount of time searching and foraging for the information they need and organizing and digesting the information they find [117, 118, 125, 145, 151, 161, 196]. Programmers often rely on search engines as a major way of information seeking (e.g., they issue more than 20 search queries every day [270]). They also navigate multiple resources from official reference documentation to large source code bases [117, 161].

It becomes more challenging when a programmer needs to work with unfamiliar code or libraries that require them to learn new concepts or frameworks. When a programmer is not familiar with the domain and the environment, they may struggle with finding relevant information due to a lack of appropriate keywords and the need to evaluate relevance, and even when they find the right piece of information or documentation, there is no guarantee that they can understand it easily. Programmers may also lack the necessary knowledge and experience to effectively navigate and understand the codebase [118].

Thus, to provide effective information support for programmers, it is necessary to understand programmers, that is, the users. At the same time, intelligent solutions should be investigated, so that all the needed information for programming, especially when it is challenging to access, can be prepared for different programmers.

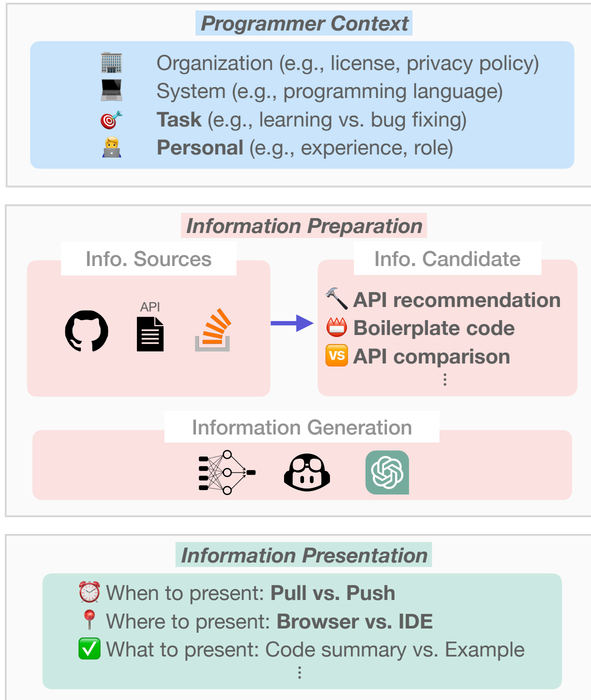


Figure 1.1: Conceptual Framework for Context-aware Information Support

1.1 Thesis: User-Centered Intelligent Information Support

My thesis aims to investigate the following claim:

By studying programmers and software artifacts to understand how programmers work with unfamiliar code, and investigating intelligent solutions to provide effective information support needed for learning, we can enhance the success of programmers' information-seeking for learning.

This thesis explores ways of providing *user-centered* information support for programmers, following the conceptual framework (Figure 1.1) containing three components.

1.1.1 Programmer Context

As there are diverse users with different information needs and preferences, that are often not explicitly expressed by the users, utilizing user context helps such systems accommodate individual differences and enhance the user experience [277]. Therefore, previous research in web search has made strides in understanding users' context using both implicit (e.g., dwell time) [277, 278] and explicit (e.g., item rating) [11, 12] feedback mined from historical interaction data.

Similar to the users of other domains, programmers have distinct information preferences and needs shaped by their experience level, tasks, and learning styles, which lead to varying web search strategies or information foraging patterns [79, 116] in software maintenance tasks [213, 233]. In addition, with an exploratory, mixed-methods empirical study on documentation page-view logs from over 100,000 users of four popular web-based services of Google in Chapter 3, I also showed that programmers have discernible usage patterns when they use documentation, and programmers' contextual factors, such as past experience with a specific product, do correlate with which documentation pages programmers visit. Thus, similar to the web search systems, I believe that programmers' information seeking can be improved by providing them with information support that are designed considering the programmers' context. To do so, I studied programmers and software artifacts like Stack Overflow posts to understand programmers' information seeking behaviors and how they relate to the programmers' context. Also, I conducted human studies with actual programmers, to understand the usefulness of information support designed considering programmers' context.

1.1.2 Information Preparation

As software engineering information is scattered across varying mediums in different formats, automated information support tools have been developed to supplement existing learning materials such as documentation. These tools extract useful information [115, 135, 251, 264] from diverse software repositories so that users accessing learning materials like documentation can easily discover richer information. However, the types of information existing tools provide were limited, at least until when I started working on this thesis in 2018, because most of them employ rule-based approaches, by employing a predetermined set of syntactic patterns that are typically derived from manual inspection. This type of approach has the benefit of simplicity, but generally suffers from low recall when certain information can be expressed in a wide variety of ways, since it is challenging to capture reliable patterns amidst the noise and diversity of real-world text.

To overcome the aforementioned limitations of pattern-matching-based extraction approaches, I used **learning-based information extraction methods**, which is another large branch of natural language processing techniques for information extraction. Supervised learning-based methods rely on the underlying model to learn to recognize patterns directly from labeled data, often achieving high precision given sufficient training data.

However, not every piece of information is documented and can be extracted, so sometimes, it should be inferred based on what is already available. With the advancement of language models that have learned the patterns already and have a sufficiently good understanding [44, 250, 272], the ability to generate information became possible, and I used them to **generate information** to fill in such holes. The generation-based information preparation also allows the system to provide information that fits users' needs, instead of providing existing information that is most similar to the users' needs.

1.1.3 Information Presentation

Even when all necessary information is available, conveying it effectively to programmers poses challenges. The volume of information required for learning can overwhelm programmers if presented all at once, leading to information overload and difficulty in finding

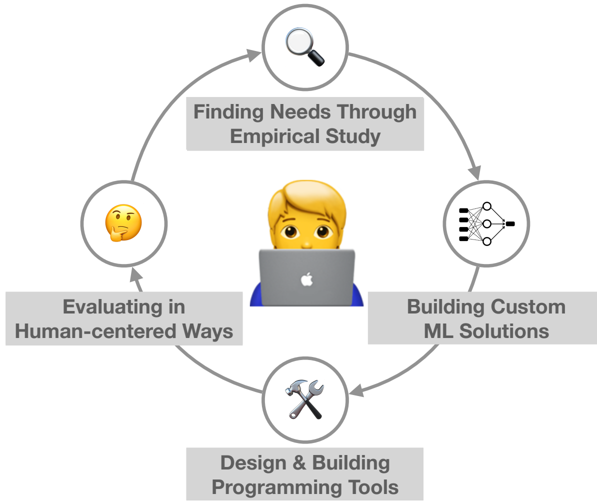


Figure 1.2: My research process

specific information. On the other hand, presenting smaller pieces of information upon request may result in critical information being missed, as programmers might not notice the need for specific knowledge when they are new to a domain or library. Additionally, the location of the information presented can cause context switching and hinder information-seeking efficiency. In this thesis, I explored various ideas for effectively receiving programmers' information needs and presenting information to them, to reduce the cognitive load of programmers and help their learning.

1.2 Research Methods Overview

This dissertation explores programmers' information seeking as a human activity, and explores what intelligent solutions can effectively assist them. To do so, my research follows the cyclical process illustrated in Figure 1.2, covering both inductive and deductive approaches.

1. Finding needs through empirical studies: Conducting empirical studies to understand programmers' needs [18, 100, 172, 174, 179].
2. Building custom ML solutions: Building and training machine learning models based on both the insights gained from others and my own studies [172, 174, 175, 177].
3. Designing & building programming assistance tools: Using the previous custom deep learning models or off-the-shelf Large Language Models (LLMs) to build tools for programmers [172, 174, 177, 179],
4. Evaluating in human-centered Ways: Evaluating the tools using HCI methods [176, 177].

1.3 Thesis Outline

In the rest of this thesis dissertation, I first discuss the history of studies of programmers' information-seeking and information support tools (Chapter 2). Next, I report on my large-scale mixed-methods study that confirms that programmers do use documentation differently depending on their user characteristics, which was missing from the literature (Chapter 3). I then introduce four prototype intelligent information support tools that extract and present information to provide **user-centered information support** for programmers working with unfamiliar code, concepts, application domains, and APIs. Motivated by these findings, this thesis finally explores the possibility of personalizing information support for programmers, by leveraging the programmers' contextual factors (e.g., familiarity with application domains) to enhance information support.

All of the works included in this proposal were done as part of collaborations with others, and to acknowledge that, I use *we* instead of the singular first person in the following chapters.

Chapter 2

Background and Related Work

In every phase of modern software engineering, developers need to work with unfamiliar code, and how well they learn such code influences their productivity significantly. So it is important to understand how developers learn and comprehend unfamiliar code. More specifically, researchers have been studying *what* information they need, and *how* they find such information. To support programmers in seeking such information, researchers and practitioners have designed and built approaches, by exploring various ways of processing and presenting information.

2.1 Information Needs of Programmers

Prior work has identified a variety of information needs of programmers, by using various research methods. For example, Ko et al. [118] found 21 types of information developers look for, through an observation study of 49 professionals. The information needs were further categorized into 7 categories, including writing code, submitting a change, triaging bugs, reproducing a failure, understanding execution behavior, reasoning about design, and maintaining awareness. Sillito et al. [234], similarly, conducted two observational studies to develop a catalog of 44 questions programmers ask during software evolution tasks. Some of these information needs can be relatively easily satisfied, but some are not, and LaToza and Myers [124] conducted a survey of 179 developers to identify hard-to-answer questions developers ask about code. To understand developers' information needs in the wild, researchers have also analyzed developers' web search behavior. Rao et al. [205] for example, found six categories of intent for searches through analyzing the logs of millions of Bing search queries.

Many developers indicate that they satisfy their information needs through software documentation. However, not everything is available in the documentation. Thus, some researchers have conducted empirical studies to derive concrete insights into what developers need from the documentation. For example, developers have expressed the need for complete and up-to-date documentation [14], because many developers rely on API reference information and code examples [158, 185] when they approach documentation with a problem or task in mind [158]. Developers also asked for a concise overview of the documentation, more rationale, and adequate explanation for code examples [158, 210, 212, 255]. Researchers have also proposed tools that can assist in more effective usage of documentation, by providing easier access to the documentation contents within developers' workflow [89, 101, 187].

There is also a rich literature studying Stack Overflow to understand what challenges developers face in practice in learning and using unfamiliar software systems, e.g., [10, 27, 142, 146, 170, 256]. For example, Beyer et al. [31] categorized Stack Overflow questions into seven high-level categories, by harmonizing the categories defined in the previous works [16, 218, 252]: API change, API usage, Conceptual, Discrepancy, Learning, Errors, and Review. Various approaches were used to identify those categories, including but not limited to manual categorizations [218, 252], topic modeling [16], or k-nearest-neighbor (k-NN) clustering [30].

2.2 Information Seeking of Programmers

To satisfy their information needs, developers often spend a lot of time searching for information using varying strategies. Researchers have investigated how developers seek such information by studying them. Specifically, many researchers have used the lens of information foraging theory (IFT) to understand programmers' information seeking. IFT assumes that human information seeking abilities have evolved like how animals forage for food [198], where a programmer is a *predator* hunting for information *prey* (e.g., piece of information) in the *information environment* (e.g., search engine), following *information scents* (e.g., keywords) that help identify the information most suited to their needs. Researchers have applied IFTs in understanding developers information seeking in requirements engineering [182], debugging [75, 128], and program maintenance [75, 127, 204]. In addition to professional programmers' information seeking, the foraging strategies of end-user programmers [120] have also been investigated.

Although at a high level, some common general strategies for information seeking exist, developers' needs and how they seek such information can vary with experience [66, 116, 125, 133, 151, 203], roles, and learning styles [52, 66, 133, 159]. For example, Costa et al. [52] found that developers with less experience with the software tended to use more types of documentation than more experienced users, and that tutorials and how-to videos were used by a greater percentage of newer users, and the newer users tended to use tech notes and forums less. Similarly, in the literature, programmers are sometimes categorized into three personas, which summarize their information seeking and problem solving strategies – systematic, opportunistic, and pragmatic [49] – that reportedly also correlate with documentation use [159]. For example, opportunistic developers tended to use documentation in a task-oriented way, focusing less on the general overview of APIs or the suggestions described in the documentation; in contrast, systematic developers tried to understand how the API works before diving into the details of a task, by systematically searching and regularly consulting documentation provided by the API supplier [159].

2.3 Intelligent Techniques for Information Support for Programmers

To supplement existing learning materials and make programmers information seeking more efficient, many intelligent techniques have developed and used in software engineering settings. In this section, I categorized them into three difference groups: pattern-matching based extraction, learninb-based extraction, and generation.

Pattern-matching based Information Extraction. The pattern-matching-based approaches employ a predetermined set of syntactic patterns that are likely to yield valid information, typically derived from manual inspection. Such methods have been used in SE research a lot, including for extracting API-related statements from reference documentation [148, 211]. This type of approach has the benefit of simplicity, but generally suffers from low recall when the target information can be expressed in a wide variety of ways, since it is challenging to capture reliable patterns for all of these.

Learning-based Information Extraction. On the other hand, (semi-)supervised learning methods rely on the underlying model to learn to recognize patterns directly from labeled data. Most state-of-the-art information extraction tools use such deep neural networks, often achieving high precision given sufficient training data. Compared to pattern-based methods, neural models tend to need substantially more labeled training data, which makes it harder to apply them to new environments. To overcome this obstacle, researchers commonly utilize models that were pre-trained on large, existing datasets [17, 230]. Such models can be fine-tuned to new domains with relatively few training samples.

Information Generation using Large Language Models. More recently, Large Language Models (LLMs), such as GPT [37], Gemini [20], and LLAMA [250], have revolutionized how we solve various tasks in natural language processing. Based on the idea of attention and the architecture of the Trnasformer, LLMs are pre-trained with vast datasets of natural language processing and code (many GBs and TBs), allowing better generalization and domain adaptation than the previous models, and even showing emergent abilities such as reasoning, decision-making, etc. Combined with instruction tuning, it has achieved a significant performance gain in many tasks, including summarization, translation, question-answering and many more, by generating responses given natural language prompts. Thus, the information support for programmers is also moving from extraction-based approaches to generation-based ones.

2.4 Existing Information Support for Programmers

To overcome some of the challenges programmers face in information seeking, researchers have built tools that can supplement existing learning resources like documentation. One popular approach has been to extract knowledge from Stack Overflow and augment more traditional forms of documentation, e.g., [104, 107, 190, 209, 251]. Researchers have also been developing tools to more closely integrate such knowledge into the development workflow, e.g., [107, 201, 202, 258]. Many types of knowledge have been in focus, including common use and misuse patterns [23, 209, 257, 258, 281], caveats [135, 136, 251], opinions on different quality attributes (e.g., usability) [42, 140, 141, 208, 254], or more generally any Stack Overflow posts discussing some given API methods, such as those invoked in the developer's local integrated development environment (IDE) context [221].

The potential and applicability of LLM-based AI programming tools have also been actively studied by many researchers. Numerous empirical studies [73, 96, 131, 223] evaluated the quality of code or explanations generated by LLMs, to test the feasibility of applying LLM into development tools [144, 246, 285] and to computer science education [18, 96, 223]. Several studies have also compared LLM-generated code and explanations with those authored by humans without LLM assistance [96, 131, 194], demonstrating that LLMs can offer reasonably good help for developers or students when carefully designed and used.

Fewer studies have specifically explored the *usefulness* of LLM-based programming tools [26, 113, 138, 166, 219, 222, 261, 272, 285, 285] with actual users or their usage data, and many of these studies have focused on code generation tools like CoPilot [4]. For instance, Ziegler et al.[285] analyzed telemetry data and survey responses to understand developers' perceived productivity with GitHub Copilot, revealing that over one-fifth of suggestions were accepted by actual developers. Several human studies were also conducted. Vaithilingam *et al.* [261] compared the user experience of GitHub Copilot to traditional autocomplete in a user study and found that participants more frequently failed to complete tasks with Copilot, although there was no significant effect on task completion time. Barke [26] investigated further with a grounded theory analysis to understand *how* programmers interact with code-generating models, using Github Copilot as an example. They identified two primary modes of interaction, acceleration or exploration, where Copilot is used to speed up code authoring in small logical units or as a planning assistant to suggest structure or API calls.

Although these studies have increased our understanding of the usefulness and usability of AI programming assistants in general, and some of the insights apply to information support, they do not show the opportunities and challenges of AI-powered tools as information support tools, with a few exceptions [152, 219]. MacNeil et al. [152] examined the advantages of integrating code explanations generated by LLMs into an interactive e-book focused on web software development, with a user study with sophomores. They found students tend to find LLM-generated explanations to be useful, which is promising, but the study was focused on providing one-directional support in an introductory e-book which is different from user-oriented need-based information support. The Programmer's assistant [219] is the closest to our work. The authors integrated a conversational programming assistant into an IDE to explore other types of assistance beyond code completion. They collected quantitative and qualitative feedback from a human study with 42 participants from diverse backgrounds and found that the *perceived* utility of the conversational programming assistance was high.

Chapter 3

Confirming Different Information Needs of Developers[1]

Due to the lack of studies on different developers' varying information needs, although there were some reports that developers' information seeking may vary depending on their different contexts, there has been no solid evidence that there is a meaningful relation in between developers' contexts and their information needs. Thus, in this section, we first confirm if the developers' context makes a meaningful impact on their information needs.

3.1 Introduction

Almost no modern software system is written from scratch, and many third-party libraries and software services are available to be reused and composed. Thus, the productivity of programmers in many domains and contexts depends on rapidly searching for relevant information to make decisions about third-party libraries or services [169, 188], and learning to use them correctly for their own systems [210, 212]. Practitioners spend a lot of time searching for and digesting relevant API information, e.g., 20% of their time according to Brandt et al. [34]. And while many sources are useful, including code examples, question and answer (Q&A) websites, and expert advice, in obtaining API-relevant information, the official software documentation remains essential [43, 81, 212].

Efforts to improve software documentation span decades, with many researchers studying documentation design experts and users to catalogue problems [43, 210, 212] and recommend best practices [210, 212, 263]. Much documentation now follows such guidelines, and new tools [157, 251] and ideas [215] have been proposed to further support developers' information needs based on such studies.

Most of these efforts involve qualitative research methods such as interviews [158, 185, 212] or lab studies with human participants [64, 100, 108, 160]. However, while generally highly informative for understanding usability issues during the early design review phase [53], such methods capture only what participants say they do, or what they do in

[1]This chapter is adapted from Nam *et al.* [179]

a controlled setting. Moreover, the number of participants that can be observed this way is typically small.

Our research goal is similar to most prior software documentation research—improving the design and usability of documentation. However, our approach is novel and complementary—mining documentation page-view logs at scale. Web mining has long been used to analyze people's experience online in more general contexts, e.g., user engagement in online news reading [121] or user satisfaction during online shopping [240]. We argue that similar approaches could apply to documentation since, after all, documentation webpages are just another type of webpage. In other words, any software documentation published on the Web, such as Adobe Photoshop and Autodesk AutoCAD, which comprise numerous documentation webpages and users, could potentially be analyzed using an approach similar to ours.

We believe that our large-scale log analysis will complement existing documentation review methods, by providing the following additional methodological advantages:

- **Allowing more scalable, computational design review**: Relying on web analytics to understand documentation usage is considerably less expensive for software providers if they have access to telemetry data for the documentation pages (e.g., from self-hosted web servers)—we expect that one quantitative user experience researcher on staff could analyze the page-view logs of hundreds of thousands of users of dozens of APIs or services in a matter of days, if not hours, following our methodology. In contrast, qualitative studies tend to focus on one API or service at a time, may require complex participant recruitment, and usually involve orders of magnitude fewer subjects. They also often involve monetary compensation (e.g., Duala-Ekoko and Robillard [64] compensated each participant with $20 for a one-hour programming study), in addition to the researcher team's time for running the studies, and collecting and analyzing the data.
- **Allowing for the discovery of less-studied documentation user groups**: As most of the smaller scale studies require study design prior to the data collection, researchers specify research questions and target participants in advance. For documentation, as it is expensive to conduct these qualitative studies, most of the studies have focused on the professional developers who use the documentation for API learning [66, 117, 158, 159], the main target usage scenario of software documentation. Large-scale log analysis, on the other hand, does contain the entire user population's data, allowing the discovery of more diverse user groups, including users who use documentation to make API adoption decisions (e.g., Product explorers in Section 3.3.3), or users who are only concerned with the cost of querying APIs (Financial users in Section 3.3.3).
- **Capturing a perspective less prone to response biases:** With qualitative studies where users need to report (e.g., survey, interview) or show their behaviors (e.g., observation study, lab study), the data can only capture what participants recall or show, which might be different from what actually happens in the wild, i.e., response bias [156]. As qualitative studies often ask participants to focus on "software documentation regularly used by participants" [43, 76, 200] to help participants recall

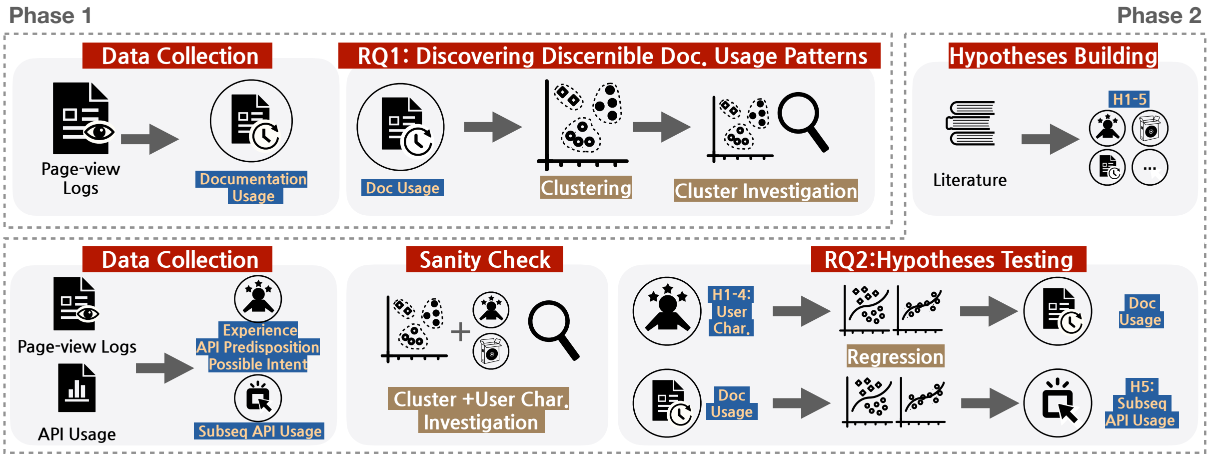


Figure 3.1: Overview of our data collection and analysis.

specifics of their experience, the bias might be even amplified. Log analysis can minimize the response bias, as the telemetry data is automatically collected.

However, to be clear, traditional non logs-based approaches can be extremely valuable and we don't advocate replacing them. Instead, we argue that a logs-based analysis like ours could be used as a first pass, to guide the design of more complex (but rich in terms of insights) approaches such as human studies.

To this end, we report on an exploratory, two-phase, mixed-methods empirical study of documentation page-view logs from over 100,000 users of four popular services of Google; see Figure 3.1 for an overview. The documentation page-view logs we had access to were privacy-preserving in a number of ways (section 3.2.3) and contained only aggregated monthly totals of which specific documentation pages someone visited and how much time they spent on each page, over the course of that month (possibly across multiple sessions). This is likely a common scenario — many companies and open-source projects can be in a position to instrument their documentation web servers to collect such basic telemetry data; at the same time, it may be undesirable to collect more fine-grained or personally identifiable data for privacy reasons. The research challenge, therefore, is determining whether there is enough signal in this big but shallow data to generate actionable insights for the documentation designers by mining it.

Overall, our two-phase study argues that the answer is "yes." In Phase I (section 3.3) we set out to explore the log data, looking for patterns of page views and trying to explain them *without* knowing who the users are or anything else about them. Given the large size of our sample, we do this using a combination of automatic clustering analysis followed by qualitative explorations and show that many page-view clusters are discernible in the log data.

In Phase II (section 3.4) we set out to formalize and generalize our qualitative observations from earlier. In an effort to understand *why* such discernible patterns exist in the page-view data, we formulate testable hypotheses about the "average" characteristics and subsequent behavior of those users, based on findings from the literature on general information seeking of developers [34, 79, 118, 128, 150, 205, 234, 245, 282] and from small-scale documentation usability studies [64, 100, 108, 117, 159, 160]. We then use the fact that all users who make requests to Google services, or had otherwise registered for accounts on Google and were browsing the documentation pages while logged in, have persistent (pseudonymized) IDs across the data. This way, we join the page-view log data with user-level data about their experience with the respective service and the platform overall, and with data about subsequent requests (after the documentation page views) to the service APIs. We first revisit the clustering results and check if the hypotheses built based on the general information seeking literature make sense in the documentation usage setting. We then conduct multiple regression analysis to test the hypotheses formulated in the first phase on this aggregated data, finding multiple sizeable correlations between patterns of documentation page-views, on the one hand, and user-level characteristics and subsequent API use, on the other hand. That is, one's level of experience partially explains their documentation browsing patterns, and one's documentation browsing patterns partially explain their intent to subsequently use the APIs.

While not intended as an exhaustive exploration of all patterns of documentation page views identifiable for the four Google services in our sample, our study does show that it is feasible to analyze page-view logs at scale to inform documentation design reviews, or to corroborate observations from smaller-scale studies [52, 158] or the anecdotal experiences of professional software engineers. Concretely, we argue (section 3.5) that even when not knowing anything else about the documentation users, the interaction histories and dwell times that are likely to be contained in the documentation page-view logs can provide actionable information at scale for providers which can help companies decide which documentation pages to redesign, and even to potentially automatically personalize documentation pages in the future, to better align with their users' needs.

3.2 Dataset

We started by compiling a dataset of documentation page-view logs for four web-based services of Google.

3.2.1 Product Selection

Google provides hundreds of web-based services to a diverse group of users and businesses, and most of the services come with one or more types of APIs, including REST APIs and gRPC APIs, as well as client libraries. However, since our study is primarily exploratory in nature, we selected only four Google products following a maximum variation sampling strategy [241], to gain an understanding of documentation use from a variety of angles. Concretely, we diversified our sample in terms of the application domain (machine learning / natural language processing vs. event analytics and management), usage context (operations infrastructure vs. potentially end-user facing), and product size and complexity (ranging from a few API methods to hundreds of API methods offered by the products). These differences are also reflected in the documentation pages, which vary in their contents

Table 3.1: Types of documentation provided for the selected products.

Genre	Type	Description
Meta	Landing (L)	Links to core documentation pages.
	Marketing (M)	A brief introduction to a product, incl. the benefits, target users, and highlights current customers.
Guide	Tutorial (T)	Walkthroughs for common usage scenarios.
	How-to (H)	Guidance on completing specific tasks.
	Quickstart (Q)	A quick intro to using the product.
	Concept (C)	Explanations for product- or domain-specific concepts.
Dev	Reference (Ref)	Details about the API elements, including API endpoints and code-level details.
	Release note (Rn)	Specific changes included in a new version.
Admin	Pricing (P)	Pricing information.
	Legal (Lg)	Legal agreement details.
	Other (O)	Other resources not included in other types, e.g., locations of the servers.

across the four products, e.g., with more or less marketing materials, how-to guides, pricing information, *etc.* All four web-based services we selected are popular, having large user bases. Specifically, P1 and P2 are machine learning / natural language processing-related products for machine translation and text analysis. And P3 and P4 are operations-related products for managing event streams and log data.

3.2.2 Documentation Usage Data Preprocessing

For each of the four products, we had access to pseudonymized **documentation page-view logs** [143] for users who visited the documentation from May 1, 2020 to May 31, 2020, UTC, while they were logged into their accounts. The page-view log data are collected automatically by the documentation servers and include the specific documentation pages visited by someone, as well as the timestamps and dwell times for each visit. We use *dwell time* to estimate user engagement with the content, following prior work [78, 277]. The data was aggregated at the month level, partially due to the volume of data being analyzed, and also to enhance the pseudonymization of the data for the privacy protection of the users (see Section 3.2.3 for details).

To reason about more general patterns of documentation use, we further labeled each individual documentation page (URL) in our sample according to its contents into one of 11 possible *types* and four aggregate categories (or documentation *genres* [66]) summarized in Table 3.1. For the first-level labeling we relied on an internal mapping table created by the documentation team, which contains meta information for the different documentation pages, including what we refer to as the *type*. The second-level labeling reflects our subjective grouping of documentation types into four high-level categories that provide related kinds of information and presentation format; we expect that these are likely to be consulted together given specific tasks and target reader familiarity with the API. To this end, we followed an open card sorting process involving two authors, one of whom is a domain expert. There were two documentation types (Other and Release note) that two authors did not agree on. The two authors resolved disagreements by comparing their definitions of genres and the rationale behind the categorization, until there were no more disagreements. All official Google documentation pages in our sample were assigned to exactly one of the documentation types and genres listed in Table 3.1, and the documentation of all four products we analyzed included all 11 documentation types. The volumes of each documentation type varied, but in general, each product documentation consisted of around 5 pages of Meta genre documentation, around 150 pages of Guide genre documentation, around 300 pages of Dev genre documentation, and around 15 pages of Meta genre documentation. The documentation of all products followed the same documentation style guide [9]; thus, the contents and styles used for each documentation type are consistent, even across the products.

Finally, we followed prior work by Fox et al. [78] and excluded page-view sessions shorter than 30 seconds, since these are more likely to be noise (e.g., a user accidentally clicked the documentation page and then left the page quickly) than meaningful visits.

3.2.3 Privacy Protections

As we analyzed the user data of Google, we followed Google's strict privacy and data access policies [7, 8] which ensure appropriate, legal, and ethical access, storing, and analysis of user data. This included, but was not limited to, internal privacy reviews with security and privacy experts, the use of differential privacy processes (more details below), wipeout and data access processes, and more. In addition, the study designs were reviewed by internal research ethics experts, methodological experts, and product experts.

We also used numerous privacy protection techniques throughout our work. First, all user-level data was pseudonymized before any of the authors had access to it. Pseudonymization maps users' accounts to randomly but persistently generated pseudonymized IDs. As the IDs were randomly generated, they could not be reversed without access to a mapping table, which the authors did not have.

Additionally, usage data was aggregated (e.g., we looked only at the number of API requests aggregated at the month granularity, not individual API requests), and had Differential Privacy [268] applied. In brief, differential privacy was used to apply sufficient noise to the aggregations such that individual records could not be identified, but the overall shape of the dataset remained meaningful / sufficiently accurate. Using established best practices, and based on guidance with internal privacy experts, we used Epsilon < 1.1, (where lower numbers yield higher degrees of privacy protection). This allowed us to analyze trends in user behavior while preserving the privacy of those in the dataset. Later in this chapter (e.g., in Figure 3.4) we include polar plots of our clusters, but choose only to visualize clusters with over 500 users, as an additional privacy consideration.

3.2.4 Limitations & Threats to Validity

First, a month might not be enough to capture the full process of learning and adopting a complex API, on the one hand, and might not capture the differences in documentation usage patterns that appear in significantly shorter periods, such as patterns in an hour or in a day, on the other hand. It is also possible that some users happen to register in the middle of our one-month window, or one may learn an API intermittently over a few months. However, such an operationalization was necessary to balance data collection complexity, privacy, and analysis scale. Given that the size of Google's general user base is very large, and the services we analyzed were already all mature, we believe that our dataset should still capture snapshots of developers at every stage of the learning process, as well as cyclical patterns of use, without the number of newly registered or intermittent users significantly affecting our results.

The use of a particular month (May 2020) can also be too short to generalize, as the documentation access might change throughout the year, and it could have been influenced by any major event related to the four target products. We did our best to choose a month without major events related to the four products we studied, and there was no event for P1 and P2, but there were two minor feature additions for P3 and two minor beta releases for P4. However, as we chose to analyze popular products with large active user groups, it is practically not possible to choose a month without any updates. We believe that selecting four very different products reduces the risk of biasing the results in a meaningful way, especially given that there was no major event that affected all of the four products during that period.

The documentation and API usage data we used for our analysis can only provide a partial representation of the entire user group's usage. Since the documentation usage data only includes page-view logs of logged-in users, the analysis does not capture the behavior of users who were not logged in, who may behave differently. In addition, the aggregated API usage data can only partially represent the outcome of API learning. For example, while making an API request requires a user to sign into the platform, browsing documentation does not, so not all documentation usage is linked to the corresponding API use. Multiple developers can make API requests using shared corporate accounts, which can obfuscate the connection between their documentation and API use.

Although the dwell time was logged when the pages were actually accessed, our measurements of time spent on each documentation page are only (over)approximations. For example, some users may keep a page open without actively consuming it the whole time, while they grab a coffee or read code from their IDE. As part of our analysis, we applied several heuristics and filters to our data to identify and remove outliers and noise, as described in section 3.3.1.

The analyses at the documentation type and genre levels introduce threats to internal validity: the analyses might not capture the possible influence of content and length of individual documentation pages, and other external confounding factors. However, the abstraction of data was inevitable due to the number of documentation pages available. We provide potential ways of introducing additional internal validity control for page-view log analysis in Section 3.5.

While our dataset includes many relevant variables, it certainly does not include all. For example, a user's position or role, their expertise in programming or in the product domain, the specific tasks during which they visited documentation pages, and the actual contents of the documentation pages, are all likely to also correlate with differences in documentation usage but are absent from our data. Moreover, we only analyze data for four products of Google, therefore it remains unclear how our findings would generalize.

Finally, the page-view analysis can only be conducted after the documentation has been available for some time, allowing for the accumulation of extensive logs. Therefore, our analysis may not be applicable for documentation writers who need to assess their content pre-release or for documentation related to products with a limited user base.

Thus, we do not expect page-view log analyses like ours to obviate human studies or other more precise research methods. However, we do expect they could be fruitful as a first step or in conjunction with more precise but more costly research methods.

3.3 Phase I: Discerning Documentation Use Patterns in Log Data

As an initial exploratory investigation to help contextualize our data, we conducted cluster analysis. This phase was necessary because although we know that developers will use documentation differently, we still know little about how much and in what ways it will differ "in the wild" and in our context. To efficiently explore the large dataset, we first used an automatic clustering analysis to discover discernible documentation usage patterns, and used sampling and qualitative analysis to further investigate the patterns.

3.3.1 Data Preparation

Table 3.2: An example of our documentation page-view data used for the clustering analysis.

User	Product	Dwell Time (minutes)				
		howto	marketing	reference	...	concept
0	P3	1	0	0	...	0
1	P3	35	0	0	...	0
1	P4	1.4	0	0	...	0
⋮	⋮	⋮	⋮	⋮	⋮	

In preparation for clustering, we first aggregated each user's total dwell times (i.e., times spent on the different pages) in May 2020 across the 11 documentation page types in Table 3.1. We recorded separate entries for each of our four separate products, if the same user happened to access documentation pages for more than one of the products that month. We then represented each user's documentation visit profile as a vector of 11 elements, capturing the total times spent across each page type that month.

Note that as a precaution before clustering, we filtered out outliers with total dwell times (sum over the 11 page types) outside of the $[\mu - 3\sigma, \mu + 3\sigma]$ interval (i.e., more than three standard deviations from the mean), as is customary. In our sample, this corresponds to users who stayed shorter than 1.39 minutes or longer than 961.91 minutes in total across all documentation pages of each of our four products during the month (in May 2020). In addition, as the distributions of dwell times we observed tend to be right-skewed, we log-transformed all positive values. This is a common transformation [217] when the data vary a lot on the relative scale, as in our case — spending 10% more time on a page is arguably much more noticeable for a 2-minute dwell time than a 10-hour dwell time.

3.3.2 Methodology

Out of many clustering approaches available, we adopted a protocol proposed by Zhao et al. [283], which is particularly well suited for large datasets. A common challenge with standard clustering methods is determining the appropriate number of natural clusters. Typically, one either chooses the number of clusters a priori, or applies techniques to automatically determine the "optimal" number of clusters. The former scenario is not applicable in our case (we do not have any empirical basis to expect a particular number of clusters), while traditional techniques to select the number of clusters automatically tend to be slow for large datasets like ours. The key innovation in the protocol by Zhao et al. [283] is combining two standard clustering techniques: first using a fast clustering method (k-means) to reduce the dimensionality of the clustering problem, and then applying a second clustering method (MeanShift) that automatically determines the number of clusters. This is computationally effective, as the second method only runs on the centroids generated by the first (k-means). To select the number of clusters as input for the first (fast) method, one typically chooses a significantly larger number than the plausible number of natural clusters, expecting that the second method will merge closely located centroids eventually to match the natural clusters. In determining the quality of the clustering results from the different parameters used, we used the following clustering performance score, as per Zhao *et al.*.

$$cp = 0.3 * E + 0.23 * D + 0.23 * \frac{k - m}{k} + 0.23 * \frac{N - n}{N} \tag{3.1}$$

In this score, the first and the second factors are used to reward the clustering performance using two well-known metrics: Shannon's entropy (E) and Dunn's index (D). The probability used for calculating Shannon' s entropy score is the normalized number of users in each cluster. Thus, entropy assigns a high value to clustering results that have a uniform distribution of users across clusters. Dunn's index measures the compactness and separation of the clusters, by calculating the ratio of the smallest distance between observations not in the same cluster to the largest intra-cluster distance. The third and fourth factors are to penalize clustering results that are too naive or complex. The third penalizes the results that are too complex, that do not improve over the naive k-means results, where m (the number of clusters after MeanShift clustering) is close to k (the number of target k-means cluster that is significantly larger than the number of natural clusters). The fourth factor penalizes results that one big cluster contains most of the users in the dataset, where N is the number of total product users and n is the number of users in the biggest cluster.

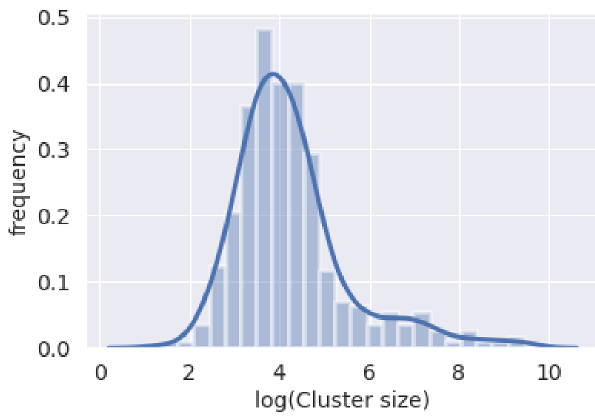


Figure 3.2: Distribution of the log-transformed number of users per cluster.

After trying several values for k and eps in the equation 3.1, we obtained the highest cp score, 0.50, which is comparable to other works [247, 283], with E = 0.71, D = 0.15, m = 316, N = 94096, n = 9789. This result was obtained for $k = 400, eps = 1.25$, resulting in 320 clusters. Most clusters consist of around 300 users and there are 18 clusters consisting of more than 1,000 users. The distribution of the number of users per cluster can be found in Figure 3.2.

3.3.3 Resulting Clusters

Figure 3.3 provides a summary of the resulting clusters, illustrating a wide array of documentation usage patterns within the 320 clusters. These patterns are evident through the variations in page views across the 11 documentation types. To comprehensively investigate these distinct patterns, after excluding small clusters with fewer than 100 users, we conducted open coding. I assigned codes to the clusters, refined them iteratively, and subjected the emerged categories to a thorough review by the three other researchers. As examples, we present four codes that were frequently assigned to clusters, along with representative examples of these clusters for reference.

Product explorers (Clusters 11, 16, 21). The time this group spent on documentation is not seemingly enough to digest the information in the documentation, and would not help one to actually use a product. Furthermore, considering that the clustering was done with a month-full of user logs, visiting only one type of documentation for few times is not likely a usage pattern of an actual product user. Cluster 16, for instance, only visits one

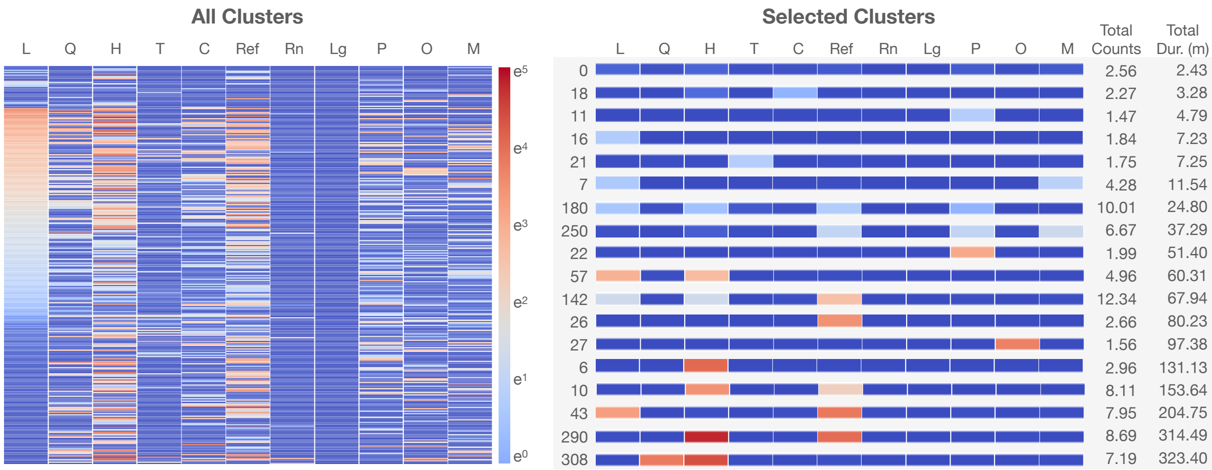


Figure 3.3: The heatmap of centroids of the 320 clusters (left), and a subset of them highlighted (right). Each row represents the documentation usage of each cluster (see Table 3.1 for the documentation type codes). The color indicates the dwell time in minutes, with the intensity encoded in e^n of time. The average total counts (# of documentation pages visited in May) and the average total dwell time (sum of dwell time on 11 documentation types) are also shown for the selected clusters (right) to help with interpretation, and the rows are sorted by the average total dwell time. For example, users of Cluster 18 (2nd row from the selected clusters) spent 3.28 minutes on average on the product documentation among 2.27 page visits on average, and spent $\approx e^1 = 2.7$ minutes on Concept type documentation.

specific type of documentation, landing, a few times, and spent only a short time on the documentation (less than 10 minutes) on average.

Documentation Explorers (Clusters 7, 18, 180). Users in Cluster 180 were similar to product explorers in the sense that they only stayed for a relatively short time (less than 30 minutes over the month), but different in that they visited more documentation pages of multiple types. We infer that these users might be new to the documentation and might be exploring it to see the available information. For example, we speculate that they might have visited Landing documentation by searching for the product name in search engines, checked the prices from the Pricing page to see if they can adopt the product, and looked around Reference documentation to see the features available.

Task-oriented users (Clusters 6, 26, 27). Users of Cluster 6 showed more distinct behaviors. Although they only visited one specific type of documentation for few times like product explorers, they stayed on the documentation pages much longer (on average, 131.13 minutes over the month). Based on the amount of time spent on documentation pages, we can infer that these users spent enough time to find what they were looking for from the information, and to fully digest it. From the number of visits, the users did not seem to explore the documentation, but stayed on few specific (or single) pages that they were interested in; this indicates that they were only interested in some of the product features, rather than an overall understanding of the product.

Versatile users (Clusters 43, 290, 308). These users visited multiple types of documentation pages and stayed long enough time on each of the type. For example, users in Cluster 290 seemed to be interested in the specific tasks described in How-to guide pages since they spent enough time on them, and perhaps visited Reference documentation from time to time when they needed more low-level information on the API calls and parameters.

We also discovered many other interesting documentation usage patterns, such as **Financial users** (Cluster 22), who stayed in Pricing documentation which only contains pricing information for an hour, and **Server engineering users** (Cluster 27), who almost exclusively visited Other documentation which provided resource-relevant information like locations of the servers.

3.4 Phase II: Factors Associated with Documentation Use

With the exploration of clustering analysis results, we were able to discover various usage patterns, including those that were not actively discussed in the existing literature [66, 117, 158, 159], like product explorers or documentation explorers. However, we could only speculate about the intention and background of the users behind those diverse documentation usage patterns. Thus, we now bring together our informal observations from Part I with the literature on general information seeking and small-scale documentation studies, to derive and test hypotheses explaining the different usage patterns based on user characteristics.

3.4.1 Hypotheses Building

Given the absence of established theoretical frameworks elucidating documentation usage behaviors, we have chosen factors that might be associated with the developers' documentation usage, informed by the prior work on developers' general information seeking in web search or software maintenance settings, and observations from the small-scale documentation studies [49, 52, 79, 133, 159].

Experience. Many studies have shown that the information seeking strategies of developers vary by their experience levels [66, 116, 125, 133, 151, 203]. For example, Costa et al. [52] found that documentation users with less experience with the software tended to use more types of documentation than more experienced users, and that tutorials and how-to videos were used by a greater percentage of newer users, and the newer users tended to use tech notes and forums less. Thus, we hypothesize,

$\mathbf{H}_1$. *High experience levels are positively associated with accessing documentation covering implementation details* (*Dev genre*)*, whereas lower experience levels are positively associated with accessing documentation covering an overview of the products* (*Meta genre*).

Product Type. Differences in typical usage contexts of the products, such as project complexity and task categories, also influence developers general information seeking [66, 79, 163]. Within our dataset, P1 and P2 are application APIs whereas P3 and P4 are operations-related products for managing event streams and log data, and we expected to see different characteristics will come with different documentation usages, and we anticipate that these distinct characteristics will be associated with different documentation usage patterns. For instance, users of infrastructural APIs are more likely to be engaged in the maintenance of large-scale software projects, which implies a greater interest in system-level products and in system-level quality attributes. Conversely, application APIs are commonly adopted by smaller projects where the applications themselves serve as core components. Consequently, we expect that users of documentation for different products will tailor their utilization accordingly. Thus, we hypothesize:

$\mathbf{H}_2$. *Documentation usage of application APIs* (*P1, P2*) *differs from that of large-scale infrastructural APIs* (*P3, P4*).

Documentation Type Predisposition. Previous research has found that developers adopt different work styles, motivations, and characteristics, and they solve programming tasks differently [49], and human studies with documentation usage also reported similar findings [159]. The work styles of developers are less liable to change over time as opposed to levels of expertise, educational background etc. Thus, we hypothesize that we can see the similar patterns in the page-view logs, that developers will stick to documentation that suits their general information foraging strategy formed by their needs and preferences, without changing their documentation usage behavior much over time.

$\mathbf{H}_3$. *Users tend to use the same documentation type over time.*

Possible Intent. Prior work [34, 205] found that developers' web search behaviors vary with their information seeking intent: they visit different types of web pages, use different queries, and overall interact with webpages differently. In particular, developers were more likely to visit official software documentation during reminding sessions and third-party tutorials during learning sessions. Developers also tend to spend tens of minutes with learning intent, but only tens of seconds to remind. Times spent in between the two extremes were mostly with clarification intent. We posit that a similar behavioral pattern can be identified within documentation-based information seeking, wherein users invest substantial time per visit when their objective is to grasp complex concepts or protocols. Conversely, they allocate less time when verifying straightforward facts or utilizing documentation as an external memory aid [117]. Thus, we hypothesize:
H$_4$. *Users who exhibit extended average page dwell times are more inclined to access documentation that offers tutorials* (*Guide genre*)*, whereas users with shorter average page dwell times are more likely to access documentation providing straightforward factual information* (*Admin genre*)*.*

Subsequent API Use. From multiple empirical studies, developers have reported that the quality of documentation is a highly influential factor in API selection process [262], and failure in effective information seeking within documentation leads them to give up on using the APIs [206]. Developers specifically reported that they examine the documentation upfront to determine "if there are good examples or tutorials that clearly explain how to use the library" [262] before they decide to adopt a library, showing the need of onboarding materials. Thus, we expect that:
H$_5$. *Accessing documentation pages providing technical information for newcomers* (*guide genre*) *is positively associated with subsequent API calls by the same users.*

3.4.2 Data Preparation

Table 3.3: An example of our API usage data used for the qualitative investigation of clustering results and the regression analysis.

User	Product	Account Age (years)	Past Succ. Req.	Future Succ. Req.
0	P3	5.96	0	0
1	P3	2.33	1222	859
1	P4	2.33	0	0
⋮	⋮	⋮	⋮	⋮

We collected **pseudonymized user-level data** and **API usage data** to extract such factors of Google's documentation users, and test whether the hypotheses in the previous section are supported by the developers' documentation usage data at scale.

Experience. To investigate the effect of experience levels in documentation usage, we measure the documentation users' experience level using two variables: *overall platform experience* and *specific product experience.* We define the experience with the platform as the user account age, i.e., years passed since signing the platform terms and conditions[2]. We define the experience with a specific product as the total number of successful API requests made to that API over the previous three months (February, March, and April 2020).

Product Type. To analyze the differences in documentation usage patterns, we recorded what each documentation usage data point was for.

Documentation Type Predisposition. As a proxy for one's possible predisposition for certain documentation types, we recorded the user's documentation *page views in the previous three months* (February, March, and April 2020).

Possible Intent. As a proxy for possible user intent when accessing the documentation, we recorded the *average per-page dwell time*, by dividing the total dwell time in May by the total number of documentation pages a user had visited.[3] We further grouped the data into three bins—less than 1 minute, between 1 minute and less than 10 minutes, and more than 10 minutes—to loosely correspond to the categories of intent (reminding, clarifying, and learning) identified by Brandt et al. [34]. The "more than 10 minutes" group most directly maps to a learning intent, while the other two groups possibly overlap with both reminding and clarifying.

Subsequent API Use. We mined the Google-collected API usage data (telemetry data) from June, July, and August 2020 corresponding to the subset of people in the aforementioned May-2020 documentation page-view log dataset, who also made subsequent API requests using the web-based services. This was possible because the pseudonymization strategy has random but persistent IDs that are consistent across documentation and API usage data. Specifically, we extracted the *number of successful API requests* made by each user (i.e., with 2XX return codes).

3.4.3 Sanity Test with Cluster Exploration

Before we formally test our hypotheses, we checked whether the hypotheses derived based on the general information-seeking literature apply at all to developers' documentation usage patterns observed in our data, by checking different clusters' user distributions. To help with our exploration, we first visualized each cluster's average dwell time, and discretized the numerical variables into four groups for each user factor, based on percentiles: `0|NA` (factor=0), `Low` (0-33%), `Medium` (34-66%), `High` (67-100%). Figure 3.4 shows the visualizations of the entire dataset, and three example clusters. We have included visualizations of other large clusters with over 500 users in our appendix[4] Using the visualizations, we selected clusters with different distributions for each factor we hypothesized would influence documentation usage. We then compared their documentation usage patterns to check if the factors we identified were related to variations in these patterns.

[2]For users who have never signed the terms and conditions for API usage, we assigned a value of 0.

[3]A more direct comparison to Brandt et al. [34] would require per-session dwell times, which we did not have access to, hence this approximation.

[4]To protect privacy (see section 3.2.3), we have not included visualizations of the remaining clusters. However, we note that these large clusters account for 77% of the total users in our dataset.

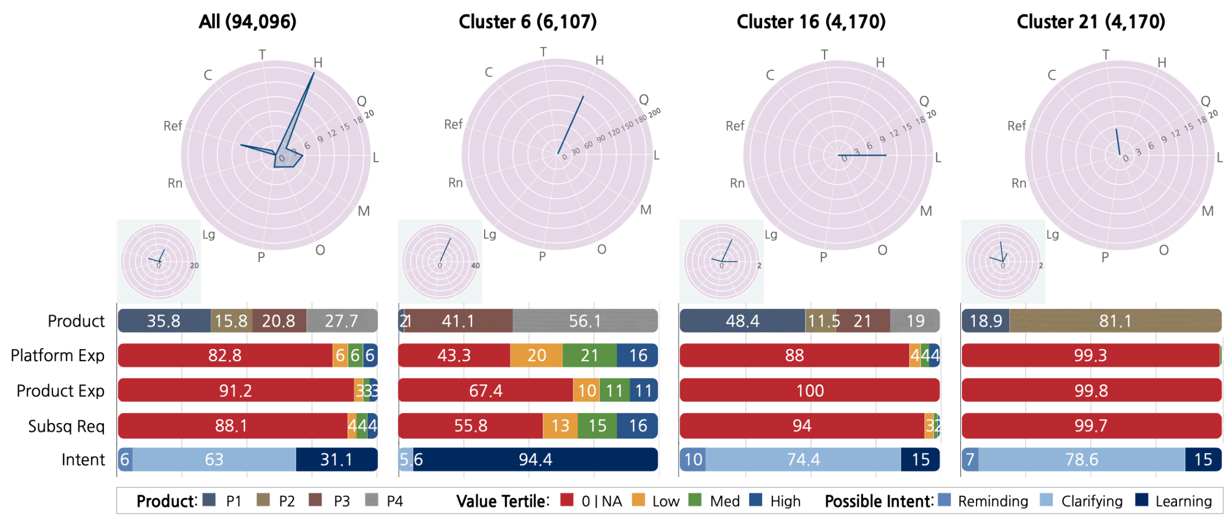


Figure 3.4: Highlights of the clustering analysis. Each polar plot displays the average time spent on each type of documentation (see Table 3.1 for the documentation type codes). The small polar plots show the average dwell time in the previous three months. Note that the ranges of the axes of the plots vary. Bar charts below the polar plots show the proportions (%) of each group in the cluster. For example, the charts of Cluster 21 can be interpreted as "In cluster 21, users without platform and product experience predominantly used Tutorial documentation (≈ 6 minutes) of P2 (81.1%) and P1 (18.9%), mostly for clarification purposes, without subsequent API requests."

H_1 (Experience): Comparing clusters with a lot of experienced users (e.g., Cluster 6, Cluster 26, Cluster 27) and clusters mostly with new users (e.g., Cluster 16, Cluster 21, Cluster 22), the dwell time of the latter was relatively shorter compared to the former. We also found that most of the clusters with more experienced users spend time on the documentation that describes lower-level details, such as Reference or How-to guide documentation, without needing to visit introductory documentation like landing or marketing pages. On the other hand, clusters with new users showed diverse documentation usage patterns, which might be because they browse the documentation while considering adopting the APIs while still being relatively unfamiliar with the products, instead of trying to learn to use the products.

H_2 (Product type): We observed that documentation usage for P1 and P2, on the one hand, and P3 and P4, on the other hand, is internally similar in different clusters — many users of the pairs ended up clustered together (e.g., Cluster 21 and Cluster 11 for P1 and P2, and Cluster 6 and Cluster 10 for P3 and P4). Clusters with a lot of P3 and P4 users visited How-to guide documentation, which might be due to their typical high project complexity requiring system-level configurations of multiple products in Google platform. In addition, we observed that clusters with the majority of users using application APIs show longer pricing documentation usage, whereas clusters of infrastructural API users show almost zero usage of pricing documentation. This could be explained by the usage context of the products: Infrastructural API users maintaining large software systems are also often employees of large corporations, with accounting and legal teams taking care of administrative tasks, removing the need to visit Pricing or Legal documentation, whereas application APIs are often used by smaller companies or individual projects whose developers are more likely to be responsible for administrative tasks.

H_3 (Documentation type predisposition): We observed a consistent trend where clusters of users who spent an extended amount of time on specific types of documentation in May also exhibited a prolonged engagement with the same documentation in previous months. For instance, consider Cluster 6 (task-oriented users), whose members demonstrated a substantial dwell time on How-to documentation in May; they also ranked second in terms of How-to documentation usage in previous months, among the clusters we analyzed. Similarly, Cluster 22 (financial users), which had the longest dwell time of Pricing documentation in May, consistently showed the longest dwell time for Pricing documentation in preceding months. Furthermore, even among clusters with lighter documentation usage, we noticed a parallel pattern: the dwell times from previous months mirrored the patterns observed in May.

H_4 (Possible intent): Comparing clusters with a lot of users with "reminding" or "clarifying" intention (e.g., Cluster 0, Cluster 16, Cluster 18) with clusters with mostly "learning" intention (e.g., Cluster 6, Cluster 26, Cluster 27), we observe that the former users spent much less time on the documentation pages on average, and also focused on documentation types like marketing and landing pages, which often provide an overview and administrative facts of the APIs, consistent with the "reminding" and "clarifying" intent reported by Brandt et al. [34]. In contrast, clusters with a lot of "learning" users visited documentation that provides more detailed guidance on how to use the products, like How-to documentation.

$\mathbf{H}_5$ (Subsequent API use): Comparing the clusters of users who made no or low subsequent API calls (e.g., Cluster 0, Cluster 16, Cluster 22) with the clusters of users who made subsequent API calls (e.g., Cluster 6, Cluster 26, Cluster 27), the latter had spent longer overall browsing documentation pages, and had spent most of their time on How-to guide and Reference pages as opposed to marketing pages, which could indicate that many had already decided to adopt the API. We also observed that the degree of such association may vary with the product. For example, compared to the users in Cluster 7 (documentation explorers) who visited Landing and Marketing documentation and had similar average dwell times, far more users in Cluster 16 (product explorers) actually made calls to the API in the subsequent months. This might be explained by their usage context: the product proportions were relatively equal in Cluster 16, but most of the Cluster 7 users visited only P1 documentation. Thus, we expect that users will need different types of information depending on their usage context, and thus the usefulness of documentation types may also vary.

3.4.4 Regression Analysis

Next we formally test the hypotheses above on our entire sample. First, we use multiple regression to test how much the various user-level characteristics we hypothesized about in $\mathbf{H}_1$-$\mathbf{H}_4$ can explain people's logged documentation visits to pages in each of our four genres (recall Table 3.1). Second, we test $\mathbf{H}_5$, i.e., to what extent developers' logged documentation visits in each of our four genres can explain their subsequent API use, again using multiple regression.

We start by estimating four logistic regression models, one for each documentation genre. In each model, the dependent variable is a boolean variable "*dwell time* $>$ 0" indicating whether or not a user in our sample accessed documentation pages of that particular genre. In addition, each model includes explanatory variables corresponding to $\mathbf{H}_1$ (overall platform experience, specific product experience), $\mathbf{H}_2$ (product), $\mathbf{H}_3$ (documentation use in the previous three months), and $\mathbf{H}_4$ (average page dwell time); see section 3.2.2 for definitions. All models include all variables. E.g., the model specification for the guide documentation genre is:

$$
\begin{aligned}
\log\left[\frac{P(\text{guide\_dwell\_time} > 0)}{1 - P(\text{guide\_dwell\_time} > 0)}\right] = \alpha + \\
&\beta_1(\text{overall\_platform\_experience}) + \\
&\beta_2(\text{average\_page\_dwell\_time}) + \\
&\beta_3(\text{product}) + \\
&\beta_4(\text{specific\_API\_experience} > 0) + \\
&\beta_5(\text{dev\_page\_views\_in\_the\_previous\_three\_months} > 0) + \\
&\beta_6(\text{guide\_page\_views\_in\_the\_previous\_three\_months} > 0) + \\
&\beta_7(\text{admin\_page\_views\_in\_the\_previous\_three\_months} > 0) + \\
&\beta_8(\text{meta\_page\_views\_in\_the\_previous\_three\_months} > 0)
\end{aligned}
$$

By jointly estimating the different β coefficients, this model allows us to estimate the strength of the association between each explanatory variable and the likelihood that users access documentation pages from each genre, *independently of the other variables included in the model*. Then, the p-value of, say, the estimated β_1 coefficient allows us to test $\mathbf{H}_1$, i.e., whether there is a correlation between platform experience and likelihood of accessing documentation genres being modeled. Similarly, we could test for correlations between platform experience and likelihood of accessing documentation pages from the other three genres with the other three models.[5]

To test $\mathbf{H}_5$ we use a similar strategy, estimating one logistic regression model with a boolean dependent variable "*subsequent requests* > 0". We restrict this analysis to the subset of users who have not made any API requests in the past months (more likely to be entirely new users), since we expect the results to be more actionable for this subset in terms of growing the API user base. We include all the same independent variables as before (the ones not directly tied to the hypotheses act as controls), except specific_product_experience which is by definition null for these users. We also include an interaction with product to test the effect of differences in products.

Overall, we took several steps to increase the robustness of our estimated regression results. First, we removed outliers (i.e., observations more than 3 standard deviations beyond the mean) for highly skewed count variables and applied log-transformations to improve heteroskedasticity. Second, we checked for multicollinearity using the Variation Influence Factor (VIF) and only kept variables having VIF lower than 2.5, following Johnston et al. [111]. Third, since we estimate multiple models, each with multiple variables, thus increasing the risk of Type I errors, we conservatively adjusted all p-values using Holm's correction procedure [99]. Furthermore, we only considered model coefficients worthy of discussion if the adjusted p-values were statistically significant at 0.01 level instead of the more common 0.05.

3.4.5 Results

Figure 3.5-top summarizes the documentation-access logistic regression results across the four models we estimated (one per genre) to test $\mathbf{H}_1$. We present our results in terms of odds ratios (OR) instead of regression coefficients to ease interpretation. All four models are plausible, with Nagelkerke [171] pseudo R^2 values (deviance explained) of 74% for dev, 16% for admin, 44% for guide, and 55% for the meta documentation genre. , Figure 3.5-bottom summarizes the subsequent-usage logistic regression model testing $\mathbf{H}_5$. The relatively high explanatory power of the models indicates that at least some of the patterns of documentation usage align with user characteristics and API usage behaviors. Figures 3.6 and 3.7 show the consistent results for complementary count-based, linear regression models that further investigate the time spent on the different pages. Here, we focus our discussion around the logistic regression results.

[5]Note that our research hypotheses in section 3.4.1 are not all equally broad, i.e., they don't all cover all documentation genres or even the same documentation genres. Our choice to model each documentation genre separately is flexible enough to allow us to draw conclusions about all hypotheses, including the broader ones, by qualitatively comparing results from the relevant models. For example, we can reason about a particular estimated coefficient β being statistically significant in multiple models corresponding to multiple documentation genres.

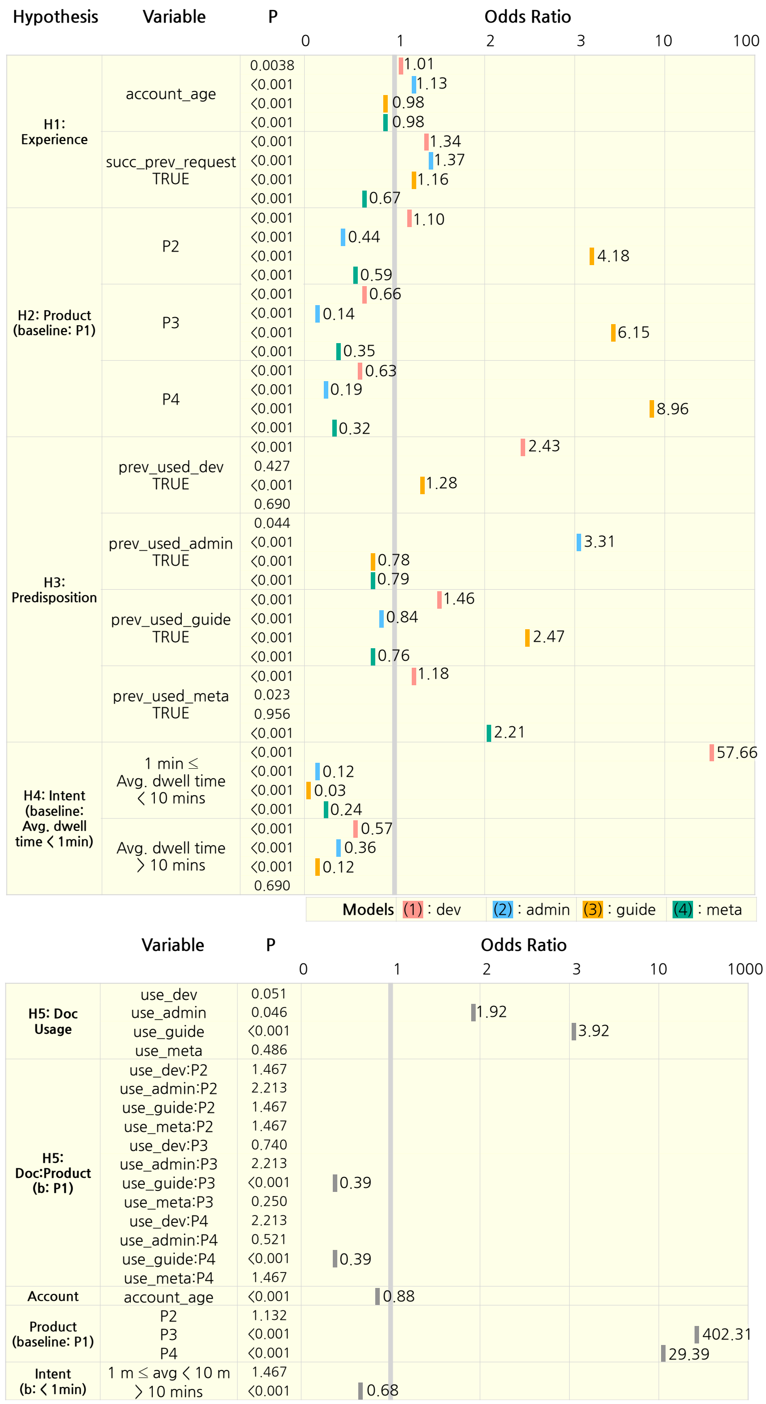


Figure 3.5: *Top*: Estimated odds ratios from the regression modeling *dwell time* > 0 for our four documentation genres. For example, the odds of accessing Dev type documentation (pink) are 1.01 times higher among users with one extra year of platform experience. *Bottom*: Estimated odds ratios from the regression modeling *subsequent requests* > 0. Variables without statistically significant coefficients (adjusted $p \geq 0.01$) are omitted.

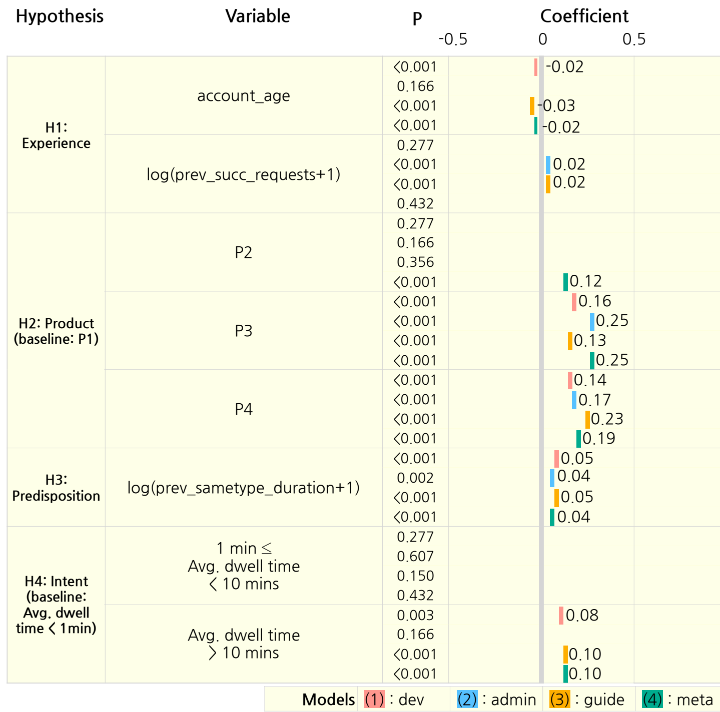


Figure 3.6: Coefficients from regression analysis predicting dwell time for four types of documentation. p-values are adjusted based on the Holm's correction [99]. Coefficients are removed for non-significant results (p>.001).

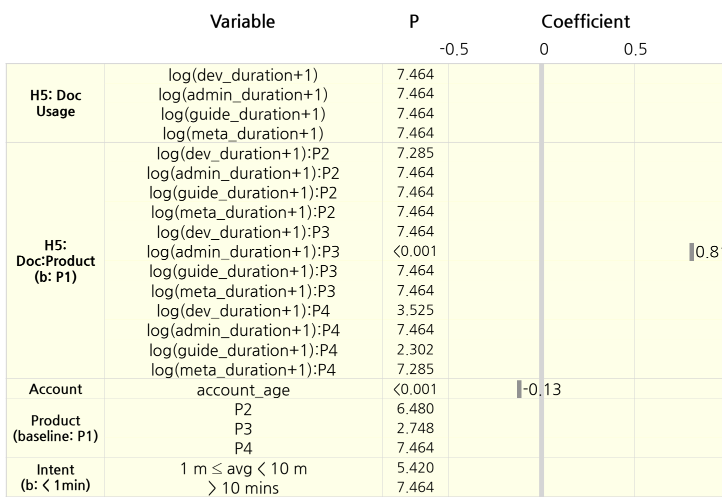


Figure 3.7: Coefficients from regression analysis predicting dwell time for four types of documentation. p-values are adjusted based on the Holm's correction [99]. Coefficients are removed for non-significant results (p>.001).

H_1 (Experience): supported. Results from the dev and meta-genre models are consistent with the hypothesis. For example, the odds of accessing reference documentation and other dev pages are 1.34 times higher among people with prior experience with the products (product experience), i.e., those who made successful API requests in the past, compared to those without, and the odds of accessing such pages are 1.01 times as high among users with one extra year of platform experience. Similarly, the odds of accessing marketing and other meta documentation are lower (OR = 0.67) among people with prior experience with the products (product experience), and the odds of accessing such pages are 0.98 times as high among users with one extra year of platform experience.

Interestingly, the results from the admin-genre model align more with the documentation genres covering implementation details than meta: the odds of accessing pricing, legal, and other admin documentation are also higher (1.37 times) among people with prior experience with the products compared to those without. This could indicate that the information in admin documentation is not only needed once, when people make API adoption decisions, but rather is consistently needed throughout their use of the API.

H_2 (Product type): supported. All four models support the hypothesis: taking P1 as the reference, the magnitude of differences between P1 and P2 is consistently smaller than either P1 and P3 or P1 and P4; i.e., the documentation page visits of large-scale infrastructural products tends to differ starkly from that of application products. Taking the dev-genre model as an example, the odds of accessing the documentation pages are only 1.1 times higher among visitors to P2 documentation compared to P1, but 0.66 and 0.63 times as high among visitors to P3 and P4 compared to P1.

H_3 (Documentation type predisposition): supported. All models show strong effects of documentation type consistency: there are correlations between the past and the future access to some types of documentation. For instance, in the admin-genre model the odds of accessing admin-genre documentation pages are 3.31 times higher among people who had also accessed such pages in the past three months compared to people who had not. As many different pages of documentation are included in each type of documentation, and the analysis is done at a month-level as described in section 3.2.4, this result does not provide conclusive evidence of the users' preference for the contents or structure of documentation pages. However, it still suggests that the documentation users have types of documentation they are more familiar with, and can access them repeatedly.

H_4 (Possible intent): only partially supported. The results for this hypothesis are mixed. On the one hand, the dev-genre model reveals a clear difference between people with long and short average per-page dwell times, as hypothesized: the odds of accessing reference documentation and other dev pages are 0.57 times lower among people with average dwell times greater than 10 minutes compared to those with average dwell times less than a minute. The model also reveals that the odds of accessing dev-genre documentation are greatest (57 times higher) among people with average dwell times between one and 10 minutes. Similarly, the models for admin- and meta-genre pages, which include marketing and pricing, are generally supporting the hypothesis.

In contrast, the model for guide-genre documentation points to the opposite finding than hypothesized when comparing to people with average dwell times less than a minute (the group with the shortest dwell times, set as the baseline in our models): the odds of accessing tutorials, how-to documentation, and the like are lower, not higher, among both people with average dwell times between one and 10 minutes as well as people with average dwell times greater than 10 minutes, compared to those with average dwell times less than a minute.

One potential explanation is that many users might use the guide documentation as a cheat-sheet, from where they copy and paste various API boilerplate [174] or usage examples. Although guide documentation was originally intended to introduce and explain products to relatively inexperienced users, it appears to be widely used by users with diverse intentions.

3.5 Discussion

We investigated the feasibility of using documentation page-view logs to inform the design of documentation. Through a series of hypotheses derived from the literature, contextualized by an exploratory analysis of our page-view log data (§3.3), and subsequently validated through a large-scale regression analysis (§3.4), we discovered that there are multiple discernible patterns of documentation use, even when the documentation pertains to the same platform, or even the same products.

3.5.1 Feasibility of Log Analysis for Documentation Review

Large-scale log analysis helps discover unexpected use. As large-scale log analysis allows analyzing all documentation usage, with less researcher efforts and costs, we could explore diverse documentation usage patterns. For example, in addition to users mainly using documentation for API learning, which was often studied in the existing literature that used smaller-scale qualitative approaches [66, 117, 158, 159], the clustering analysis discovered additional large clusters of users who only check pricing documentation (Cluster 22: financial users), or that many users make many API requests without even visiting reference documentation (Cluster 27: task-oriented users). The cluster exploration and the hypotheses testing also revealed that expected documentation usage can differ from actual use. For example, although how-to documentation is often regarded as introductory for new users [66], we observed that users with more product experience made more visits to the guide documentation (Cluster 6: task-oriented users) than those with less product experience, which was also confirmed by the regression analyses ($\mathbf{H}_1$). While the cause or intent behind these unexpected uses cannot be found solely with log analysis, our observations might be useful in designing more focused human studies. Moreover, we believe that a similar analysis can be used to answer broader research questions like "How does documentation usage change over time as users develop their expertise with the products?", or "What are the strategies developers use for information seeking in documentation?"

Page-view log analysis is informative but could be further refined. The analysis could be extended to also account for the structure and content of the documentation pages, in addition to the factors we considered. For example, although the top web search results given the query `Google [product]` were marketing documentation for all four products, the second result varied between a guide documentation page for P4, and landing pages for P1, P2, and P3. Thus, in interpreting the differences in documentation usage between products, whether intended or not, differences between the documentation structure and external resources should also be taken into account. Analyzing *referrer* pages, i.e., the pages accessed by a user prior to loading a particular web page, might be useful in understanding how such differences affect the documentation use [195]. We propose this direction for future research.

In practice, the analysis plan can be adapted based on the analysis goals. In this work, we employed a mixed-method approach to gain a comprehensive understanding of Google documentation usage. This involved both exploring the data and validating our hypotheses. Each of these analyses complements the other, offering distinct advantages and considerations. For example, clustering analysis proves valuable in uncovering common and unexpected usage patterns, requiring less quantitative data analysis expertise to get started. However, it is important to note that interpreting clustering results can be subjective, and conducting a detailed investigation of every cluster may not always be practical. Subsequently, performing regression analysis adds a layer of confidence to our findings, providing a comprehensive overview of the dataset. In practice, it may not always be necessary or feasible to conduct both types of analysis due to differences in skill requirements. In such cases, the choice between the two can be made based on the specific goals of the log analysis. For instance, a user experience (UX) researcher seeking a lightweight usability review might opt for a quick cluster analysis and interpretation as demonstrated in Section 3.4.3. If stronger evidence is needed to support hypotheses, especially for design refactoring, engaging a quantitative UX researcher or data scientist to perform regression analysis following a clustering study would be a more suitable approach.

3.5.2 Recommendations for Documentation Providers

Through the log analysis, we found that documentation usage can vary based on the users' experience in product and platform ($\mathbf{H}_1$), the type of product described ($\mathbf{H}_2$), and many other factors ($\mathbf{H}_3$, $\mathbf{H}_4$). This suggests that established knowledge on documentation usage may not always be generalizable to all target users. Here, we highlight some of the specific implications for how to design improved documentation catering to users with different characteristics.

Explicitly mention the target audience of documentation.
Previous studies [158] found that developers often experience difficulty in determining which documentation type to select when searching for a particular piece of information. We posit that this is because documentation for different products adheres to varying documentation standards and categorizes information differently, and it takes time for developers to learn these distinctions. Since we confirmed that developers' documentation visits are correlated with their characteristics, we posit that explicitly indicating the intended audience of the documentation will assist them in selecting the appropriate types

and pages of documentation to access (i.e., provide strong "scent" in the information foraging theory [199]).

Duplicate important information for information discovery. As our models show (**H_3**), users are more likely to visit types of documentation that they have accessed in the past. Although it is often considered to be better to *modularize* the documentation, this can be problematic if important information is only presented on a specific page, as the user might not always discover that [100, 177]. This observation is consistent with the finding of Meng et al. [158] that developers often skip sections in the software documentation based on their problem solving strategies. Thus, to reduce the risk of developers missing important information, we recommend providing such information in multiple types of documentation, or at least providing prominent functional links to the page providing such information.

Provide product-specific starting points. We discovered that there are variations in visit patterns among products with distinct characteristics. This is expected because different types of information are provided and needed depending on the purpose or domain of the product (**H_2**). For instance, for infrastructural products such as P3 and P4, many users (Cluster 2: task-oriented users) accessed how-to guides providing instructions for the configuration settings, but for application products like P1 and P2, many users visited tutorial documentation pages providing walkthroughs for a simple use case (Cluster 21: product explorers) that aid new users in quickly familiarizing themselves with the products. However, for users who are new to the products with little understanding of them, it will be challenging to know what documentation type or page will be the best starting point [118], especially because there are a plethora of documentation pages per product. Thus, to help the new users quickly grasp the gist of the products, we recommend providing product-specific recommendations about which documentation pages to use to start learning, as similarly recommended in Jeong et al. [108]. Most commonly accessed documentation pages or pages that correlate with subsequent API requests, which can be acquired from the page-view logs, will be good candidates for the recommendation, as they were already proven to be useful for other users. We note that we do not recommend changing the documentation templates or navigation structures, because inconsistent inter-product information organization can hinder information foraging of users, especially those who use multiple products from Google. A designated space for the product-specific documentation recommendation in a landing page or a navigation tab will allow users to know where to look if they become lost.

Nudge new product users to visit guide-genre documentation. When developers select third-party libraries, the quality of documentation is perceived as a good sign of the library's quality [262], and when a user is not able to find appropriate learning resources, it becomes a major obstacle in getting to know the libraries resulting in user frustration [206]. Our results suggest that guide-genre documentation is particularly effective in influencing the decision to adopt a product (**H_5**), although one might think that landing documentation is beneficial for them since it provides an overview of the products. We believe that guide-genre documentation is helpful in making the adoption decision, as it describes what the products offer and help developers gauge what they need to do for onboarding, which corresponds to what new users look for from documentation [206]. Thus, although other documentation pages will be useful in the end, nudging developers to visit guide-genre documentation as early as possible may help them perceive the quality of documentation positively, and adopt the API.

3.5.3 Longer-term Vision: Personalization

While we distilled actionable recommendations for how to adjust the design of software documentation taking into account many dimensions of user characteristics that might affect their usage, doing this manually may be unrealistic when many products are involved. Instead, we argue that the time is ripe for approaches to *automatically personalize* the documentation. Personalization is not a new topic and has already proven to be effective for other services like media streaming and search engines [249]. Prior research on general web search has also made significant progress in designing effective personalized recommender systems to increase the long-term engagement of users [277], using both implicit (e.g., dwell time [277, 278]) and explicit (e.g., item rating [11, 12]) feedback mined from historical interaction data as an indicator of users' interests and needs. As the dwell time mined from documentation page-view logs can capture some user characteristics, in addition to the interaction histories that page-view logs contain by design, we expect that personalizing approaches can also be used in the documentation domain. Here, we present three directions to improve developers' information foraging on documentation using page-view logs.

Documentation recommendation. First, we argue that it is time to go from static approaches of documentation recommendation (for example, consider the omnipresent navigation links like "Recommended content" or "What' s next" or "Next topic," that typically point to the same target page regardless of which user is browsing) to dynamic ones that take user characteristics into account to provide more relevant suggestions. An ideal scenario is perhaps one where the recommender system has access to the developer' s code repository or profile, that reveal the developers' needs and background that are known to correlate with their documentation usage (e.g., their product and platform experience), as we discovered from the analysis. Short of that, we show that some signals about user-level characteristics are present in much more modest and more widely-available log data on previous documentation page visits. A recommender system could learn to profile users based on previous page visits (similar to our clustering) and, given that knowledge, suggest the next documentation pages to visit from among those that users in the same cluster have visited or interacted with before.

Within-documentation search. Personalization can also be applied to within-documentation search engines. Many previous studies of within-documentation search engines showed the need for efficient navigation [108]. Typically, software documentation contains information for both novices and experts, sometimes implicitly within a single page, other times explicitly across dedicated separate pages. For example, a difference between a 'basic' and an 'advanced' tutorial could be that the advanced tutorial describes APIs with more flexible capabilities, which require additional parameters. One way to personalize is query modification [228], by expanding the user query using additional terms inferred from user profiles. As above, the user profiles can be approximated from documentation page view logs; for example, when a user' s documentation page view pattern is similar to Cluster 6 (task-oriented users), with high levels of guide documentation visits that correlate with product experience level, the system can infer that the user is experienced. Then, given a search query "how to set up P1," the system could augment the query along the lines "how to set up P1 *advanced user*," which should bias the search results towards the dedicated advanced pages.

Documentation filtering. Another idea is that a "smart" documentation system could automatically filter what information is being shown depending on the user. For example, when a user has already accessed platform-common information (e.g., authentication) from other products, the system can hide/fold such parts for new APIs the user is reading about, to make information foraging more efficient. Similarly, one could imagine hiding/folding other parts of a documentation page, such as the code examples, for users that prefer to develop a more conceptual understanding first [158]. These examples both require data on historical accesses of other documentation pages by the same users (or by users in the same cluster), which is often included in the page-view logs.

3.6 Summary

Through the log analysis, we discovered discernible patterns of documentation usage, showing that documentation users can have diverse information needs. By testing our hypotheses derived from the clustering analysis, we confirmed that users' information needs can vary based on the type of product described ($\mathbf{H}_2$), the user backgrounds ($\mathbf{H}_1$), and many other factors ($\mathbf{H}_3$, $\mathbf{H}_4$). This suggests that considering different users' characteristics may lead to more effective information support for developers.

Chapter 4

Automatic Extraction of Boilerplate Client Code[6]

When programmers are new to a particular API, they often lack the experience to differentiate between essential, unique code and the repetitive, standard code snippets that make up boilerplate, because they yet possess the nuanced understanding required to differentiate the repetitive, standard snippets of boilerplate code from the critical, unique aspects of their implementation. Without the ability to easily identify and segregate boilerplate, new programmers can become bogged down in trying to decipher and adjust non-essential parts of the codebase, diverting attention from focusing on innovative and critical development tasks. In this chapter, we explore the feasibility of automatically identifying boilerplate code form client code, so that (1) API providers identify this API usability issue easily, and (2) the new programmers to quickly grasp the core functionalities of an API after learning about the boilerplate code patterns of APIs they learn.

4.1 Introduction

Almost all modern software programs adopt and use a large number of APIs. Therefore, dimensions of API usability, including learnability, effectiveness of use, and error-proneness, are increasingly becoming significant concerns for API designers [32, 165, 207]. To investigate API usability issues and to improve APIs, researchers have used several methods such as lab studies [239] and API design reviews [70, 153]. The understanding gained from such studies, along with the insights from experienced API designers, have led to the development of guidelines for API designs and heuristics for evaluating APIs [32, 165, 167, 207]. However, despite these efforts, many APIs are still difficult to use [168]. In particular, API designers have reported that anticipating how developers will use their API in the wild is difficult and leads to usability challenges when developers use the API in unexpected ways [167]. API designers have also reported significant trouble discovering what are the usability barriers at scale [167]. Although online sources such as Stack Overflow and GitHub may contain ample amounts of real client code or insights into how program-

[6]This chapter is adapted from Nam *et al.* [174]

```java
import org.w3c.dom.*;
import java.io.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;

// DOM code to write an XML document to a specified output stream.
private static final void writeDoc(Document doc, OutputStream out) throws IOException{
    try {
        Transformer t = TransformerFactory.newInstance().newTransformer();
        t.setOutputProperty(OutputKeys.DOCTYPE_SYSTEM, doc.getDoctype().getSystemId());
        t.transform(new DOMSource(doc), new StreamResult(out));
    }
    catch(TransformerException e) {
        throw new AssertionError(e);  // Can't happen!
    }
}
```

Listing 4.1: Writing an XML document to a specified output stream in Java may involve significant boilerplate code for initialization and error handling [32].

mers perceive APIs, designers report that there are not so many automated approaches to mine usability data from these repositories at scale, nor to gauge the severity of the usability issues [167].

In contrast, mining software repositories techniques have long been used to identify API usage patterns [214]. For example, existing API usage pattern mining tools such as ExampleCheck [84, 281] and PAM [77] automatically identify API methods that are frequently called together in client code. Primarily, these tools have been designed to help users learn a new API, by identifying idiomatic usage examples, as well as to help API designers gain insights into how their APIs are being used. We argue that API usage pattern mining tools may also help reveal certain API usability issues.

Specifically, we focus on one particular grievance that developers express repeatedly [54, 191, 225, 265] in online discussions about APIs (and programming languages more generally): *boilerplate code*. Wikipedia [51] refers to boilerplate as "sections of code that have to be included in many places with little or no alteration", and code "the programmer must write a lot of to do minimal jobs." One Stack Overflow user [191] calls boilerplate "any seemingly repetitive code that shows up again and again in order to get some result that seems like it ought to be much simpler"; most users agree that boilerplate is tiresome to write and error-prone [54, 225, 265]. Listing 4.1 shows a typical example: Whenever one wants to write an XML document to a specified output stream in Java, which is a common usage scenario, this requires significant boilerplate code. One could imagine that this use case could be accomplished natively by calling a single API method such as `writeXML`.

From an API designer's perspective, *the existence of boilerplate code may serve as an indicator of poor API usability.* This is because the need for boilerplate code often indicates that the API does not directly provide the methods that programmers need, so the extra code is needed to do even common tasks. Another cause may be that the API designers assume users will need the flexibility to put things together in multiple ways, but most users do not, so everyone uses the same collection of methods in the same way [253]. Users may also use boilerplate code even though there are already implemented API methods that can succinctly perform the task, which indicates discoverability problems [253].

However, despite general consensus on the undesirability of having to write boilerplate code, as well as API design guidelines explicitly mentioning boilerplate as an anti-pattern [32, 165, 207], the concept remains largely undefined and understudied. We start by reviewing boilerplate code examples and definitions from multiple sources (section 4.2). Through qualitative analysis, we confirm that boilerplate involves sections of code that have to be written repetitively to accomplish common and otherwise simple tasks that users largely do not want to think about. Moreover, we find that the main reasons for boilerplate code are underlying language and API limitations. We also find that developers and API contributors make efforts to reduce the amount of boilerplate code by introducing new helper functions and abstractions.

Next, we present MARBLE (Mining API Repositories for Boilerplate Lessening Effort), an automated technique for identifying instances of boilerplate API client code. Since a key property of boilerplate is that it is repetitive, we designed MARBLE on top of an existing API usage pattern mining approach, specifically PAM [77], which is automated and can be run at scale. However, not all idiomatic API usage patterns that an approach like PAM extracts, of which there are typically many, should be considered boilerplate. Therefore, we developed novel filters using AST comparison and graph partitioning (section 4.3) to identify, among the frequent API usage patterns, those which are most likely to involve boilerplate. By reducing the number of false positives, API designers could then focus manual review on the most likely candidates. The source code of MARBLE is available online [173].

We evaluated MARBLE on 13 Java APIs, for which we mined around 10,000 client code files from GitHub open-source projects, with 768 client code files per library on average after random sampling. Our results (Section 4.4) show that not only does MARBLE return a sufficiently short list of boilerplate candidates for manual review to be feasible, but also that more than half of these candidates are considered boilerplate by two experienced Java programmers, where one of them is an API designer at a large software company. To further the discussion about what boilerplate is and how it impacts APIs, we discuss some of the boilerplate candidates and suggest potential API design improvements. The full list of boilerplate candidates mined is available online [173].

Note that we are *not* claiming that all boilerplate code is bad, or that boilerplate code should always be eliminated. In fact, some of the patterns we identified as boilerplate are important to leave as-is to achieve other code quality requirements, such as increased readability or separation of concerns. However, as has been proposed elsewhere [168], we argue that these kinds of API design decisions are best made with full knowledge of the tradeoffs. We argue that MARBLE provides data which may be used in practice by API designers as a basis for such discussions. We also recognize that our method, like any other data-mining approach, is only applicable after an API has sufficient client code using it, and is therefore complementary to lab studies and API design reviews.

In summary, we contribute: i) the boilerplate API code mining problem; ii) properties which can be used to identify boilerplate code; iii) an automatic boilerplate code mining algorithm; iv) an empirical evaluation on 13 Java libraries.

4.2 Studying Boilerplate Code

As far as we have been able to find, studies of boilerplate code, or studies that even mention boilerplate code, are scarce (exceptions include [32, 112, 122, 207]). Mostly we have found it to be "I-know-it-when-I-see-it," with the existing explanations being vague and abstract, rather than deterministic.

At the same time, although boilerplate code is regarded as something that programmers want to avoid [54, 225, 265], and API design guidelines suggest that API designers should reduce the need for boilerplate code [32, 165, 207], we still have not seen any studies of whether some boilerplate code is induced by APIs, and if so, whether it is possible to reduce it at the API level.

Thus, to help understand the characteristics of boilerplate code, and the impact of API design on the need for boilerplate in client code, we first investigated three research questions:

- RQ 4.1: What is a good definition and what are common properties of boilerplate code?
- RQ 4.2: What are reasons for needing boilerplate code?
- RQ 4.3: How do API users and API authors deal with boilerplate code?

4.2.1 Resources

We reviewed the literature, surveyed our social media contacts, and reviewed Stack Overflow questions and GitHub commits. Mainly, we looked for boilerplate code examples, but when available, we also collected the rationale behind the boilerplate designation, reasons for the boilerplate, and how programmers dealt with boilerplate. We looked for Java boilerplate code examples involving at least one API call. We chose Java because the API usage pattern mining technique we build on (section 4.3) was tested for Java. In some communities (e.g., web developers), boilerplate code is used as a synonym for template code [91], but we exclude this context as we are looking for boilerplate related to API usability.

Literature

We searched for definitions or explanations of boilerplate code in Google Scholar [32, 112, 207], blog posts, online discussion boards (e.g., reddit) and Wikipedia. When available, we also collected boilerplate code examples.

Survey

We asked our Twitter contacts to share boilerplate code examples and the reasons behind the boilerplate designation. Overall, 8 participants submitted 1 to 3 boilerplate examples each and all provided the reason why they thought each example qualifies as boilerplate.

Stack Overflow

I identified five popular Java API tags (`android`, `swing`, `jdbc`, `spring-mvc`, `jsp`) in Stack Overflow and manually collected questions asking about how to reduce boilerplate code, using Stack Overflow search queries (e.g., "[Swing] boilerplate"). We checked the first page (15 questions) of the results for each Java API tag, and collected boilerplate code examples and the reasons why the questioner thought it was boilerplate.

GitHub Commits

We identified and cloned the top 10,000 most starred Java repositories from GitHub, using the March 2018 version of GHTorrent (details in section 4.4). Then, we identified all commits including the keyword "boilerplate" in the commit message. Finally, two researchers manually coded all the matching commits.

4.2.2 Definition of Boilerplate Code

We investigated the available definitions of boilerplate code from the literature, and iteratively discussed the boilerplate examples among the research team (which includes an experienced API designer in a large software company, who is often involved in large software projects using APIs), distilling common properties. We did not use GitHub commit data in this analysis because (1) it does not explicitly express the characteristics of boilerplate, and (2) it does not indicate the exact location of boilerplate code in many of the code changes.

Undesirable

Commonly, boilerplate code is identified using subjective properties, sometimes explicitly: "It's a subjective definition" [191]. Mostly, such properties have *negative connotations*. One source calls it "uninteresting, unchanging, repetitive, and/or tedious" [238]. Another common but subjective property is that boilerplate code is needed even for simple functionality. The highest voted answer from Stack Overflow defines it as "it ought to be much simpler" [191]. We summarize all of these properties as being "undesirable."

High frequency

Most of the definitions and explanations require that boilerplate code occurs frequently in client code, such as "shows up again and again" [191], or "code that has to be included in many places" [51]. Frequency is a particularly intuitive property given the negative connotation of boilerplate code: indeed, if it were rare, its impact would likely be reduced. The high frequency property also implies that boilerplate API code examples should be found among idiomatic API code examples, as the latter are by definition frequent, hence our choice to base our approach on an existing API usage pattern mining tool [77].

Localized

The statements constituting boilerplate code are usually *closely located near each other*, rather than spread over multiple methods or files. All examples from Stack Overflow and the survey, and three examples from Google Scholar [32, 112, 207] were parts of a single method. The Wikipedia example of getters and setters within a class [51] is the only one not limited to a single method.

Little structural variation

The boilerplate code instances appear in *similar form without significant variation.* Many sources describe that it "gets copied and pasted" [207], and is used "with little or no alteration" [51]. We also found that many explanations of boilerplate code describe the examples as "I find myself writing the same ugly boilerplate code" [248], or "a lot of code that must be duplicated" [19].

This corresponds to the definition of code clones, especially "templating clones" [112]. However, while code clones need not occur with high frequency to be considered clones, boilerplate should occur frequently (Property 2).

4.2.3 Understanding Cause of Boilerplate Code

Method

Two researchers performed closed coding for all of the boilerplate related commits we collected from GitHub. As one property of boilerplate code (*RQ*4.1 Property 4) corresponds to a subcategory of code clones, we borrowed Roy and Cordy's "reasons for cloning" [220] as our starting set of codes: development strategy (reuse approach, programming approach), maintenance benefits (avoiding, ensuring, reflecting), overcoming underlying limitations (language limitations, programmers' limitations), and cloning by accident (protocols to interact with APIs and libraries, programmers' working style). As there were commits referring to different types of boilerplate (e.g., boilerplate license), we also coded the commits with types of boilerplate: boilerplate, client (i.e., reduce the boilerplate code using the API), comment (i.e., boilerplate in the comments such as license, javadoc), and Non-Java. Each commit was assigned one type and one reason based on its commit message and code diffs. Two researchers started coding collaboratively and, after 10 agreements, each separately coded half of the data. In total, we randomly sampled and coded 120 commits, and the two coders reached 87.5% agreement (Jaccard Index) both for boilerplate types and for reasons, on 20% of the data.

Results

Among 120 commits, 40 of them were commits to reduce the use of boilerplate code. We found that the predominant reason for needing boilerplate was overcoming underlying language limitations (mentioned in 19 commits). Examples of this include needing to initialize many getters/setters and verbose error handling in Java. 11 were induced in order to interact with APIs, for example, tagged as "Protocols to interact with APIs". Some of the boilerplate code was due to questionable API designs (e.g., requiring the client to cast the output by providing an abstract object), but some seemed inevitable due to the design patterns or apparent trade-offs in the design of the APIs. For example, an API adopting a builder pattern usually involves a lot of boilerplate to set properties of an object. Another 10 were due to programmers using the API inefficiently, such as using an API call which is not ideal that requires more code.

Since most boilerplate code instances are by-products of language and API limitations, analyzing boilerplate code can help review their API designs and find usability issues. Despite some of the limitations being unavoidable, such as error handling in Java, there are many other situations where API designers can reduce the need for boilerplate, such as by adopting annotation libraries or introducing helper functions. Boilerplate code due to programmers using the API inefficiently may be a signal that there are discoverability issues, so the documentation and tutorials might need to be improved to overcome the conceptional gap between the API designers and API users.

Note that while we were able to code every boilerplate instance with codes from "reasons for code cloning" [220], which indicates that boilerplate can be considered a type of code clone, the two are not identical, as clearly not all code clones can be considered boilerplate under our definition (high frequency, localized, API related). Therefore, while our approach to automatically mine boilerplate candidates (Section 4.3) starts from an existing API usage pattern miner, future work could also consider boilerplate mining approaches that start from code clone detectors, but exploring this goes beyond the scope of the current work.

4.2.4 Programmers' Efforts to Reduce Boilerplate Code

Method

To answer *RQ*4.3, two researchers performed descriptive coding on the changes that were made to reduce the amount of boilerplate code – either boilerplate within the source code itself, or boilerplate that is needed by the client to use the library. We used the same 40 boilerplate instances found in commits from *RQ*4.2 (i.e., attempts to reduce the boilerplate code by editing the project code), and 6 commits that changed the API itself (i.e., attempts to reduce the boilerplate code using the API). We coded based on the code diffs and commit messages, and extracted the means used to reduce boilerplate code.

Results

The majority of boilerplate code reductions within a project were made by introducing new helper functions or classes, either by writing new ones, or by including a function or class from an external API. For the simple Java-specific boilerplate such as getters/setters, some used annotations (e.g., Project Lombok [149]) or injection to reduce the amount of boilerplate. When changing the API to reduce the client-side boilerplate, programmers added more processing into the library, thereby reducing the need for pre/post processing for the input/output of API calls. Some made the interfaces more specific to reduce the need for parsing or casting in the client code. Also, like within-project boilerplate reduction, some commits added a set of helper functions or new classes to allow users to have a more specific but simpler interface, which can usually be done without making breaking changes to the API.

4.3 Mining Boilerplate Code

Using the results from the previous section (Section 4.2.2), we seek to find code instances that contain calls to a target API and satisfy the properties of boilerplate code we identified: (1) are undesirable, (2) occur frequently in client code, (3) occur within a relatively condensed area, and (4) are used in similar forms without significant variations.

To this end, we designed MARBLE, which combines an API usage pattern mining technique with a graph partitioning algorithm to identify candidate boilerplate code from software repositories. MARBLE consists of several steps, depicted in Figure 4.1 and described below. In summary, we first identify a large set of API usage patterns which represents our initial set of boilerplate candidates. We then filter out any patterns that are spread over multiple methods, or which have many variants, to finally provide a short list of boilerplate candidates that satisfy all of the properties above, except for Property 1 (undesirable). These candidates could then sorted and contextualized with the real-world client code, and delivered to the API designers so they can review the candidates for Property 1. We intentionally designed the process in this order because testing Property 4 (little structural variation) is computationally expensive. By filtering out the candidates that do not satisfy the other properties, we are able to reduce the number of AST comparisons (section 4.3.3).

4.3.1 API Usage Pattern Mining

We start from an existing API usage pattern mining technique, to collect boilerplate candidates containing one or multiple target API calls and satisfying the high frequency property. Specifically, we chose PAM (Probabilistic API Miner) [77], a state-of-the-art parameter-free probabilistic approach as of 2020, which is fully automated and available open-source. In their evaluation, Fowkes *et al.* [77] found that PAM returns less redundant and less numerous results compared to other API usage pattern mining algorithms we compared.

The PAM Core

PAM uses a *probability model over API call sequences* to identify “interesting” sequences of API calls / API usage patterns. Given a target API, the model can be trained unsupervised on a corpus containing code from open-source GitHub repositories. Concretely, PAM first parses each source file and extracts the sequence of target API calls within each method (only Java code is currently supported), using a depth-first traversal of the abstract syntax tree (AST). At the same time, frequency information for each API call over methods is

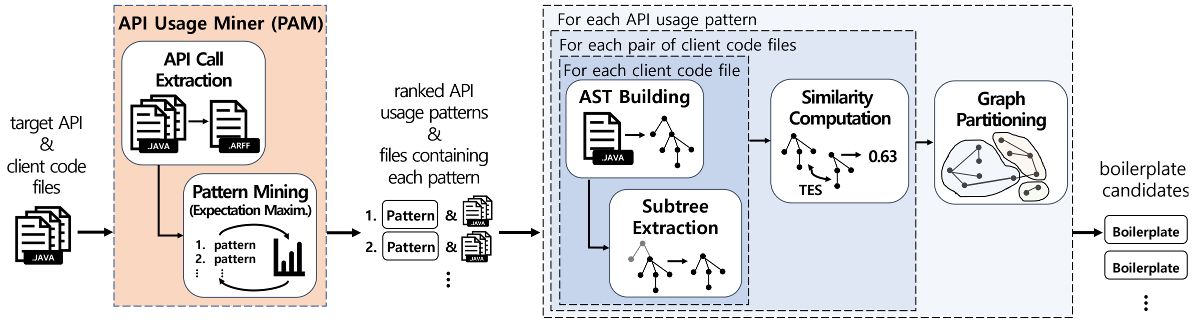


Figure 4.1: Overview of our mining process and the steps involved.

recorded. For example, given `javax.xml.transform` as a target API and Listing 4.1 as one client method using it, PAM's API call extractor returns

- `javax.xml.transform.TransformerFactory.newInstance`
- `javax.xml.transform.TransformerFactory`
  `.newTransformer`
- `javax.xml.transform.Transformer`
  `.setOutputProperty`
- `javax.xml.transform.dom.DOMSource.<init>`
- `javax.xml.transform.stream.StreamResult.<init>`
- `javax.xml.transform.Transformer.transform`

Then, PAM uses expectation-maximization (EM) [164] to iteratively infer "interesting" API usage patterns (i.e., sequences of API calls) and learn the probability model. That is, an API call sequence `[A, B]` is "interesting" if the two API calls `A` and `B` occur together more often than expected by chance, given the individual frequencies of `A` and `B`. The EM algorithm iteratively interleaves API call sequences, and searches for the set of patterns that maximizes the probability that the model assigns to all client methods in the input dataset. For more details we refer to the original paper by Fowkes *et al.* [77].

PAM returns a ranked list of API usage patterns $P = [P_1, P_2, \ldots, P_n]$, where $P_i = [c_1, c_2, \ldots, c_m]$, is an API call sequence. For example, when we run PAM on `javax.xml.transform`, it returns:

- $P_1 =$ `[javax.xml.transform.dom.DOMSource.<init>,`
  `javax.xml.transform.stream.StreamResult.<init>]`
- $P_2 =$ `[javax.xml.transform.TransformerFactory.`
  `newTransformer,`
  `javax.xml.transform.Transformer.transform]`
- $P_3 =$ `[javax.xml.transform.dom.DOMSource.<init>]`
- $P_4 = \ldots$

Modifications to the PAM Core

As our main goal is to help API designers identify the patterns that are likely to reflect API usability issues, the list of candidates to be considered must be relatively short, since such reasoning requires designers' manual effort. We modified the base PAM algorithm to reduce the number of false positive boilerplate patterns returned. This step involved setting two thresholds empirically, which we did after reviewing a sample of PAM results: First, if there is a pair of patterns such that one fully contains the other (e.g., P_1 and P_3 above), we remove the sub-pattern (P_3) unless the number of occurrences is more than 50% different from its super-pattern's, to avoid reporting multiple small variations of one boilerplate candidate. For example, if there is a sequence `[a, b, c]` which occurred 100 times among the client code files and another sequence `[a, b]` occurred 120 times, we keep `[a, b, c]` and ignore `[a, b]` because `a`, `b`, and `c` are mostly used all together. However if `[a, b]` occurred 500 times, we do not ignore it because it is likely that there are other uses not involving `c`. Second, to avoid reporting rare and project-specific boilerplate code candidates, we also ignore patterns that occurred in less than 5% of client code methods for a given API.

Limitations

The returned API usage patterns are sequences of API calls, without any structural information. This ensures that the returned patterns are robust to variations in local context, e.g., conditionals, loops, exception handling, *etc.*, which is desirable when the goal is mining generic API usage examples. However, this is at odds with our third boilerplate requirement that the call sequence should appear in similar form without significant variation.

Another limitation of PAM for boilerplate mining is that the order of API calls matters. When a boilerplate instance involves multiple API calls that can be used in any order, such as `getHeight()` and `getWidth()`, PAM would consider `[getHeight(), getWidth()]` and `[getWidth(), getHeight()]` to be different sequences, and the "interestingness" of this API usage would be lower than it should be.

Finally, PAM was originally designed to capture the usage patterns of a single library, whereas API usage patterns or boilerplate can have multiple libraries involved.

We address these limitations in the following steps, by also considering the context around the "interesting" API call sequences.

4.3.2 AST Extraction

To decide whether an "interesting" API usage pattern involves boilerplate, we should also consider the (structural) context around the API calls. For example, if other methods (e.g., built-in language APIs) are always used around or between the target API calls, or if the sequence of target API calls is always used inside a certain loop construct, we should also consider this context as part of the candidate boilerplate instance. Therefore, to determine this context, given a list of API usage patterns and a list of client code files containing instances of those patterns $[(P_1, F_1), (P_2, F_2), \ldots, (P_n, F_n)]$, respectively, for each P_i we extract and post-process the ASTs of the files in F_i. Moreover, since we are only interested in code that occurs in local areas (Property 3 above), i.e., the areas around the target API calls, we restrict this analysis to individual methods and split the file-level ASTs (which correspond to entire classes) into method-level subtrees.

Still, the method-level AST subtrees may contain nodes unrelated to target API calls and the candidate boilerplate pattern. To narrow down the relevant parts of the method-level AST subtrees S, we use a simple slicing heuristic: we extract the smallest sub-subtrees of each subtree S, which completely encompass the target API call pattern. For example, given an AST of the client code in Listing 4.1 (Figure 4.2) and an API usage pattern $P =$ `[javax.xml.transform.dom.DOMSource.<init>, javax.xml.transform.stream.StreamResult.<init>]`, we extract the first common ancestor of the `DOMSource` and `StreamResult` nodes, i.e., the subtree rooted at `Method Invocation`.

For patterns with a single API call, the smallest subtree is the same as the API call, which means we do not acquire further contextual information. Therefore, we modify the smallest subtree heuristic for these patterns, and find the smallest `if/loop/try` subtree containing the API call. However, this heuristic may not always extract a smaller subtree than the entire method (e.g., if no `if/loop/try` is used in the method). Based on the third property that statements constituting boilerplate code are closely located near each other, we applied another heuristic: when the subtree has over 20 method invocations, trim the sub-subtrees that are far from the sub-subtrees containing the API calls. We chose the threshold 20 informed by the examples collected from the qualitative study in section 4.2.

If an API call of a pattern occurred multiple times in a client file, there might be multiple potential subtrees. In this case, we use the smallest one, following Property 3: the further the calls are apart, the less probable it is that they form a single pattern. In the case that the full smallest pattern occurs multiple times in a client code file, we keep multiple subtrees.

4.3.3 Graph Partitioning

As the third step in our approach, we check Property 4: whether the API call sequence is used consistently in similar contexts (i.e., structures) throughout the client code.

To capture this property, we devise an approach to 1) compute the similarity between all pairs of subtrees containing the API call sequence contexts; and 2) cluster together similar subtrees. Intuitively, if there are many clusters with low similarity, this indicates that there are many different ways a sequence of API calls is being used at the code level, suggesting that the pattern is less likely to be a part of boilerplate, as per Property 4. In contrast, if there is a cluster having a number of subtrees, and the similarity between them is high, the cluster (i.e., specific use case) can be a boilerplate candidate.

Pairwise Similarity

Given a list of subtrees for each client file in F_i containing a same API pattern P, we compare every pair of subtree lists from $\langle f_i, f_j \rangle$ in F_i, and calculate the similarity between them. We use AP-TED (All Path Tree Edit Distance) [192] as our distance/inverse similarity measure, since it is memory efficient and fast. Other tree differencing algorithms such as GumTree [69] could be applied as well.

To calculate the AP-TED, we visit each subtree in pre-order, collecting the types of each node (e.g., MethodDeclaration or IfStatement). To avoid noise from lexical details, such as variable

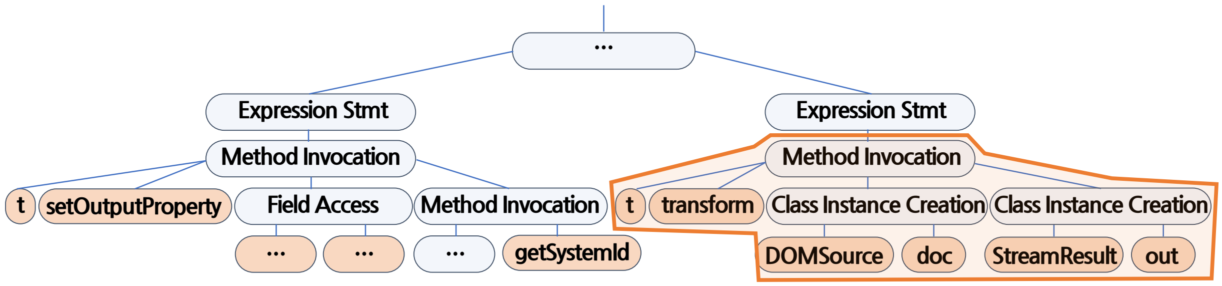


Figure 4.2: A part of the AST for the code in Listing 4.1 and the extracted subtree (colored) for the API usage pattern [DOMSource.<init>, StreamResult.<init>] using our slicing heuristic.

names, we only collect nodes for: loops (e.g., ForStatement), error handling (e.g., TryStatement), conditions (e.g., IfStatement), casts, and method invocation types. For the MethodInvocation nodes, we also collect the names (e.g., `newInstance`) to compare different API calls used in the boilerplate candidate. To overcome PAM's limitation that it only considers the usage patterns of one target API, we also collect the names of method invocations that are not from the target API. By doing this, even though PAM is not able to capture external API calls as part of a sequence, our approach can still use them to calculate the similarity, and the external calls will be seen in the boilerplate candidate if they frequently occur together with the target library's API calls. It also helps mitigate PAM's other limitation—not capturing a set of API calls into a usage pattern unless they occur in the same order.

To calculate the similarity using the tree edit distance, we invented *TES*, the Tree Edit Similarity. When s is a list of subtrees in f, each of which encompass the target API, given two lists of subtrees $\langle s_1, s_2 \rangle$ for $\langle f_1, f_2 \rangle$, we define *TES* as:

$$TES(s_1, s_2) = max\left(\frac{1}{e^{AP\_TED(s_{1i}, s_{2j}) \cdot 0.1}}\right) \tag{4.1}$$

In the case that the client code file uses the pattern multiple times so there are multiple subtrees for s_i, we calculate the distance between every pair of subtrees, and use the maximum value for the next step.

Clustering

With these pairwise similarity values, we build a weighted graph for each pattern, in which nodes n_i are client code files, and edges $n_i \rightarrow n_j$ are weighted by $TES(n_i, n_j)$. We then cluster the nodes in this graph, to capture the different contexts (structures) in which an API call sequence is being used. As the similarity values between client code files are computed based on the ASTs, boilerplate candidates that are structurally similar would likely be clustered together. On the other hand, even if the same API calls are used in two client files, if their structures are significantly different, or external API calls around the pattern are significantly different (which the edge weight captures), they would likely be clustered separately. Therefore, after graph partitioning, the clusters would indicate structurally-different usage patterns given a sequence of target API calls.

For clustering we use the Louvain community detection algorithm [33], a heuristic method based on modularity optimization: Given a weighted graph of n nodes, Louvain first assigns a different cluster to each node. Then, for each node n_i, Louvain calculates the gain in modularity by removing n_i from its cluster and placing it into its neighboring n_j's cluster. If the gain is positive, and maximum among the gains from other neighbor nodes, the algorithm removes n_i's cluster, and merges n_i into n_j. The process repeats and is applied until there is no further improvement in modularity. Secondly, Louvain adjusts the weights of the edges. The weights of the edges between the new clusters are the sum of the weights of the edges between nodes in the corresponding two clusters. The algorithm keeps iterating the first phase, merging clusters, and second phase, adjusting weights, until a fixed point.

We applied this technique for three main reasons: (i) unlike most other graph clustering algorithms for which the number of clusters should be given as input, Louvain determines it as part of the algorithm; (ii) its computation time is short; (iii) it was originally designed for large networks (e.g., 118 million nodes [33]), hence we expect it to scale up well.

Additional Filtering

The clustering algorithm does not guarantee that within a cluster the client code instances of the API usage pattern are *all* highly similar among each other, i.e., that they all represent the same boilerplate candidate instance. To further prune spurious clusters which may increase noise in the results, we require that the average pairwise *within cluster similarity* is greater than a threshold. Empirically, we observed that patterns involving many API calls would have more variance in the subtrees and thus the similarity would be lower than short usage patterns, even though qualitatively they would appear similar; therefore, when the pattern is longer, the similarity threshold should be lower. We set this threshold for average TES (Equation 4.1) to be $1/e^{2(x+1)\cdot 0.1}$, where x is the number of API calls in the sequence. For example, when a usage pattern contains only one API call, the threshold is $1/e^{(2\cdot 1+2)\cdot 0.1} = 0.67$, i.e., we discard clusters with average within-cluster similarity below 0.67.

4.3.4 Viewer

Since boilerplate code has the subjective properties discussed in section 4.2 that cannot be automatically tested, manual review of the candidates is necessary. To help API designers efficiently review them, we implemented a viewer for the boilerplate candidates. Based on the intuition that more verbose boilerplate candidates should be reviewed first by API designers, MARBLE's viewer ranks all the boilerplate candidates by their length, and for each one, displays usage examples from three representative client code files from different GitHub projects.

4.4 Evaluation

In this section, we evaluate the accuracy and potential usefulness of our mining algorithm, by answering:

- RQ4.4 (Validation): How well does MARBLE identify known boilerplate examples?
- RQ4.5 (Precision): How many of the boilerplate candidates found by MARBLE would human experts agree with?
- RQ4.6 (Practicality): Does MARBLE return a reasonably short list of boilerplate candidates for manual review?

Table 4.1: Summary statistics on the number of candidate boilerplate instances for the APIs in our dataset.

API	API Patterns	Boilerplate Candidates	Found Known Boilerplate	Precision	Client Files	Client w/ BP	Avg. Len.
`android.app.ProgressDialog`	134	12	True	0.92	641	296	5.96
`android.database.sqlite`	508	7	True	0.57	796	96	7.49
`android.support.v4.app.ActivityCompat`	26	5	True	0.60	486	93	8.01
`android.view.View`	940	11	True	0.36	1,051	100	9.16
`com.squareup.picasso`	79	0	False	-	565	-	-
`java.beans.PropertyChangeSupport`	32	8	True	0.38	604	48	6.04
`java.beans.PropertyChangeEvent`	32	5	True[7]	0.00	749	-	-
`java.io.BufferedReader`	39	3	True	0.67	998	343	3.52
`java.sql.DriverManager`	30	0	False	-	744	-	-
`javax.swing.JFrame`	185	0	False	-	791	-	-
`javax.swing.SwingUtilities`	71	2	False	0.50	800	14	6.64
`javax.xml.parsers`	196	3	True	1.00	893	39	11.23
`javax.xml.transform`	325	3	True	0.67	871	45	9.2
Total	2,597	59	0.69	0.56	9,989	1,074	6.06

[1] During the evaluation, we came to the conclusion that the known boilerplate instance for `java.beans.PropertyChangeEvent` from Stack Overflow is not a strong boilerplate code. Therefore, even though MARBLE mined the similar pattern from the client code (Found Known Boilerplate: True), none of the candidates from this library is labeled as boilerplate code (Precision: 0.00).

- RQ4.7 (Usefulness): Does MARBLE identify informative boilerplate candidates that could help review an API?

RQ4.4 is to test whether MARBLE finds the 13 boilerplate examples corresponding to 13 Java APIs we collected from the literature, survey, and Stack Overflow in Section 4.2.

RQ4.5 evaluates MARBLE's false positive rate. Quality assurance tools, such as defect prediction [119] or static analysis [110], should generate few false warnings to be usable in practice.

RQ4.6 evaluates MARBLE's practicality. Reviewing boilerplate candidates and investigating potential usability issues require manual effort from the API designers, as labelling something as boilerplate is ultimately a judgement call. On the same set of 13 APIs for which we collected 13 known boilerplate examples, we evaluate whether MARBLE returns a sufficiently short list of candidates in the right order, so that it may be usable in practice. Specifically, we test to what extent our filtering steps involving AST comparison and graph partitioning (discussed above) will help to substantially reduce and rank the list of candidates for manual review compared to the baseline PAM [77].

RQ4.7 is to qualitatively evaluate MARBLE's usefulness. Two researchers manually reviewed all the mined boilerplate candidates for the same 13 APIs, and analyzed to what extent the candidates signal places where the APIs might be improved.

4.4.1 Experimental Setup

Dataset

To collect API client code, we identified and cloned the top 10,000 most starred Java repositories from GitHub, using the March 2018 version of GHTorrent [87], excluding forks and repositories marked as deleted. We then mined the Java source files importing the APIs, by matching import statements (e.g.,`import javax.xml.transform`). For APIs with more than 800 client code files, to reduce the runtime of our experiments, we sampled files randomly, ensuring 95% confidence level and 3% margin of error. Table 4.1 gives an overview of our dataset.

Each row in the table is a separate API on which we ran MARBLE, one API per each of the 13 known boilerplate examples. The "API Patterns" column shows the number of API usage patterns returned by just running PAM, and the "Boilerplate Candidates" column shows the number of boilerplate candidates that MARBLE identifies. The "Found Known Boilerplate" column shows whether MARBLE found the known examples. The "Precision" column shows the percentage of boilerplate candidates that the two researchers labeled as actual boilerplate among the number of the candidates that our approach retrieved for each API. The "Client Files" column shows the number of client code files that were used for boilerplate mining. The "Files w/ BP" column shows the number of client code files that involve at least one boilerplate instance. The "Avg. Len." column shows the average length of the boilerplate candidates. The precision on the "Total" row is the micro-average precision; that is, the average precision after aggregating the data of all libraries.

Implementation Details

For the *API usage pattern mining* part (Section 4.3.1), we adjusted the output generation part of PAM's public implementation [155], without modifying the core algorithm. We ran PAM using default parameter settings: 10,000 iterations with a priority queue size limit of 100,000 candidates.

For the *AST comparison* (Section 4.3.2), we wrote a Java program to parse and traverse ASTs, and extract subtrees with our heuristics. To compare the subtrees, we used the `apted` library [57], but customized the cost model to weigh insertion, deletion, and rename operation equally. For the *graph partitioning* (Section 4.3.3), we wrote a Python program to build a graph, preform graph partitioning using the NetworkX package [244], and filter spurious clusters using our heuristics. The final boilerplate candidate viewer generator for the API designers is implement in Python, and generates a html page for each API.

4.4.2 Results and Discussion

RQ 4.4 (Validation): How well does MARBLE identify known boilerplate examples?

We ran MARBLE on all 13 APIs represented in the discovered boilerplate examples (Table 4.1), and manually compared the returned candidates to the known examples. Among the 13 known boilerplate examples, MARBLE could identify 9 (69%).

Three out of four false negatives were not caught by MARBLE because they incorporate a variety of real-code (i.e., non-boilerplate code) inside them, like `invokeLater` in `javax.SwingUtilities`. Although the boilerplate wrapping the real-code was repetitive and the same for every usage, the non-boilerplate part varied widely among the client code files, which lowered the similarity between the client code AST subtress containing this pattern. This could be improved in the future by applying program analysis to more accurately slice the API-call-related and unrelated parts of the code.

Another false negative was a builder pattern, which MARBLE did not identify because client code files used different combinations of setter calls. A Stack Overflow user [260] complained about this because the same builder was needed multiple times within the project, which does not necessarily mean that other programmers use it in the same way in other projects. Since MARBLE's goal is more general, to inform API designers about potential boilerplate in a wide range of client code, this example was not exactly in scope. However, MARBLE could be extended to also identify these within-project boilerplate examples if API designers feel the need, by adding the within-project pattern frequency to the algorithm.

We conclude that MARBLE is valid.

RQ 4.5 (Precision): How many of the boilerplate candidates found by MARBLE would human experts agree with?

MARBLE returned 59 boilerplate candidates overall for the 13 APIs in our sample (table 4.1). To compute MARBLE's precision, two researchers labelled each candidate as potentially boilerplate or not. We first separately labelled all of the candidates (77% inter-rater agreement), then discussed disagreements until reaching consensus, finally updating the labels. The main criteria for the boilerplate designation were whether it potentially reduces the API usability, and whether it could be further abstracted.

As a limitation, note that we labelled some candidates as false positives even though they resemble boilerplate (verbose and seemingly abstractable), because we could not confirm that these could significantly lower the API usability without looking at the number of occurrences within a single file or a single project, which goes beyond the scope of this work. As discussed above, we designed MARBLE to identify boilerplate candidates across a large number of client

```java
if (ActivityCompat.shouldShowRequestPermissionRationale(this,
        Manifest.permission.READ_EXTERNAL_STORAGE)) {
} else {
    ActivityCompat.requestPermissions(this,
            new String[]{Manifest.permission.READ_EXTERNAL_STORAGE},
            MY_PERMISSIONS_REQUEST_READ_EXTERNAL_STORAGE);
}
```

Listing 4.2: Boilerplate code instance of Android ActivityCompat client codes.

code files / projects to help API designers focus on boilerplate that might affect more users. However, while we were reviewing the candidates, we found that within-project boilerplate might also impact API usability. For example, as a programmer, it could be annoying if three lines of an API sequence need to be duplicated across many methods within a project, even though those three lines might be rarely used in other projects.

Overall, MARBLE's precision is 56%: the two annotators agreed that 33 out of 59 candidates could be considered boilerplate. We conclude that MARBLE has acceptable precision.

RQ 4.6 (Practicality): Does MARBLE return a reasonably short list of boilerplate candidates for manual review?

Comparing the MARBLE results to PAM's, the API sequence miner our approach is built on (section 4.3.1), we observe that MARBLE significantly reduces the number of resulting instances by applying further filtering on the PAM output using AST comparison and graph partitioning. The reduction is from a mean of 200 usage patterns per API with PAM down to a mean of 4.5 boilerplate candidates with MARBLE (median down from 79 to 3); the largest reduction is for `android.view.View`), from 940 down to 11 (Table 4.1).

We also measured the time it takes to label the boilerplate candidates, to roughly estimate the time needed for a designer's manual review. One researcher, who is an expert API designer, took less than 3 minutes per boilerplate candidate, and when I did it, it took around 5 minutes per candidate. Since we did not have enough experience with some of the APIs, it took more time to read the documentations and use cases. However, for API designers, we believe that it would take less time to review the boilerplate candidates, and find potential usability issues.

We conclude that MARBLE returns a sufficiently short list of candidates for manual review to be feasible.

RQ 4.7 (Usefulness): Does MARBLE identify informative boilerplate candidates that could help a designer review an API?

We manually reviewed all 59 boilerplate candidates identified (full list available online [173]), looking for causes and potential improvements.

Given the space constraints, we discuss only three boilerplate candidates returned by MARBLE.

Android ActivityCompat. Listing 4.2 shows potential boilerplate involving Android's ActivityCompat, with two API calls: `ActivityCompat.shouldShowRequestPermissionRationale` and `ActivityCompat.requestPermissions`.

```
@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion, int currentVersion) {
    Log.w(TAG, "Upgrading_test_database_from_version_" +
            oldVersion + "_to_" + currentVersion +
            ",␣which_will_destroy_all_old_data");
    db.execSQL("DROP_TABLE_IF_EXISTS_data");
    onCreate(db);
}
```

Listing 4.3: Boilerplate code instance of Android Database SQLite client codes.

This boilerplate is to ask for a certain permission from a user, but also provides an explanation if the user has already denied the permission request from this app. `shouldShowRequestPermissionRationale` returns true if the user has previously denied the request, but did not select the "Don't ask again" option in the permission request dialog; or false if the app has never asked a permission, a device policy prohibits it, or the user has selected the option.

MARBLE identified that out of 486 files importing ActivityCompat, this pattern is used in the same format in 36.

One potential redesign for better usability in this case is to abstract this into the API by adding a simpler method which handles permission checking and request rationale internally: if the permission is not granted, it checks if the permission request has been already denied or not, requests the permission with or without explanation, and sends the results to the client.

However, we hypothesize that there could be a design rationale behind the current design, e.g., possibly to improve the privacy of the users of Android applications. Although our proposed abstraction could help new Android developers get started with the API, lead to less code, and be less error prone in these common cases, this could also give an impression that providing rationale on permission requests to the Android application users is not critical.

We argue that this trade off is only valid when users understand the API designer's rationale, or at least that the rationale actually plays out as expected. If most users just copy and paste this boilerplate code without much thought, this design decision could reduce API usability without any benefits. Therefore, alerting API designers to situations where their design decisions result in boilerplate may help them review to what extent their design rationale is valid.

Android Database SQLite is an open-source relational database library in Android. Listing 4.3 shows a boilerplate code instance to upgrade a database by dropping tables and creating a new one, which requires using `SQLiteDatabase.execSQL` and `SQLiteOpenHelper.onCreate`.

Although `OnUpgrade` was intended to provide flexibility for users, MARBLE found that 40 client code files out of 68 overrode it in the same way, similar to Listing 4.3, by dropping tables using `execSQL` and creating new ones with `onCreate`. To mitigate this boilerplate, as discussed in section 4.2, API designers could make the common usage, such as logging the update, dropping the table, and recreating the database with a new version, as the default functionality of `onUpgrade`. This would allow users to write less code in general, and also give them some flexibility if needed.

Another way to reduce this type of boilerplate is to use annotation libraries (e.g., the Spring framework [36], or Object Relational Mapping libraries), which offer an object-oriented interface to the relational database. Annotations and ORM tools reduce the need for simple CRUD (Create, Read, Update, and Delete) boilerplate, and many libraries have adopted them (e.g., Neo4j-OGM). In fact, while reviewing the client code using this boilerplate, we observed that 10 of the client code files have adopted GreenDAO [40], which is an ORM tool for Android. The fact that many

```java
if (videoControlsView != null) {
    this.seekBar = (SeekBar) this.videoControlsView.findViewById(R.id.vcv_seekbar);
    this.imgfullscreen = (ImageButton) this.videoControlsView.findViewById(R.id.vcv_img_fullscreen);
    this.imgplay = (ImageButton) this.videoControlsView.findViewById(R.id.vcv_img_play);
    this.textTotal = (TextView) this.videoControlsView.findViewById(R.id.vcv_txt_total);
    this.textElapsed = (TextView) this.videoControlsView.findViewById(R.id.vcv_txt_elapsed);
}
```

Listing 4.4: Boilerplate code instance of Android View client codes.

clients adopt a certain helper function or a tool can be a signal for API designers to update their API similarly, or recommend these tools to their clients for better usability. This boilerplate also shows that seeing the common patterns of use, and whether they could be abstracted by an annotation framework, might be useful for the helper library designers as well, in understanding the needs of users, and developing a new library.

Android View. Notably, as the APIs we used for evaluation are popular and actively maintained, in some cases we could actually find the improvements that had already been made by the API designers. The boilerplate candidate was still detectable in our dataset because the clients had yet to upgrade to the newer version of the API.

Listing 4.4 shows a classic boilerplate code for Android View. These 5 lines of code are to find the views from the XML layout resource file with the given IDs. The pattern consists of multiple uses of one API call: `android.view.View.findViewById`. We could observe that 217 files out of 518 files use this method at least three times in a row. It is already verbose since developers need to call this method multiple times, but even worse because null checking and typecasting are also needed.

A straightforward way to reduce this verboseness is to make the return type of `findViewById` to a generic `T`, to eliminate the need for manual typecasting. In fact, Android changed the method's definition from `View findViewById(int id)` to `T findViewById (int id)` starting with Android 8.0 [86]. This shows that 1) API designers care about the boilerplate code instances which reduce the API usability, and 2) informing API designers about the boilerplate candidates can actually lead to usability improvements.

Like the previous boilerplate candidate, another way to reduce this type of boilerplate is to use annotation libraries. There are several libraries providing annotation supports for this boilerplate (e.g.,`@BindView` of ButterKnife [266]), by helping users easily map the view ID declared in an XML layout file with the Java variable.

4.4.3 Threats to Validity

Our approach may be biased by the small number of APIs we tested it on. However, the boilerplate examples cover various domains and design patterns, and we believe that the properties we identified will generalize.

Note also that we only used 13 externally-known boilerplate examples to extract boilerplate properties and as a validation set to empirically choose the different thresholds involved. Still, MARBLE was able to discover many previously unreported boilerplate examples, which reduces the threat of overfitting.

We only evaluated our algorithm with popular Java APIs which have hundreds to tens of thousands of client files, but it is possible that the usefulness or performance of our algorithm varies for other libraries, which are relatively new or less popular. Also, as we analyzed the boilerplate instances with a single API designer and a Ph.D. student, others may disagree that our tool identifies boilerplate that is worth looking at.

4.5 Summary

In this chapter, we devised MARBLE, a new boilerplate mining algorithm based on four properties of boilerplate code that we identified from many sources (undesirable, high frequency, locality, and limited structural variation). Through an evaluation with 13 Java APIs, we demonstrated that it is feasible to automatically extract boilerplate code, which has been considered to be hard to extract, with an approach designed with a thorough understanding of the characteristics of the information.

Chapter 5

Information Support for Programming with Unfamiliar Libraries[8]

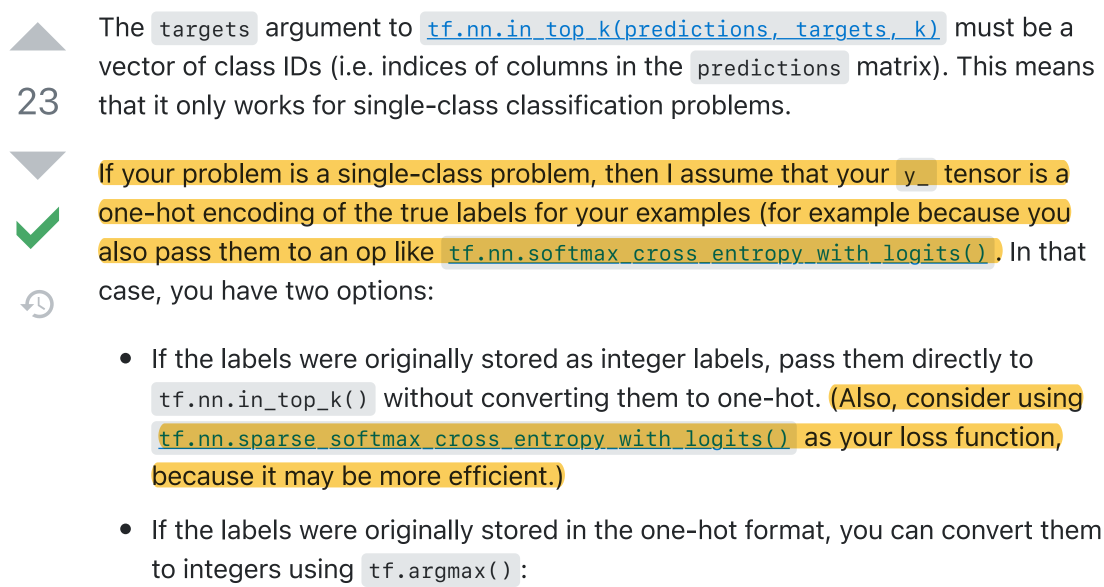
In this chapter, we conducted a study to assess the benefits of context-aware support through a human study. As a first user context dimension to explore, we chose the task type, and designed an information support tool for developers writing code using unfamiliar libraries. Among many challenges in obtaining the necessary information for learning unfamiliar library, one key challenge is: finding the appropriate API types and methods needed for a particular task (i.e., *discoverability* [63, 212, 239]). We observed that the traditional pull-based information support, where users request information when they are aware of the API methods they need, is not effective in addressing the discoverability issue. To address this, we proposed a push-based information support approach that proactively presents information to developers within their workflow. Our aim was to provide users with comparable API methods in their search engines, enabling them to discover diverse ways to utilize an API and access relevant execution facts. We hypothesized that such information will help developers not only discover more of an API, but also understand the API better, such that they can make more informed decisions about which methods are applicable to their task or which are preferable given alternatives.

However, comparable API methods are not often available in the reference documentation, so extracting information on comparable API methods is challenging and not readily available on demand. To tackle this challenge, we employed a learning-based information extraction approach as our proposed solution for overcoming incomplete information. To evaluate its effectiveness, we compared the performance of the learning-based information extraction approach with pattern-matching-based and heuristic-based approaches.

5.1 Introduction

New libraries and frameworks constantly emerge, each with their own Application Programming Interfaces (APIs), so developers must frequently learn new APIs to stay competitive. Indeed,

[8]This chapter is adapted from Nam *et al.* [177]


23

The `targets` argument to `tf.nn.in_top_k(predictions, targets, k)` must be a vector of class IDs (i.e. indices of columns in the `predictions` matrix). This means that it only works for single-class classification problems.

If your problem is a single-class problem, then I assume that your `y_` tensor is a one-hot encoding of the true labels for your examples (for example because you also pass them to an op like `tf.nn.softmax_cross_entropy_with_logits()`. In that case, you have two options:

- If the labels were originally stored as integer labels, pass them directly to `tf.nn.in_top_k()` without converting them to one-hot. (Also, consider using `tf.nn.sparse_softmax_cross_entropy_with_logits()` as your loss function, because it may be more efficient.)
- If the labels were originally stored in the one-hot format, you can convert them to integers using `tf.argmax()`:

Figure 5.1: Fragment from a Stack Overflow answer by mrry / CC BY-SA 3.0 illustrating the tacit crowd knowledge on comparable API methods softmax_cross_entropy_with_logits and sparse_softmax_cross_entropy_with_logits. We highlighted the sentences supporting the comparison.

researchers have concluded that "being able to learn new technical skills is likely more important [to a software engineer] than any individual technical skills" [137].

Learning a new API requires many types of knowledge [245] and there are many documented challenges to obtaining these [212]. Among them, two key challenges are: finding the appropriate API types and methods needed for a particular task (i.e., *discoverability* [63, 212, 239]), and identifying all the relevant information about their execution behavior, usage patterns, side effects, alternatives, and many others [146, 174, 180, 209, 281].

Lacking such knowledge can affect developers in at least two important ways. First, by its very nature [227], software engineering work involves careful consideration of design alternatives and often tradeoffs between competing objectives. This happens at all levels, including when choosing, within a given library or framework, the most appropriate API types and methods to use while balancing complexity, runtime performance, memory consumption, but also readability, understandability, amount of boilerplate involved, *etc.*, as there are typically multiple ways to implement the same functionality [158]. Thus, one can expect that the harder it is for developers to access the relevant information to make informed decisions, the less appropriate their choices would be. Second, although modern APIs are often large and sophisticated, many of their types and methods are rarely used in practice, in part due to insufficient discoverability [100]. In addition to affecting the aforementioned decisions, this can also slow down software developers, and, in the long term, lead to missed opportunities for developers to take advantage of the most effective technological solutions.

In acquiring API knowledge, developers tend to consult a variety of sources, as no single one is typically complete. These include the official reference documentation [132], but also crowd-based knowledge sharing websites like Stack Overflow (SO), which have long been recognized as usefully complementary [104, 107, 142, 170, 190, 251]. In particular, Stack Overflow discussions are a good source of knowledge on API method comparisons, which developers ask about quite frequently [146]. Such comparisons can expose developers to more diverse ways to use an API, thereby improving discoverability, and can also surface more relevant execution facts about the API, thus improving learning and programming using the API. Yet, this information on comparable API methods is generally not easy to extract and not available on demand, despite its value.

Sometimes, Stack Overflow discussions start with the author asking explicitly for a comparison between API methods, e.g., "What is the main difference between StringBuffer and StringBuilder?" Prior research using natural language processing (NLP) techniques can identify such posts with high accuracy when the language is well structured and amenable to detection using syntactic patterns [146, 148, 264]. However, comparisons between API methods on Stack Overflow are also often tacit—they arise organically as part of answers, without the original poster asking for them explicitly, and the relevant sentences for the comparison are not clearly identifiable or even grouped together, but rather can be scattered throughout a post. Consider the example in Figure 5.1, where the author of the answer offers relevant information about the relationship between two methods from the popular machine learning TensorFlow API, softmax_cross_entropy_with_logits and sparse_softmax_cross_entropy_with_logits, unprompted by the question. It is sometimes difficult even for humans to recognize that the answer compares the two methods, and then to extract the most relevant sentences informing the comparison (indeed, we ourselves had difficulty with this; see Section 5.2). Therefore, it is clearly challenging to design a pattern-matching-based classifier to do the same, since in addition to recognizing which answers include such API method comparisons, one would also need to extract and encode generalizable syntactic patterns for training.

Instead, we show that (semi-)supervised, deep-learning based methods developed in the NLP knowledge extraction community [181] can learn to recognize API method comparisons *directly from labeled data*, bypassing the need to manually identify elaborate syntactic patterns. Moreover, we show that such models can be trained efficiently, with reasonable effort for data labeling, thus taking the next step towards the goal of automatically extracting knowledge from unstructured Stack Overflow posts.

Our work consists of three main parts. First, we develop an **annotation protocol** and use it to compile a **dataset** of 266 pairs of comparable API methods identified in a statistically representative sample of Stack Overflow answer posts discussing the TensorFlow API. As part of our annotation effort, we also label the sentences within a post that are most relevant in support of the comparison.

Second, we design and run a **human subjects study with 16 participants completing a series of tasks in two experimental conditions**, *viz.,* with and without access to a custom-built browser plugin that augments the TensorFlow API reference documentation with the information on comparable API methods from our labeled dataset. We analyze both quantitative and qualitative data from the human study, showing, among other things, the extent to which having access to such information can help with understanding the API design space given task requirements, and what requirements participants have for such a tool.

Third, informed by our annotation effort and human subjects study, we develop and evaluate **SOREL** (Stack Overflow RELation extractor)**, a deep-learning-based knowledge extraction engine.** SOREL identifies pairs of comparable API methods *and* explanations of their relationship from unseen Stack Overflow answers by learning and extrapolating from our hand-annotated

Table 5.1: Summary statistics for our annotated data.

Variable	Count
Annotated SO TensorFlow answers	587
SO TensorFlow answers with comparable API methods	198
Identified comparable TensorFlow API pairs	266
Unique TensorFlow methods mentioned in the annotated answers	642
Unique TensorFlow methods with comparable API methods	279
Sentences in the answers	4,298
Supporting sentences for the comparable API methods	737

dataset. Our evaluation results show that SOREL outperforms baselines and pattern-matching-based approaches, and even the recent off-the-shelf large language model, and can discover relevant novel facts compared to the training data, the official documentation, and simple Google search. As such, the SOREL approach, which could be applied analogously to other APIs beside TensorFlow as long as they are covered by sufficiently many Stack Overflow discussions, can be useful to API designers and tool builders looking to improve API documentation and learning.

5.2 A Benchmark of Comparable API Methods

Before we tested the benefits of providing comparable API methods and automate the extraction, we first created a benchmark of pairs of comparable API methods from Stack Overflow, as such dataset was necessary for both the user study and model training. In particular, we focused on the popular TensorFlow machine learning package (45,996 questions; 33,460 answers with at 1+ votes). Table 5.1 lists basic statistics for our sample, after filtering out 13 of the 600 answers ($\sim 2\%$) that were longer than 512 words, i.e., more than our deep-learning model can handle efficiently.[9]

Annotation Protocol. Next, we developed a labeling protocol to extract, for every Stack Overflow answer in our sample, a) the pairs of comparable API methods mentioned in the answer; b) for every pair, a list of the most relevant sentences from the answer, describing how the methods are related. Creating such annotations is both time-consuming and difficult: relations manifest in a diversity of ways and are often not explicit or well structured, but inconsistencies in labeling risk wreaking havoc on any downstream learner when using so little data. To ensure that our annotations were consistent, replicable, and generalizable, we created a detailed annotation protocol, refined through several pilot phases. The final protocol contains annotation steps, the definition of the relation, examples, and notes about edge cases. To increase validity, two researchers annotated each set of documents separately, measured their inter-annotator agreement (IAA), discussed disagreements, and updated the instructions. The IAA score for the first round of separate annotations (23 documents) was 0.30, and it rose to 0.82 in the second round (38 documents), after resolving disagreements and refining the protocol. As values over 0.8 are generally regarded as good agreement, both by the free-response kappa proponents [41] and the interpretation guideline for Cohen' s kappa [123], we considered the instructions adequate at that point. After this, I annotated the remaining documents alone following the final instructions.

[9]Only 3 of the 13 (0.5% of the entire 600 sample) contained comparable API methods.

Figure 5.2: Overview of our browser plugin: (1) When comparable API methods exists in our labeled dataset, the extension inserts a "vs" icon. The user can hover over it to activate the scrollable tooltip (5), which displays (2) the pair(s) of comparable API methods, each with links to their reference pages; (3) the relevant sentences for the comparison; (4) a link to the Stack Overflow answer where the sentences were extracted from.

Resulting Dataset. Table 5.1 summarizes basic statistics for the resulting dataset: the 198 answers with comparable API methods contained around three TensorFlow methods each; around one tenth of all TensorFlow symbols were mentioned as part of any comparison; around one-third of posts contained (typically several) related pairs, which are best described with about three sentences on average.

5.3 Information Presentation

To test our hypothesis, we conducted an IRB-approved human subjects study to investigate how providing developers with (1) a list of comparable API methods and (2) for each pair, a textual description of the comparison can assist developers in using new APIs more effectively. We first built a Chrome browser extension (Figure 5.2) that displays the information on comparable API methods we collected during our previous annotation effort. We deliberately chose a tooltip design to reduce information overload (tooltips only show contents when users trigger them), and to make the crowd-based information easily distinguishable from the official documentation. The tooltip is available on any webpage, including the official API documentation, Google search result pages, and Stack Overflow. We then ran a study that involves participants completing a set of tasks in two conditions, with and without access to the tooltip extension, and collected both quantitative and qualitative data on task performance and tool use.

5.3.1 Study Design

Participants. After advertising our study broadly inside the university community (Slack channels, posted flyers in Computer Science buildings, and personal contacts), we recruited 12 participants (6 men, 6 women) having a general understanding of machine learning (ML), who could understand the task requirements (9 PhD and 3 MS students, all in ML-relevant fields). We recruited additional 4 participants (all men, 1 research scientist, 1 ML lead, 2 MS students) from outside our university after advertising our study on Twitter. To minimize the possibility of the participants knowing solutions to our tasks without needing to search, we specifically looked for participants who had not used TensorFlow for more than 6 months.

Tasks. We designed a diverse set of eight ML-related programming tasks that mimic real-world TensorFlow use, ranging from tensor manipulation to image processing. For each task, participants were given the requirements as a short natural language description, an example input–output pair, and some starter code, and were asked to complete the implementation using appropriate TensorFlow API methods. We intentionally designed the tasks to have more than one acceptable solution (involving different TensorFlow API methods), so we could better test the participants' understanding of all possible options.

Experimental Design. We chose a within-subjects design, where each participant was assigned four of the possible eight tasks (to keep the participation effort manageable), two with our browser plugin enabled (treatment) and two with it disabled (control). We used the Youden square [98] (incomplete Latin square) procedure to counterbalance the tasks and order in which they are presented to participants, to prevent carryover effects. Control (plugin disabled) and treatment (plugin enabled) were randomly assigned. Overall, each of the possible eight tasks was used four times in the treatment condition and four times in the control condition.

Procedure. We conducted the study via a video conferencing tool, with each session taking about 60 minutes. At the beginning of the study, we asked participants to install the browser plugin, share their screen, and think aloud while completing the tasks. Before their first task in the treatment condition, we introduced the plugin briefly and showed the participants a short demo of how it worked. We also informed participants that there could be multiple solutions to a task, and asked them to make deliberate choices. Participants were free to use or read any web pages. To complete each task, we asked participants for their chosen API method names (but not to run any code). We then asked a few interview questions to understand their prior knowledge with the task and whether they understood the different options to make an informed decision, i.e., to list the API methods they considered and briefly describe the differences between them. At the end of the study we conducted a general interview eliciting participants' impressions of using the plugin and the usefulness of having the information on comparable API methods in completing the tasks.

5.3.2 Analysis

Data Collection. For qualitative data, we transcribed the interview parts of the video recordings. For quantitative data, we computed six outcome variables: (1) task completion **time** (in seconds); (2) number of search **queries**; (3) number of web **pages** visited; (4) **correctness** of the participant's solution; (5) the participant's **awareness** of comparable API methods in that context; and (6) their **understanding** of the differences between the comparable API methods. To account for a possible confounding factor we also rated each participant's **prior knowledge** of the task based on the interview responses, on a scale ranging from 0 (has no experience) to 3 (recalls the method name without search).

Analysis. To compare the six outcomes between tasks completed in the treatment and control conditions we estimated six mixed-effects multivariate regression models (one per outcome variables), with **prior knowledge** and the **condition** (treatment vs control) as fixed effects, and random intercepts for **task** and **participant** to account for variation in task difficulty and participant ability.

5.3.3 Results

None of our models for **time**, **queries**, or **pages** showed a statistically significant effect for **condition** at $\alpha = 0.05$. Thus, we could not find sufficient quantitative evidence to conclude that having access to information on comparable API methods has a significant impact on task completion times or web search queries. However, it is possible that our tasks were insufficiently complex[10] to uncover such differences between conditions with statistical confidence. We also could not find statistical evidence that presenting comparable API methods assists developers in selecting what we consider as the ideal API methods for a given task.

However, we did find clear evidence of increased **awareness** (coefficient = 3.03, z(59) = 3.18, p = 0.0015) and increased **understanding** (coefficient = 2.64, z(64) = 2.77, p = 0.0057) of the API methods in the treatment (tooltip) condition, supporting our hypothesis that **presenting comparable information helps developers understand the design space of API**: the odds of being aware of comparable API methods are about 20 times higher (exp(3.03)) among participants with access to the tooltip information compared to those without; similarly, the odds of understanding the differences between the comparable API methods are about 14 times higher. Many participants typically discovered the API methods they ended up submitting as their answers from among the comparable API methods suggested by the tooltip in the treatment condition. Specifically, we found that in more than half of the tasks (17 participant-task pairs out of 32), participants *newly* discovered the API methods *they submitted as their answers*, among the comparable API methods suggested by the tool.

From the qualitative analysis, one common theme was also that **the tooltip information on comparable API methods was often novel and welcome**. Almost all interviews appreciated having the list of comparable API methods, both for discoverability reasons (e.g., "[*otherwise*] *it would have been pretty hard for me to have actually found the correct documentation.*"–P16) as well as usability reasons (e.g., "*The tool allowed me to explore more methods more easily in the same page without retyping the search keyword.*"–P1). A few participants mentioned that such a tool could help close the "lexical gap" between search queries and web documents, e.g., "*Sometimes, I'm not-so-clear about what I' m looking for*" (P12).

5.4 Learning-based Information Extraction

In the previous section, we provided evidence for the benefit of knowledge extraction, and in particular presenting comparable API methods, to support developers' understanding of the API design space. We also derived requirements for an ML model to help automate this knowledge extraction process. In this section we present SOREL (Stack Overflow RELation extractor), an ML model that collects comparable API methods and relevant supporting natural language sentences from Stack Overflow answers.

To discover new facts, represented as triplets (entity 1, relation, entity 2) in unstructured text, one typically follows a two-pronged knowledge extraction approach. First, Named Entity Recognition (NER) is used to locate and classify entities. Relation Extraction (RE) then identifies relations of various kinds (including "no relation") between entity pairs.

In general, both problems are hard and the focus of active research in NLP, e.g., [105, 242]. In our work, since we are extracting only API method entities, we found that the NER step

[10] E.g., on average users made 1.3 (treatment) to 1.4 (control) web searches per task.

can be done accurately using pattern matching based on the full list of TensorFlow API symbols,[11] therefore we focused on the RE problem instead. In the RE step, we aim to identify the comparative relation “is comparable to” in an input Stack Overflow answer with more than one TensorFlow API call. To make the identified relations more useful, we also identify one or more sentences from the same answer, that offer the evidence supporting each identified relation. This provides a type of (extractive) summary for each relation. Therefore, our task is divided into two sub-problems:

- Comparative-relation extraction (RE) between entities.
- Supporting-evidence prediction (SEP).

5.4.1 Model Architecture

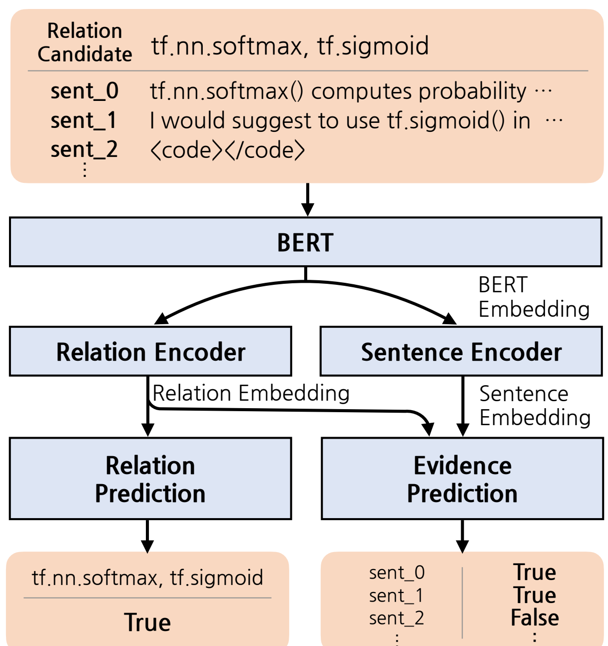


Figure 5.3: The architecture of SOREL, which learns to infer the comparison relation and the supporting evidence.

Figure 5.3 shows the architecture of our model, which is inspired by Yao et al.’s DocRED [276]. The model consists of four main components:[12]

BERT. We use a language representation model called BERT (Bidirectional Encoder Representations from Transformers) [61] to represent Stack Overflow answers. Due to its bidirectional nature, it provides deeper contextual information, which can help with document-level relation extraction. We used the pre-trained BERT model and a tokenizer from the original BERT paper

[11]In 50 randomly selected answers tagged with TensorFlow, the pattern matching approach missed just 12 out of 133 mentions of TensorFlow methods, achieving 91% recall and 100% precision.

[12]For ablations of the model design, see Table 5.2.

provided via huggingface.[13][14] Due to the small amount of training data, we kept the BERT model weights frozen during the training, fine-tuning just the classification layers – a common approach when using small fine-tuning datasets [102, 103].

Relation Encoder generates representations for a pair of entities. Embeddings of the same entity which occur multiple times are averaged across the document, to allow sharing their learned representation across occurrences. The relation encoder combines two entities' averaged BERT embeddings using a bilinear function, which captures the mutual agreement between their respective representations. Similar to Yao *et al.*, we concatenated each entity representation with a relative distance embedding before passing it to the bilinear function [276]. The relative distance embedding represents the relative distances of the first mentions of each unique pair of entities in the document, informing the model of how closely together the two are mentioned. See [276] for details.

Sentence Encoder generates contextualized representations for each sentence in the input document. We use a two-layer bidirectional LSTM (BiLSTM) model to represent each token in the input document. BiLSTMs combine two LSTMs, one traversing the sequence forward and one backwards, to allow integrating information from the context on both sides of each token. This helps the model capture broader context around each sentence specifically for the sentence prediction task. An alternative would be to fine-tune the underlying BERT model, but we found this to be ineffective due to the small size of our training data (Table 5.2). A single-layer, low-dimensional BiLSTM contains comparatively far fewer parameters to calibrate. To obtain a sentence embedding, we use the BiLSTM's output representation of the first token of sentence, which is the same (`[cls]`) token that BERT models use to extract sentence embeddings.

5.4.2 Model Training

As RE and SEP are highly connected tasks, we train SOREL RE and SEP objectives simultaneously, by combining the two losses: $loss = \alpha * re\_loss + sep\_loss$. We adjusted the hyperparameter α experimentally so the two losses converge at a similar rate, as minimizing the sep_loss involves more trainable parameters and thus typically takes more iterations.

We randomly split our annotated dataset from Section 5.2 into a training and test set in a 4:1 ratio. We tuned the model hyper-parameters through 5-fold cross validation on the training portion (470 samples). In selecting hyper-parameters, when there are two settings with very similar overall performance, we selected the one that yielded a lower RE loss, under the premise that better comparable API methods extraction benefits a broader group of developers than having a good summary of the differences (as evidenced in Section 5.3.3).

To maximize the utility of our limited training data, we trained with a relatively low learning rate (1e-5) and frequently checked the held-out results to ensure that we captured the best performing model. We also added *input dropout*, which randomly omits 40% of tokens from the input at training time [237]. This has the effect of preventing the model from overfitting on known inputs by artificially creating many versions of the same inputs, a form of data augmentation.

Table 5.2: Overall performance on each subtask, and ablations (on test set) of the model components and of the training set size (%).

	RE				SEP			
	A	P	R	F1	A	P	R	F1
Train	93.7	85.6	76.6	83.7	80.0	73.2	65.5	71.5
Val	89.3	64.3	89.6	67.9	76.2	65.9	61.6	64.7
Test	84.5	71.3	55.0	67.3	77.6	75.5	47.8	67.6
SOREL	**84.5**	**71.3**	55.0	**67.3**	**77.6**	**75.5**	**47.8**	**67.6**
- BERT	80.1	56.5	**59.3**	57.0	61.8	39.7	29.8	37.2
- BiLSTM	83.5	67.2	55.7	64.6	70.4	58.4	36.7	52.2
+ Finetune	80.9	58.6	58.6	58.6	69.8	57.3	34.0	50.4
All data (470)	84.5	71.3	55.0	67.3	77.6	75.5	47.8	67.6
2/3rd (315)	85.8	77.6	54.3	71.4	75.8	64.9	58.4	63.5
1/3rd (155)	84.5	73.0	52.1	67.6	73.9	81.1	27.6	58.5
1/10th (45)	78.8	53.3	64.3	55.2	65.9	45.9	17.9	35.0

5.4.3 Evaluations with Test Data and Ablations

We now evaluate how well SOREL can extract knowledge on comparable API methods from Stack Overflow answers, focusing first on the test subset of our manually annotated data. Table 5.2 summarizes the train / (average) validation / test performance scores and ablations of the model components and the training set size. The final model achieves around 67% F1 score and around 80% accuracy on both tasks (RE and SEP) on the test set, which is a reasonable return given the small amount of training data.

Training custom word embedding instead of using BERT drops performance significantly on both RE and SEP. This confirms the benefit of adopting a large pre-trained language model when only a little training data is available. Not using a BiLSTM for SEP also reduced the performance significantly, which shows that SEP relies on this component to capture the global context beyond the initial representations offered by BERT. An alternative option, fine-tuning BERT along with our other parameters, decreased performance, as we might expect given the small size of the dataset.

Models show fairly steady gains in performance as we train on a progressively larger subset of the training data, starting from just 45 samples. While RE performance fairly quickly saturates, SEP performance progressively improves with more data. Still, we are content that a corpus of our size is right around the smallest size (and thus requires the least annotator overhead) where our model performance is adequately capable of generalizing, achieving balanced F1 results for RE and SEP, and accuracy around 80% on each.

[13] https://huggingface.co/bert-base-uncased

[14] We also considered BertOverflow [242], which was pre-trained with a Stack Overflow corpus, but found the original BERT to work better.

Table 5.3: Recall comparison with Google Autocomplete, Google's Top-5 results, TensorFlow documentation, heuristics, DiffTech [264], and APIComp [148]), ChatGPT [2] on test set (%) after excluding deprecated API methods.

	SOREL	Auto	Top-5	Doc	Heur.	DiffTech	APIComp	ChatGPT
RE	55.3	28.0	18.2	29.2	30.3	33.3	16.7	51.51
SEP	55.2	-	-	-	48.1	5.7	26.4	-

5.4.4 Comparison with Baselines and Prior Work

For the relation extraction (**RE**) sub-task, we compare SOREL with: (1) general-purpose web search; (2) the TensorFlow reference documentation; (3) a simple heuristic approach; (4) an adaptation of the most closely related prior work approach, DiffTech [264], which can identify pairs of similar technologies (e.g., *golang* and *javascript* for programming language) from Stack Overflow, a task which is conceptually similar to ours, albeit at a higher level of abstraction; and (5) ChatGPT [2]. See Table 5.3 for a summary of the results.

[RE] SOREL vs Google Search

Developers commonly find information on comparable API methods using a generic search engine. Given a pair of fully-qualified comparable API methods (A, B) in our test set, we enter the Google search query "A" and record whether B is mentioned anywhere within the top-5 Google search result landing pages; we then repeat the search in reverse, starting from "B". The recall measure we report for Google Search for the pair (A, B) is the average of the two. In addition, we separately inspect the Google search autocomplete suggestions for the query prefixes "A" and "A vs" (similarly "B" and "B vs") and check for the presence of B (A) among the suggestions. SOREL identified roughly twice as many relations as Google search in both cases.

[RE] SOREL vs Reference Documentation

Similarly, given a pair (A, B) we inspect the API reference documentation page for A and record whether it contains a link to B and vice versa, excluding all pairs involving a deprecated API call, which have no documentation page. Again, SOREL identified roughly twice as many relations as were mentioned in the official documentation.

[RE] SOREL vs Heuristic

Next, we compare SOREL to a simple, intuitive heuristic: if a Stack Overflow answer mentions exactly two API methods, consider those two to be comparable and record the pair. Across the 66 ground truth comparable pairs in test set, there are 52 answers with exactly two API methods mentioned, so one could hope to extract 52 comparable pairs. However, many of these pairs were not actually comparable (only 20 pairs were), so this simple heuristic is quite noisy, corresponding to 30.30% recall and 38.46% precision. Recall the example in Figure 5.1 – comparable pairs are often found within answers mentioning more than two API methods; the two API methods mentioned in answers with exactly two methods are often not comparable.

[RE] SOREL vs Word2Vec

We compare SOREL to DiffTech [264], the prior work closest to ours. DiffTech collects pairs of similar technologies (e.g., libraries, frameworks, programming languages) based on the intuition that frequently co-occurring Stack Overflow tags corresponding to different technologies may share a similar meaning. Specifically, DiffTech embeds tags with a Word2Vec [162] model trained on a corpus of tag sentences, and identifies pairs of tags as related when their embeddings have a cosine similarity greater than 0.4. While the DiffTech approach is designed to solve a different problem, the intuition carries over to our context: API methods may frequently co-occur with others they are comparable with, even though they rarely have dedicated tags. Therefore, we trained a Word2Vec model using API methods from our entire corpus of 33,460 TensorFlow answers and tested how many of the labeled comparable pairs in our test set have embeddings more than 0.4 cosine-similar. Only 16 (24%) of the comparable API method pairs in our labeled test set were among each other's top-5 nearest neighbors, and 22 (33%) were among the top-20, highlighting that our adaptation of the approach struggles to match comparable API calls based on co-occurrence statistics alone.

[RE] SOREL vs ChatGPT

We also compare with ChatGPT [2], a large language model that is trained to generate a text response given a prompt. Given a pair of fully-qualified comparable API methods (A, B) in our test set, we generated a ChatGPT response using a prompt template, "In Tensorflow, what are the comparable methods for A" and recorded whether B is mentioned in the response; we then repeat it in reverse, using "B" in the prompt template. The recall measure we report for ChatGPT for the pair (A, B) is the average of the two. Despite the significant model size difference, SOREL achieved slightly better recall compared to ChatGPT, showing the benefits and potentials of designing and training intelligent solutions with a good understanding of the problem.

Next, we turn to the Supporting-evidence prediction (**SEP**) sub-task and compare SOREL to: (5) a heuristic approach; and (6) two approaches based on linguistic patterns, inspired by related prior work [148, 264].

[SEP] SOREL vs Heuristic

Intuitively, one might expect that given a pair of comparable API methods (A, B), all the sentences mentioning either A or B can be considered as supporting evidence. We test this heuristic for the true positive pairs from the RE step, for a fair comparison. In this limited setting, assuming the RE step is perfectly accurate, the heuristic approach performs quite well (Table 5.3), but SOREL nevertheless manages to improve over it.

[SEP] SOREL vs Pattern Matching

Pattern-matching-based approaches have been used before to extract information from online documentation, with the two closest predecessors arguably being the DiffTech [264] approach discussed above for RE and APIComp [148]. A fundamental difference between SOREL and such approaches is where human effort is spent during the "training" process. SOREL requires human annotation effort to label API pairs and supporting sentences as valid / relevant or not. Pattern-matching-based approaches require human effort to identify the linguistic patterns needed to extract relevant sentences from text. Fundamentally, SOREL's type of effort can be expected to scale more easily to new APIs and sources of documentation, because it requires less expertise in NLP. Hence, we argue that designing a pattern-matching-based approach for our task *from scratch* may not be preferable. Still, one can ask a pragmatic question – what if the predefined linguistic patterns from DiffTech [264] and APIComp [148] generalize well enough to extracting supporting evidence for comparable API methods from Stack Overflow *without any additional effort*? We investigate this next.

For DiffTech [264], the authors identified and validated a series of linguistic patterns based on sequences of part-of-speech tags (e.g., "RBR (comparative adverb) JJ (adjective) IN (preposition)" as in "more efficient than") for extracting supporting sentences for the identified pairs of comparable technologies. Testing these same patterns on the answers in our sample containing 22 pairs identified by the previous RE step, we find that out of 70 sentences in 16 answers that were labeled as supporting evidence, only 4 sentences matched DiffTech's exact patterns (5%).

APIComp [148] is designed for a different usage scenario – to explain, given a pair of API methods, not necessarily "comparable" per our definition, the relationship between them using text extracted from reference documentation. APIComp first extracts sentences describing an API element from official reference documentation using linguistic patterns, and then aligns and compares the extracted sentences given an API knowledge graph. To test whether the same linguistic patterns carry over to our task, we applied the APIComp patterns to extract API statements from Stack Overflow answers in our test set, finding that only 33 out of 125 known supporting evidence sentences were matched (26.4%), identifying only 11 comparable API pairs out of 66 available (17%).

We conclude that pattern-matching-based approaches, that were commonly used for similar tasks in the past, are not directly applicable to our task, and that anyway it may not be preferable to design a custom pattern-matching-based approach for relevant sentence extraction for our comparable API methods task. Recall is typically low because relations can be expressed in a variety of ways, hard to capture with reliable patterns, and pattern building typically requires significant amounts of manual effort.

5.4.5 Generalization to Larger Dataset

We next turn to our most ambitious target: assessing how well our model can generalize and rank insights from all 33,460 up-voted answers tagged with TensorFlow. After filtering out posts containing fewer than 2 Tensorflow symbols, 2,014 documents remained. Using our model, we identify 433 pairs of comparable API methods with 744 supporting evidence sentences. We report two such <u>new</u> (not included in our training data) relations in Table 5.4, as an example.

We manually reviewed the top 50 answers as sorted by the descending probability of both the relation and the supporting evidence sentences (according to the model's predictions, multiplied together). As many unique pairs of API methods were discovered in multiple Stack Overflow answers, we focused on relations detected twice or more, which we expect are more likely to be both accurate as well as relevant to developers. To compute the accuracy for RE on these, we follow the same guidelines as in Section 5.2, marking relations correct when the relation between the two entities was factual and explicitly represented in the answer. Similarly, for SEP, we labeled samples correct when the predicted sentences explain the relation sufficiently without missing sentences. Note that SEP is dependent on the results of the RE, the model cannot

Table 5.4: A sample of new comparable API methods pairs and their supporting evidence extracted from Stack Overflow.

API Pair	**`tf.convert_to_tensor`, `tf.constant`**
Evidence	[SO Answer: 50981199] Each time a tensorflow operation expects a Tensor input but receives something else, it tries to convert it with `tf.convert_to_tensor`, and if successful, proceeds normally with that output. In case of a constant like 2, but also np.arrays, lists, tuples, or virtually any container or (well-formed) hierarchy of containers, `tf.convert_to_tensor` will create a Const operation, just like the one you would create by hand by calling `tf.constant`. `tf.constant` is nonetheless useful because it allows you, among other things, to name your constant in the graph and to control its data type (with name and dtype arguments respectively).
API Pair	**`tf.nn.embedding_lookup`, `tf.gather`**
Evidence	[SO Answer: 46440226] If `params` is a single tensor, the `tf.nn.embedding_lookup(params,ids)` operation treats `ids` as the indices of rows in `params`. If `params` is a list of tensors or a partitioned variable, then `ids` still correspond to rows in those tensors, but the `partition_strategy` (either `"div"` or `"mod"`) determines how the `ids` map to a particular row. Alternatively, you can use the `axis` argument to `tf.gather()` to select columns from `U`:

predict a correct supporting evidence sentence for a "faulty" relation or "no relation". Therefore, we only considered correct RE predictions when computing scores for SEP.

On these 50 relations extracted from top-scoring answers, the RE precision is 50% and SEP average accuracy is 77%.[15] Importantly, 90% of these correct relations were novel—not present in the training data. These results highlight that despite training on a small amount of data, a model like ours can show its strength when deployed on a very large corpus, where facts abound and high recall may be less important.

5.4.6 Error Analysis

Next we describe the patterns we observed in the mistakes the model makes. One common type of mistake was false positives due to lack of explicit topic change. Unlike API reference documentation pages, Stack Overflow posts often cover multiple topics in a short document. Therefore, the model sometimes failed to recognize the abrupt topic change and predicted a relation assuming that two entities were discussed in the same context. This error might be mitigated by preserving the original delineation of paragraphs and code blocks in the input data, though low textual consistency is a fundamental issue with Stack Overflow posts [146, 258].

Another common type of false positive is answers describing sequential use of API methods, such as ones explaining a code snippet line by line. As many of the labeled comparable API

[15] As there is no label for this data, we cannot compute other metrics for RE. For SEP, we averaged the accuracies of each answer.

methods were described in sequence, the model often made wrong predictions in this case. This confusion is rather reasonable; when annotating the data, we occasionally found it challenging to determine whether methods in a pair were explicitly compared or simply mentioned in sequence.

Finally, code in text was a general issue for both RE and SEP. Although we filtered out code snippets, many answers contain code intermixed with their text. Our model may fail to capture such code's context and meaning, which results in some false-positive API method pairs predicted as having a relation whenever they are simply used together.

5.4.7 Threats to Validity

A threat to the external validity of our results is that we only used TensorFlow answers to build and evaluate our model. Therefore, our model might not generalize to other ML libraries such as PyTorch, or other domains such as Cloud or Graphics. While we do not expect other libraries and domains (with sufficient programmer interest) to have particularly less informative Stack Overflow answers, it remains a subject of future work to adapt our protocol to such use-cases.

We did not compare our model with state-of-the-art document-level relation extraction models built for general corpora such as Wikipedia [93, 243]. While we adopt a similar methodology, our corpus is both much smaller, less structured, and more domain-specific than the general Wikipedia corpus. We believe omitting such a comparison is reasonable as our goal was to explore and demonstrate the feasibility of exploiting Stack Overflow's relational insights using machine learning. As such, we do not claim that our model presents the best performance for this problem.

5.5 Discussion

We showed that providing extracted comparable API methods can help developers understand the design space of APIs. We further built SOREL, which using relatively little labeled data can jointly extract relations and supporting evidence from new Stack Overflow answers. Next we present implications of our work for practitioners and researchers in this area.

ML-based Knowledge Extraction To our knowledge, this is the first application of learning-based relation extraction, as well as supporting evidence prediction, on Stack Overflow data. The state-of-the-art knowledge extraction techniques we use make our pipeline a good fit for extracting structured knowledge from an environment as unstructured and diverse as Stack Overflow. A next step may be to annotate a corpus spanning multiple libraries, such as Pandas and PyTorch, following our protocol. Besides enabling SOREL for new libraries, this might also benefit performance overall: developers likely use similar patterns when comparing and contrasting terms from APIs, which could be useful to a model even when targeting just one API. Our results in Table 5.2 suggest prediction accuracy can yet improve with more data.

Reference Documentation In the control condition (tooltips disabled) in our study, discoverability issues emerged due to a lack of "functional links" in the API documentation, that could only occasionally be retrieved via Google search, or had similar enough method names to be located nearby in the method navigation bar. However, when the reference documentation includes an explicit note about functionally-related methods, participants were able to discover them quite easily (see Section 5.3.3). Our detected comparable API methods could thus be especially useful for API documentation writers, to improve API discoverability. While automatically identified relations require the API to be popular enough on Stack Overflow and may include false positives or outdated pairs [74, 281], those well-versed in the API can identify correct relations in a list of extracted relations with little effort. This can quickly yield dozens of new functional links to boost API discoverability.

API Knowledge Support Tools We believe it is worth developing other types of API knowledge acquisition tools like our prototype. One finding from our study that we believe might generalize is related to differences in problem solving and information foraging strategies between groups of users (we saw this especially around participant reactions to the summary sentences), which have also been noted previously [49, 100, 159]. This suggests that when designing knowledge support tools for developers, allowing users to configure the types and depth of information presented to them (e.g., whether to provide the summary) could be useful.

Another factor to consider is where the tool will be deployed. From the study, we found that the usefulness of our tool was highly dependent on Google search results, as most users activated the tooltip while on the Google search page. Some participants also suggested utilizing search queries so that they do not see methods that are less relevant to their tasks from the list (see Section 5.3.3 for details). We believe that similar constraints (or benefits) will exist for other platforms (e.g., IDE), and thus, how users will consume the presented knowledge should be considered when designing the tool.

5.6 Summary

From the user study, we provide evidence that showing comparable API methods and summaries about their difference can improve developers' understanding of the API design space, and help them select API methods by taking into account the differences between alternative solutions. This supports my claim that information presentation considering the user context, push-based support instead of pull-based, can enhance developers' information collection for learning.

We further showed that a learning-based model can reasonably accurately extract such knowledge from unstructured Stack Overflow answers: our model identifies comparable API methods with a precision of 71% and summaries for these with 75% precision. Compared with existing pattern-matching-based approaches, SOREL outperformed, showing the benefit of using learning-based approaches.

Chapter 6

Testing the Feasibility of Generation-based Information Support[16]

After testing the effectiveness of learning-based approaches, we tested the feasibility of generating information so that we can provide information support that is more suited to the user's needs. We focused on the same task context as the previous work, where a user needs to write code using unfamiliar libraries. We aimed to provide a sequence of API methods that will fulfill developers' needs, to help them quickly write code given input and output pairs. In this work, we used a pair of input and output values as a way to specify the user's needs and provided a sequence of API methods that can fulfill the needs. By building a language model that predicts a sequence of API methods, we focused on investigating whether a generation-based approach can be useful in providing information support for programmers learning new APIs.

6.1 Introduction

One of the cherished dreams of the programming languages research community is to enable the automated synthesis of programs based on a specification. Synthesis approaches have been designed around several different forms of specification, e.g. a formal specification, or natural language description, or input-output examples (aka demonstration), or a combination thereof. Just as well, several different approaches to synthesis have been researched; see Related Work.

Our focus is on coding assistance for users of numeric libraries such as PyTorch, Tensorflow, Numpy, Pandas, and the like, each of which provide powerful data manipulation routines behind an API, and the API functions are generally side-effect free. We assume a specification in the form of a single input-output example, and we are looking for a straight-line program consisting of calls to API functions. We choose *enumerative synthesis* (explained in the next subsection) as the underlying synthesis approach. Our research goal—shared with recent works such as DeepCoder [24], TF-Coder [229], Autopandas [28], and others—is to speed up plain enumerative synthesis using machine learning (ML).

[16]This chapter is adapted from Nam *et al.* [175]

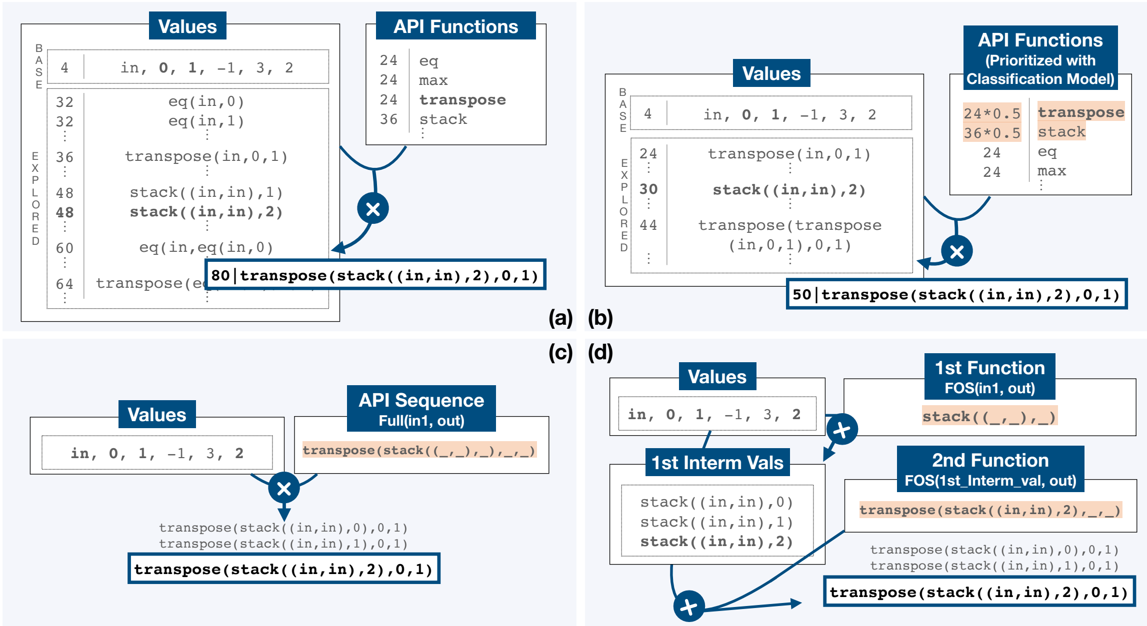


Figure 6.1: Overview of ML guided enumerative search algorithms. (a) Weighted enumerative synthesis without ML model incorporation [229], (b) weighted enumerative synthesis with one-time ML-based prioritization [24, 229], (c) incorporation of Full-Seq prediction mode, (d) incorporation of First-Of-Seq prediction mode. Red-highlight indicates the API functions predicted by ML models. The underscore is a placeholder for argument values. Numbers (in (a),(b)) on the left side are the costs assigned to the values and API functions.

Here is an input matrix as well as the desired output matrix, and the synthesis problem is to come up with a sequence of function calls that would convert the input to output. We will use the PyTorch API for this purpose.

```
in = [[5., 2.], [1., 3.], [0., -1.]]
out = [[[5., 5.], [1., 1.], [0., 0.]],
       [[2., 2.], [3., 3.], [-1., -1.]]]
```

The desired code fragment for this example is:

```
transpose(stack((in, in), 2), 0, 1)
```

The goal of program synthesis is to arrive at this expression, given only the input and output. Keep in mind that it is unlikely that random guessing of an expression will work: there are tens if not hundreds of available functions, and each function might take more than one argument. Thus, a systematic search is necessary.

Basic enumerative synthesis Refer to Figure 6.1, part (a), where we illustrate an enumerative synthesis in the style of Transit [259] and TF-Coder [229]. The idea is to organize the search in the order of increasingly complex expression trees, where the complexity is approximated by a *cost*. We assign a cost to each available value, and to each operation, which here are API functions. (The cost of a function is assigned heuristically, e.g. based on a global frequency of usage.) At each step, we work with a budget, which grows in successive steps. Expressions that can be formed from existing values within the budget are added to a pool of values. For instance, the expression `stack((in,in),2)` would cost the two times the cost of `in` plus the costs of the value `2` and the function `stack`. In the figure, this cost comes to 48, based on the cost of `stack` being 36. The value computed by this expression is added to the pool of values, along with the expression that computes them and its cost. The process continues, with increasing budget at each step, until the desired output value is found. The expressions added to the pool of values are shown in the figure.

Trying likely functions first The enumerative search presented above is slow, and gets exponentially slower if a larger expression is needed to get the job done. A reason for this slowness is that the turn of the actually needed API function might come in quite late, as enumerative synthesis makes its way through the smaller cost budget and cheaper functions. Balog et al [24], in their seminal work DeepCoder, described a machine learning based strategy to accelerate enumerative program synthesis. DeepCoder's insight is to re-assign costs to functions—based on a machine learning model over the given input and output—such that the function(s) more likely to be needed in a *given* situation are prioritized. See part (b) of Figure 6.1. Here, given the specific input and output, DeepCoder's machine learning model, adapted to our setting, correctly deems `transpose`, and `stack` as likely to be needed. Operationally, an enumerative synthesis process (e.g. as implemented in TF-Coder [229]) can lower costs of these operations by some factor, so they are likely to be tried in preference to other API functions. The hope is that if the ML prediction is accurate, and the discounted costs work out, the process of enumerative synthesis can be sped up considerably.

Predicting function sequences Our thesis is that ML can be used in the setting of enumerative synthesis of API-centric code in a more powerful way: not for prioritization, but instead to directly predict the sequence of API functions that required to go from input(s) to the desired output. We describe two ways in which such a predictive model can be used to accelerate enumerative synthesis.

The first way in which we use this prediction model is to just let it predict the entire sequence of API functions in one shot, given the input and the output. In our running example, the model will predict `stack, transpose` as the sequence. See Figure 6.1, part (c). Given this sequence, the enumerative synthesizer will only look for values to fill into the function call arguments (shown by "_"). If the model predicts correctly, the search space that an enumerative synthesizer faces is vastly reduced, leading to possibly significant speedups.

A second way in which we use this predictive model is to use it as a "first-of-sequence" (FOS) predictor. See Figure 6.1, part (d). Given the input and the desired output, the FOS predictor only predicts the *first* function in the sequence needed. Say it predicts that function is `stack`. The synthesizer tries out a set of concrete arguments for `stack` from the values pool. The result of evaluating `stack` on each of these sets of arguments is added to the values pool; these are intermediate values in the desired computation. Next, for each intermediate value thus obtained, the synthesizer invokes the model again, this time giving it the intermediate value (in place of the input) and the desired output value. Say the model now predicts that the first function in the *remaining* sequence needed is `transpose`. The synthesizer then looks for appropriate arguments for `transpose`. At this point, one of the argument choices would provide the desired output. Compared to the full prediction, the point of this FOS mode is that it gets to predict on the basis on *known* intermediate values, a bit akin to teacher forcing [267] in sequence prediction, and can

Table 6.1: Sample of synthesized programs with Full-Seq model guided enumerative synthesis and the synthesis time comparison. More examples can be found in the Appendix B.

Synthesized Program	Full-Seq (s)	no ML (s)
`eq(in1,unsqueeze(in1,1))`	0.18	0.8
`tensordot(in1,transpose(in2,0,1),1)`	0.32	2.06

be successful more often than the full prediction mode; but it can be less efficient than one-shot prediction of the entire sequence.

On the running example, here are the comparative times to a successful solution: plain enumerative synthesis, 54.79 seconds; DeepCoder-style ML-based prioritization, 34.71 seconds; our API sequence prediction, FOS mode, 0.49 seconds; and API sequence prediction, Full mode, 1.45 seconds. In this example, the full sequence prediction mode took a tad longer than the FOS mode: this is because the correct full sequence was in top-3 but not top-1, whereas in the FOS model, the correct choice was at top-1. In general, we have found the full sequence mode to be faster than the FOS mode. Other examples of Full-Seq guided synthesis are available in Table 6.1.

Contributions We make two contributions in this work. First, we present a way to incorporate powerful predictive models in the context of enumerative program synthesis. On a suite of benchmarks (adapted from Stack Overflow) for PyTorch, using our ML models reduces the (mean, max) synthesis time from (10.01, 96.53) to (1.04, 9.58). By contrast, an adaptation of the idea of DeepCoder [24] reduces the (mean,max) synthesis time only to (7.44, 77.00). (See Section 6.6.3, Table 6.4.)

Second, our main technical advancement is in being able to carry out prediction of a sequence of API functions, given the input and the final desired output. Specifically, our model predicts one API function at a time and executes each predicted API function to convert the (intermediate) input state into another intermediate state until it becomes the target output state. Here, the intermediate states are not given to the model, but the model learns to represent what *would be* concrete intermediate values in the latent space during the training time. The ability to execute the API functions in the latent space indicates that the model learns the API function semantics (i.e., the relation between the input and output states) rather than the sequence distribution of the training dataset, and allows the model to generalize to unseen sequences or lengths. See Section 6.7.

6.2 Backgrounds

Program Synthesis has a rich literature, including example driven LISP code generation [94, 226], deductive synthesis from decades ago [154], sketch completion using satisfiability [236], bit-vector manipulations [109], string processing [90, 189], data processing [235, 274], syntax transformations [216], database queries [273], data wrangling [71, 72, 129], and the highly successful programming-by-examples system FlashFill [90]. FlashFill uses enumerative synthesis, where a space of programs is explored in some order, until one that fulfills a requirement—typically one or more examples—is found. Transit [259] is another well-known work in enumerative synthesis, where the exploration is arranged in terms of finding sub-expressions in order of their *costs*. Increasingly more costly expressions are attempted, using expressions previously computed.

ML for Program Synthesis. With advances in ML, researchers tried to adopt ML on top of the enumerative search for more efficient program synthesis [24, 28, 183, 186, 229]. Our work is closely related to DeepCoder [24], TF-Coder [229] and BUSTLE [186]. Using a prediction-guided enumerative synthesizer, they show the benefits of predicting API functions that are needed *somewhere* given a synthesis instance. However, they all use an explicit featurization over these input-output values, which is not easy to generalize to other programming languages. Also, they only predict the presence or absence of API functions, the prediction was only used to prioritize operations in the enumerative search, rather than directly predicting the API function(s) in sequence. With the ML model guiding the search, BUSTLE takes an approach similar to ours, which gives feedback to search iteratively, whereas the models of DeepCoder or TF-Coder only give feedback in the beginning of the search. However, BUSTLE and DeepCoder only support simple DSL tasks, which may not be generalized for real-world API-based synthesis.

Neural Program Synthesis. Approaches like [15, 25, 38, 60, 189, 189, 279] directly use neural networks for end-to-end synthesis [25, 38, 60, 189] to generate string transformation programs from examples. These works generally use the encoder-decoder model. In particular, the encoder embeds the input/output strings, and the decoder generates the program sequences conditioned on the input embedding. However, these approaches are mostly built and evaluated with simple DSL tasks, mostly with simple string transformation. In this work, we worked on the real-world tensor manipulation library PyTorch. Although our evaluation does not cover the full range of PyTorch, we found several challenges in expanding this work into more complex programs, such as the scalability issue in training data generation and diversity of the API parameters especially in the tensor domain.

Execution-guided Program Synthesis. Recent works have tried to exploit program execution to learn better representations for the neural program synthesis [38, 45, 46, 68, 184, 231]. Some of the approaches [38, 231] use program interpreters to provide the actual intermediate execution results, and a more recent approach [46] learns the latent representation to approximate the execution of partial programs using a separate "Latent Executor". We also learn the representation of the execution of partial programs and demonstrate that (see Section 6.4). However, we capture the intermediate execution results as part of the main recurrent model, without needing to use the separate module to approximate or execute the program.

6.3 Learning to predict API sequences

Our technique works based on supervised learning over a large number of input and output examples, trained over individual API functions, or on sequences of API functions. Since the availability of real training data is a pervasive problem in ML, we use synthetic data generation similar to prior program synthesis work [232]. We pipe randomly generated diverse inputs through sequences of API functions and collect resulting outputs (see Section 6.4.4). This helps capture the behavior of a single or a sequence of API functions in terms of how it transforms its input to the output.

Once trained, the model is able to predict a sequence of API functions. It can predict for input-output pairs that were never seen in the training data; thus it generalizes in the data space as long as the query input-output pairs are in distribution. More interestingly, it can predict sequences of API functions that were not seen in the training set either. This latter point is crucial, because the way we train the model, it learns to *compose* new, previously unseen sequences from the behaviors learned from training sequences.

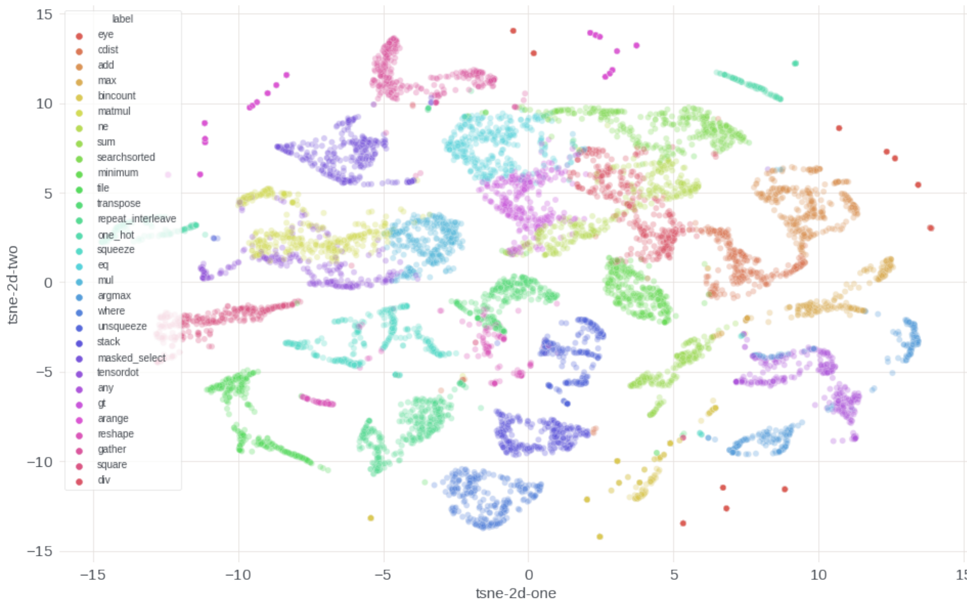


Figure 6.2: Visualization of embedding space of input-output pairs.

Before we present operational details (Section 6.4 onwards), we would like to present some intuitions behind our proposed ideas. We start with a basic classification model designed to predict one API function, given an input and desired output; and then build over it a compositional model that is designed to predict a sequence of API functions. The importance of examining the classification model on its own was crucial in our own journey, because it helped overcome several challenges in synthetic data generation for training. (In actual synthesis application, we use the model that predicts function sequences, described after this.)

Predicting a function from input-output data The first intuition we use is that for many common API functions, their behavior—the relationship of output to inputs—has simple patterns. Moreover, the behavior of a function is *discriminable* from behaviors of other functions based on simple clues. Many functions simply move around elements of a data structure (e.g. `transpose` or `reverse`) in easy-to-recognize patterns. In other cases, the operation is a simple element-wise computation.[17] This suggests that a feed-forward neural network can be trained to predict likely functions—as in a multi-classification problem—from a representation of the input and output data. Such a network would have to be trained on large amounts of input-output examples and their known (ground truth) functions.

Figure 6.2 shows the tSNE plot for 5000 input-output pairs. The classification model was trained over synthetic data generated to classify among one of 33 API calls from PyTorch. The figure shows that the input-output pairs – or rather, their embeddings – map to visually distinct clusters, corresponding to the function calls that would be needed to go from the input to the output. The reason this clusters cleanly is that the network learns to pick up the essential patterns that appear in the input and corresponding outputs.

[17]There do exist operations with more complex behaviors, but here we limit ourselves to simple ones (see Appendix A.

Predicting sequences of functions The case of predicting an API *sequence*, such as `stack` followed by `transpose`, is harder. The intermediate values that flow between API calls are not known ahead of time, so it is not possible to reconstitute this sequence simply by invoking the classification model (for one API method name) over successive pairs of inputs and outputs. Moreover, learning to recognize the intended sequence from among all possible sequences, based on an input and the final output, can be difficult, for reasons for computational cost, for a classification model that predicts over a fixed collection of sequences of API function names.

This is where a second intuition comes into play. Given an input and a final output, we can imagine a model that predicts the *first* function in the intended sequence of the API functions that would process the input and eventually produce the (final) output. Crucially, we train this model as a *recurrent* unit, such that it not only predicts the first API function needed, but additionally produces a representation (in the embedding space) of the output of that first API function. (The correspondence of the internal representations to intermediate values is further explored in Section 6.7.) This representation, along with the final output, can then be passed to a *recurrent* invocation of the model, to make it predict the next API function in the sequence. In this way, we can train a compositional model for API sequences.

In our running example, the model first predicts `stack` based on `in` and `out`. Importantly, it also computes an internal representation of the intermediate value `stack((in,in),2)`. It then predicts `transpose` based on this internal representation and `out`. This is the principle by which the model is able to compose even longer previously unseen sequences.

Here we emphasize that the model is *not* predicting the next API token (e.g. `transpose`) based on the tokens that came before (e.g. `stack`), as is done in code completion models [97]. At each step, the prediction is based *only* on (an internal representation of) the program state, as opposed to on program text. This is a new capability, which could optionally be combined with additional signals such as previous tokens, if desired.

6.4 Technical Details

In this section, we explain our models, the training and the inference. We will use Figure 6.3 to show details using an example.

6.4.1 Notations

We will work with the following entities:

- $\mathbb{T}$ for domain values, tensors or vectors (or lists thereof)
- $\mathbb{E}$ for embeddings, which are vector representations internal to a neural network
- $\mathbb{D}$ for distributions, which are probability distributions over names of API functions. For $d \in \mathbb{D}$, $d(f)$ is the probability of function f.

We will use some auxiliary operators:

- embedding, denoted by $[\![.]\!] : \mathit{encoding}(\mathbb{T}) \to \mathbb{E}$
- concatenation, denoted by $.\#. : \mathbb{T} \times \mathbb{T} \to \mathbb{T}$

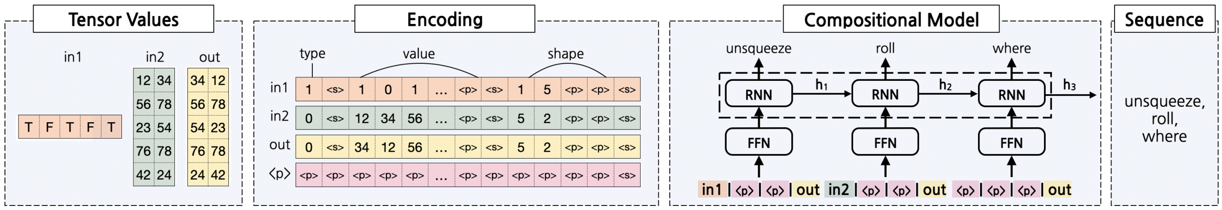


Figure 6.3: Illustration of Compositional Model on an example. The inputs are in the Tensor Values box, and the expected prediction is shown in the Sequence box.

6.4.2 The *encoding* function

Before passing the input and output tensors to the models, we encode them into a fixed-length vector (Figure 6.3-Encoding). We extract three different pieces of information from the tensors: (i) tensor values, (ii) tensor shapes, and (iii) tensor types, and combine them as a sequence separated by a special separator $< s >$, i.e., $\boxed{X = \text{type } \texttt{<s>} \text{ shape } \texttt{<s>} \text{ value}}$, such that, the models can learn from all the three modalities together.

To manage the wide range of tensor values in the model, we normalize the values as follows: we encoded the values greater than 100 into 100, values greater than 1000 into 101, and similarly for the negative values. The intuition is based on how developers recognize patterns: when a value becomes large enough, the importance of the least significant digit decreases in pattern recognition.

Finally, all domain inputs and output encoding are concatenated together. We support up to 3 inputs and one output. Dummy inputs are added when there are less than 3 inputs to keep the model input size the same for all examples.

6.4.3 Compositional Model

We train a model to predict the sequence of API functions $s_f = [f_1, ..., f_n]$, given a task specification $\phi = \{inp, out\}$, where *inp* is a list of input tensors that have gone through the sequence of API operations s_f, and *out* is the final output tensor.

In this description, we assume all functions take two inputs: first, the result of the previous computation, and second, a "local" input, e.g. inp_i here that comes from *inp*. Define $args_i = (f_{i-1}(args_{i-1}), inp_i)$, for $i > 1$, and $args_1 = (\_, inp_1)$. We train a model G, such that for $i = 1..n$:

$$f_i = G([\![args_i, out]\!]) \tag{6.1}$$

The embedding $[\![.]\!]$ of the encoded inputs is obtained using feed-forward networks (FFN), whereas G is rendered by employing recurrent neural networks (RNNs)[18]. The operation of *one* cell of an RNN has the type: $RNN : \mathbb{E}_1 \times \mathbb{E}_2 \to (\mathbb{E}_3, \mathbb{D})$ where,

- $\mathbb{E}_1$ is the hidden state coming from the previous cell, or a zero value;
- $\mathbb{E}_2$ is the embedding of the local input and the final output; we permit each function to have an optional additional input;
- $\mathbb{E}_3$ is the output hidden state being passed to the next cell;

[18] Technically, bi-directional RNNs [224].

- $\mathbb{D}$ is the prediction of API function from this cell; technically it is a distribution from which we take the argmax.

Then,

$$\mathbf{h}_i, d_i = RNN_i(\mathbf{h}_{i-1}, [\![inp_i\#out]\!]) \tag{6.2}$$

where $\mathbf{h_{i-1}}$ and $\mathbf{h_i}$ are incoming and outgoing hidden states, respectively, and d_i the predicted distribution. We expect $f_i = argmax(d_i)$.

Figure 6.3 Compositional Model shows three units of the model for an example. In each unit, the encoding passed to the feed-forward network is similar to the one used before to create $[\![inp_i\#out]\!]$. When f_i needs to use $f_{i-1}(args_{i-1})$, we mask the position as empty (”<p>” in the figure) so that the model exploits $\mathbf{h}_{i-1}$. Embedded encodings are passed to RNN units, and each unit further projects the input embedding into the RNN embedding space to generate h_i, using information flowed from adjacent units, h_{i-1}. Finally, the output of each unit is passed to a softmax layer (not shown here) to produce a probability distribution over API functions.

6.4.4 Synthetic Data Generation

To train a neural model so that it can understand the behavior of API functions, a large number of corresponding input-output pairs is necessary. Unlike other problems exploiting ML models, collecting real-world data from code repositories (e.g., GitHub) is not applicable here because we need runtime values, not static information such as static code. Therefore, we randomly generate input/output values, and use the synthetic dataset for model training.

For each API function, we randomly generate input tensors, run the API functions with them, and capture the corresponding outputs. In other words, we create a set of input/output values in a black-box manner: we do not assume API functions' implementation details or internal behaviors. As it does not require understanding internal program structures, it is easy to generate a large number of input-output pairs without much manual effort and can be easily parallelized.

However, as even a simple API operation in modern libraries (e.g., PyTorch) imposes many constraints, inputting random values will generate many runtime errors due to the constraints violations. To reduce such errors, we exploit API specification, and generate a set of inputs with the valid combinations (see Listing 6.1 for an example of generating data points for `torch.sum`). By excluding invalid combinations of arguments to each API function, we can speed up the data generation and generate a large synthetic dataset that can capture API function input/output benign behavior.

Listing 6.1: Example data generation code for `torch.sum`

```
def generate_sum_IO():
    in_tensor, tensor_size = random_tensor()
    dim = random_dimension(0, tensor_size)
    if dim == len(tensor_size):
        out_tensor = torch.sum(in_tensor)
    else:
        out_tensor = torch.sum(in_tensor, dim)
    return (in_tensor, out_tensor)
```

6.5 Incorporating ML in Enumerative Synthesis

Here we formally describe how the ML models were incorporated into the enumerative synthesis. Please refer to Figure 6.1 and Section 6.1 for a walk-through of these on an example. A detailed description of our implementation and the pseudo code for each synthesis approach can be found in Appendix B.

Basic enumerative synthesis. As a baseline, we implement an enumerative synthesizer without any ML models. Basic enumerative search starts with a set of base values and enumerates over combinations of operations and the values.

The list of base values includes `inp`, other basic constants such as `0`, `1`, `-1`, or heuristically-chosen values such as the dimensions of the given variables (e.g., `3`). Then, starting with the base values, the search enumerates ways of applying operations to previously-explored values and expands the set of known values. There are various ways of iterating the operations and the values (e.g. based on syntactic size as in Transit [259]), but we use weighted enumerative search, which is the approach of TF-Coder [229]. It does so in the order of increasing *cost*. Operations and values are assigned costs based on their complexity: less common and more complex operations are assigned higher costs, and the common and simple operations are assigned lower costs. Costs are additive, so common operations and simpler expressions are explored earlier. The costs are manually set by the synthesizer developers, but only needed to be set once, and it will be used for all tasks.

Prioritizing likely functions with an ML model. As the needed operations for a specific problem are not known to the synthesizer ahead of time, the costs seeded in it will not always be ideally suited for all problems. TF-Coder [229] and DeepCoder [24] address this problem using an ML model to re-weigh all operations before the enumerative search starts. Given a task specification (i.e., input/output examples), it invokes a multi-label classification model to predict the probability of each needed operation and re-weighs them accordingly with the goal of encountering the needed operations earlier in the search. We trained a multi-label classification model following DeepCoder [24].

Compositional Model - Full-Sequence. In this mode, the compositional model predicts a sequence of API functions $s_f = [f_1, f_2, ..., f_n]$ given the final output *out*, and the inputs to each API function $[inp_1, inp_2, .., inp_n]$[19]. The synthesizer invokes the compositional model with the specification, predicts a sequence of operations, and searches only the parameter values (e.g., *dimension*) that were not provided in the specification.

The Full-Seq mode completely bypasses enumerative search over operations. Instead, the compositional model predicts the API functions needed in a synthesis instance as well as the order of those APIs in the synthesized code. Thus, the synthesizer does not need to search the operation space, but only needs to search the combinations of base values.

Compositional Model - First-Of-Sequence. In the First-Of-Seq mode, given an input and a final output, the compositional model predicts the most probable API function needs to come in the sequence. As enumerative search keeps track of the intermediate output value, we can iteratively invoke a compositional model, and compute the intermediate values using the predicted API functions, which can be used to predict the next API function.

[19] As the program synthesis task specification only provides an order agnostic list of inputs, the synthesizer needs to search through different combinations of them to generate *a list of input tensors to each API call* to invoke the compositional model. In this section, we assume that the list of input tensors to each API call is provided.

6.6 Evaluation

In this section, we first describe the training and evaluation dataset (Section 6.6.1), and evaluate the trained API function sequence prediction model (Section 6.6.2). Then, we investigate the prediction-guided synthesis (Section 6.6.3). Finally, we show the generalizability and the compositional property of our model (Section 6.6.4).

6.6.1 Dataset

Program synthesis benchmarks. We evaluated the effectiveness of our approaches with a subset of TF-Coder's SO benchmarks [229]. These benchmarks contain 50 tensor manipulation examples collected from SO, each containing input and output tensor values and the desired solutions in Tensorflow. To evaluate our approach that supports PyTorch, we first translated them into PyTorch and excluded tasks that we could not translate by hand. Among the 33 API functions needed for the remaining 36 benchmarks, we selected 16 functions covering 18 benchmarks (Table 6.2-Stack Overflow) from the core utility that modifies values (e.g., `add`) or shapes (e.g., `transpose`) of tensors, create them, or manipulate them in similar ways. These operations were chosen because the model can clearly observe the behavior of each API function solely from input and output pairs (i.e., no side effects). The full benchmarks we support are available in Appendix B.

Table 6.2: Statistics of the dataset used in this study. Numbers in parentheses indicate the length of the sequences.

	Synthetic Dataset			Stack Overflow
	Train	Valid.	Test	Test
# of unique seqs (len)		16 (1) + 186 (2)		8 (1) + 7 (2)
# of in/out values	5.5M	10K	10K	18

Synthetic data generation. To train our sequence prediction model that work for the SO benchmarks, we synthesized a dataset as per Section 6.4.4. We synthesized 202 unique API functions sequences by using the exhaustive combination of 16 API functions, with 1 or 2-length sequences. From the 272 (16 + 16*16) possible sequences, 70 sequences were removed due to the constraints.

For each API function sequence in the training dataset, say f_1, f_2, f_3, we ran f_1 with randomly generated input and other parameter values (e.g., dimension, mode, etc.). Then, f_2 takes f_1's output as input and takes other random input tensors, if necessary. We treat f_3 similarly by propagating f_2's output.

To generate diverse and unbiased input-output pairs, we cover different properties of the functions, such that the model can explore the broad data space of input-output pairs.

It took 1-person week to encode the API specifications to write valid data generation code by reading the PyTorch documentation. To avoid expansion to large input values and to let the model learn the patterns sufficiently, we used a fixed range of values (from 0 to 20) and the size of tensors (up to 3 dimensions, and up to 5 elements in each dimension), to prevent the tensors from being dispersed too much.

We created the dataset with 100,000 input-output pairs for each unique API sequence (Table 6.2-Synthetic). We split the dataset into training, validation, and test sets. The training, validation, and test sets included all 202 API sequences, but the input/output values were not overlapped across the datasets.

6.6.2 Sequence Prediction Model

We trained both Full-Seq and First-Of-Seq variants using the training set of the synthetic data, and evaluated it with (1) the test set of the synthetic data, and (2) SO benchmarks.

Table 6.3: Model accuracy for unseen input/output values.

	Synthetic-Test	Stack Overflow	
Model	Top-1	Top-1	Top-3
Full-Seq	79.36%	35.29%	76.47%
First-Of-Seq	66.88%	52.38%	76.19%

Observation. Table 6.3 shows the result. The model's top-1 testing accuracies of the 10K synthetic test set are ∼79%. Among 18 SO benchmarks, the Full-Seq model found 13 sequences are in the top-3 (72.22%), and among them 6 are in the top-1 (33.33%). In comparison to the Full-Seq model, the First-Of-Seq model's top-1 accuracy is better. This is not surprising as First-Of-Seq model has more information (actual values of the intermediate inputs) than the Full-Seq variant. However, surprisingly, the top-3 accuracies of both are almost similar. These results indicate that the compositional model perhaps learned a representation of the intermediate states of the API operation sequence: even without passing the true intermediate values, the Full-Seq model behaves at per with the Full-Seq model at top-3.

6.6.3 Prediction-guided enumerative synthesis

Using the trained Full-Seq and First-Of-Seq variants, we first evaluate our approach, against vanilla enumerative synthesis (similar to [229]). We further compared with a multi-label prediction model, a setting inspired by DeepCoder. [24]. Table 6.4 shows the results.

Existing synthesizers vs. compositional model. Among the 18 tasks, both vanilla enumerative search and a synthesizer prioritized with a multi-label classification model could synthesize all tasks. The new variants incorporating our models synthesized 17 (First-Of-Seq) and 14 (Full-Seq) correctly, out of 18 and 17 respectively. However, although they synthesized fewer solutions, they required less time to synthesize the solutions: 5.87 seconds (First-Of-Seq)

Table 6.4: End-to-end program synthesis results; our models in **bold**. Time, Max, and Median show the average, max, and median synthesis time of found programs.

			Time		
	Found	Not Found	Mean	Max	Median
Enumerative	18	0	10.01	96.53	0.46
Multi-label	18	0	7.44	77.00	0.32
First-Of-Seq	17	1	5.87	59.93	0.39
Full-Seq	14	4	1.04	9.58	0.25

Table 6.5: Model accuracy for unseen input/output values, trained with a dataset covering all SO benchmarks.

Model	Synthetic-Test Top-1	Stack Overflow Top-1	Stack Overflow Top-3
Full-Seq	88.15%	68.57%	91.42%
First-Of-Seq	65.44%	51.61%	79.03%

and 1.04 seconds (Full-Seq) on average, whereas the existing synthesizers took 10.01 and 7.44 respectively.

We see this speed up because predicting the sequence reduces the search space. As the compositional models return a sequence of API functions, the enumerative search can focus on the argument values instead of iterating over the API function sequences. Note that the multi-label classification model also suggests potential API functions, but instead of function sequences, it provides us with a set of functions. Thus, in the worst case, the associated enumerative search has to explore all the possible combinations increasing the synthesis time.

The difference in synthesis time between the compositional models and the baselines is not big in simple tasks (e.g., `any(in,-1))`, which is why the difference in median in Table 6.4 is not significant enough. However, when it comes to more complex tasks like in Figure 6.1, the difference becomes significant: 54.79 seconds with plain enumerative synthesis vs. 0.49 with First-Of-Seq mode.

One caveat of compositional models is that the model prediction is not the bottom-up method but a one-shot approach. Therefore, when the model fails to predict the sequence correctly, it cannot synthesize the program. However, as the whole search can be done quickly, the time overhead is not high even when one tries with the compositional model and employs other approaches once it fails.

First-Of-Seq vs. Full-Seq. Between the two variants, First-Of-Seq was able to synthesize more programs. For example, First-Of-Seq successfully synthesized the desired program `where(lt (in,1),in,1)` by predicting `lt` and `where` correctly in top-3. However, Full-Seq failed, and predicted [[`eq`,`where`], [`eq`,`mul`], [`gt`,`where`]] as top-3, which are close enough, but not entirely correct. This is expected. First-Of-Seq only has to get the first element of the sequence right; the next element is in fact the first element of the result of the *subsequent* prediction, which in turn, is based on the actual intermediate value computed by the first function predicted. Whereas, the Full-Seq gets only one chance to get the entire sequence right without knowing the intermediate values. When a sequence is correctly predicted, Full-Seq model could synthesize the solutions faster; it only needs to be invoked once, without the intermediate values computation.

6.6.4 Evaluating Generalization

To see whether the model truly learned the functionality of the API functions and learned their compositions, we tested whether the model can generate new API sequences that were not present in the training data. From the original synthetic dataset 6.2, we removed the data of 7 API function sequences with length-2 that were included in Stack Overflow benchmarks, and trained the First-Of-Seq and Full-Seq models. Our hypothesis was that if the models are able to learn the compositional property, instead of learning the distributions of sequences, they should be able to generate unseen sequences by composing API functions into a sequence.

Observation. Not surprisingly, the accuracy drops from the Section 6.6.2 result. Nevertheless, out of 8 benchmarks with 2 sequences, we can still predict 4 sequences at top-5 (50% accuracy) with Full-Seq, and 6 sequences (75% accuracy) with First-Of-Seq. In this setting, we sometimes narrowly miss some function sequences. For example, we miss a benchmark [`lt, where`], however it predicts [`eq, where`], and [`gt, where`] instead. Note that `gt` and `eq` have very similar functionalities to the intended API function `lt`. Both the models can correctly predict sequences like [`unsqueeze, eq`], [`matmul, add`], etc. As expected, the First-Of-Seq model works much better than Full-Seq model.

To further check the model's ability to generalize to unseen 3-length sequence, we randomly picked 71 unique 3-length sequences made out of 16 API functions and collected 100 instances of them with different input/output values. This gives a total of 7100 test samples. We used the model trained with only sequences with length 2. Overall, at top-5, model's accuracy is ∼34% when queried with unknown sequences and unknown values. However, the model can predict 69 out of 71 sequences correctly at least with one input-output. The only two sequences the model missed are [`add, mul, any`] and [`add, unsqueeze, ne`]. In contrast, [`where, expand, matmul`] was predicted correctly around 97% time. These results indicate the model's ability to generalize.

6.6.5 Scaling to a larger set of API functions

First-Of-Seq vs. Full-Seq. We created a dataset covering all 36 SO benchmarks (1- to 3-length) in section 6.6.1 with 33 API functions (up from 16 before). Synthesizing training data covering all exhaustive combinations of the 33 API functions, up to 3-length sequences, gives us a huge number of combinations (33x33x33=35,937). Generating the corresponding training data (>35B samples) and training the model accordingly is challenging. Therefore, as a start, we selected 65 unique sequences that can be a solution to one of the 36 benchmarks.

Observation. Table 6.5 shows the result. Compared to the model trained with the exhaustive combinations of 16 API functions (Section 6.3), the Full-Seq model trained with this dataset actually performed even better for (91.42% top-3 accuracy in SO benchmark data). We conjecture that this is because the model could focus its learning on the semantics of API function sequences that are more likely to be in the test set, rather than learning the semantics of all combinations. Unlike the accuracy difference in Full-Seq models, First-Of-Seq variant achieved slightly lower top-1 accuracy with this dataset. This is because the number of API functions it needs to learn has increased from 16 to 33. For First-Of-Seq, as it predicts each API function independent from others, it could not benefit from having a training set with the targeted sequences.

Generalizability. We also tested the generalizability of our model when it is trained with a dataset covering all 33 API functions needed to synthesize the solutions for 36 SO benchmarks. Due to the aforementioned challenge, we could not synthesize a dataset like in Section 6.6.4; instead, we synthesized a dataset with 598 random combinations of 1 or 2-length sequences of 33 API functions, that has not appeared in SO benchmark.

Observation. 7 out of 18 (39%) two-length unseen sequences were predicted correctly with Full-Seq, and the first API of a sequence was correctly predicted by First-Of-Seq with 69% accuracy. They both achieved lower accuracy than the ones trained with the exhaustive combinations, as this model had fewer sequences to learn from, which is critical in generating unseen sequences.

6.7 Why composition works?

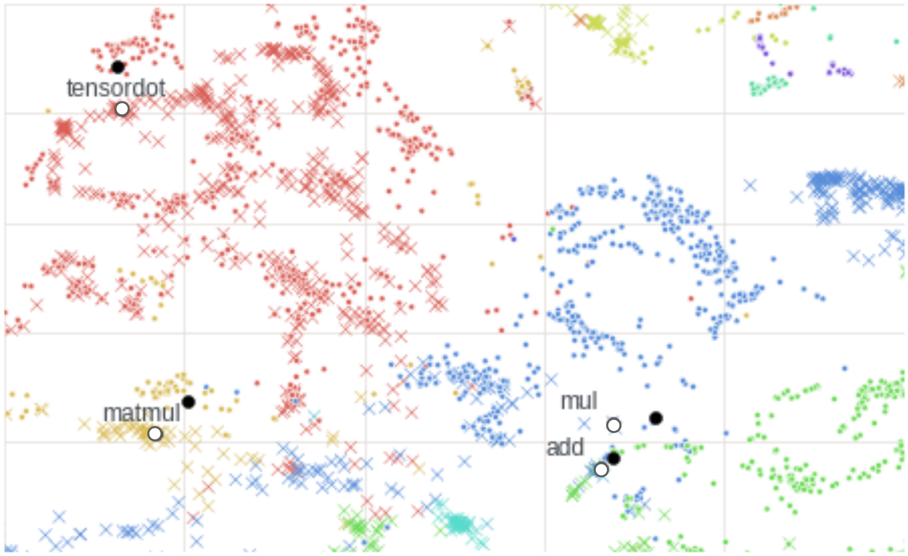


Figure 6.5: Proximity of h_2 and h'_2 pairs for some inputs (in white and black respectively), against a backdrop of h_2 (crosses) and h'_2 (dots).

We show that a unit of our compositional model has an interesting property: it learns to convert its incoming hidden vector to its outgoing hidden vector in a way consistent with the *semantics* of the API function it predicts, albeit in embedding space. This property is crucial for predicting a sequence compositionally.

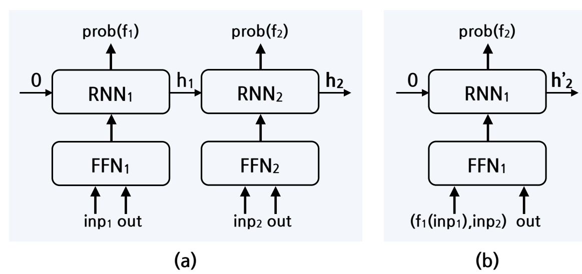


Figure 6.4: Illustration of compositional learning: $h_2 \approx h'_2$.

In Fig 6.4(a), we show two units of the compositional model, where the first one predicts function f_1, on the basis of inp_1, out and the previous hidden vector, if any. That unit also produces a hidden vector h_1. The second unit produces a hidden vector h_2. It may also consume further local input (such as inp_2). Fig 6.4(b) shows an alternate situation in which we give the *result* of $f_1(inp_1)$ directly as input to the first unit, which then produces h'_2.

The interesting property is that h_2 and h'_2 are close together in the representational space. Figure 6.5 shows a part of a tSNE plot of h_2 (crosses) and h'_2 (dots), as described above, for inputs drawn from our benchmarks. As we expect, black and white markers in the figure show that $h2$ and $h2'$ for the same inputs are arranged close to each other in the embedding space. In producing h_2 (or h'_2), the RNN unit did not care whether it was given h_1, the representation produced by the previous RNN unit, or directly given $f_1(inp_1)$. In this manner, successive hidden states contain information analogous to the results of concrete computations: $f_1(inp_1)$, $f_2(f_1(inp_1))$, and so on. (In the actual model, these functions need not be unary, as implied here.)

Composition property formally. Refer to the notation introduced in Sec 6.4.3. If instead of h_{i-1}, RNN_i is given (the embedding of) the concrete intermediate value computed by the computation so far, its observed behavior is the same in both cases. Formally,

$$\begin{aligned} \mathbf{h}_i, d_i &= RNN_i([\![f_{i-1}(args_{i-1})\#out]\!], [\![inp_i\#out]\!]) \\ &\approx RNN_i(\mathbf{h}_{i-1}, [\![inp_i\#out]\!]) \end{aligned} \tag{6.3}$$

Thus, the hidden vectors capture an abstraction of intermediate values that would have arisen in concrete computation $f_1(i_1), f_2(f_1(i_1))$, and so on. The composition takes place as each unit of RNN makes a local decision based on incoming hidden vector, which is set up to capture the result of the corresponding concrete computation so far.

6.8 Limitations

Our results are promising, yet preliminary in many ways, and we have not established generality in several dimensions. First, we support a small set of API functions and have carried out a limited evaluation. As the number increases, the training data size also increases, and training the model well becomes harder due to computational needs. The robustness of training is a challenge in general.

Second, the model's ability to generalize to unseen sequences is crucially dependent on training over a broad diversity of API sequences. This is challenging as we go to a larger number of API functions, because we cannot cover all permutations exhaustively. However, as seen in Section 6.6.5, the model can still learn the semantics reasonably well if the training data covers the sequences in the test set. Thus, the future works may benefit from creating a training dataset containing a distribution of sequences representing the real-world API usage patterns, through API usage mining [174, 281].

Third, we have explored the model's training and inference on relatively short tensors, with small data ranges, and have generally worked only with integer data. In a real application, tensors can be out-of-distribution with respect to the model.

Fourth, we have worked only with PyTorch. We believe the work can be replicated easily to NumPy and Tensorflow API functions, because of their similar nature (acting over arrays of numbers.). Farther out, we may need to invent additional techniques. For example, Pandas is designed to deal with records (or "dataframes") containing labeled axes (rows and columns); so it supports not only numeric manipulation, but also label-based slicing. Here we may be able to borrow some insights from the Autopandas work [28]. Finding ways to deal with the above threats is in our future work.

6.9 Summary

By building a language model and training it with a large dataset generated by fuzzing, we could confirm that it is feasible to use the generation-based approach to provide information support that is suited to the user context.

Chapter 7

Generation-based Information Support Considering Developer's Task Context[20]

In previous studies, it has been established that developers' characteristics are related to their usage of documentation, which serves as a proxy for their information needs. We also confirmed that it is viable to use a generation-based approach for information support. The main focus of this chapter was to assess the feasibility of utilizing this approach while taking the developer context into account.

7.1 Introduction

Building and maintaining software systems requires a deep understanding of a codebase. Consequently, developers spend a significant amount of time searching and foraging for the information they need and organizing and digesting the information they find [117, 118, 125, 151, 161, 196]. Understanding code, however, is a challenging task; developers need to assimilate a large amount of information about the semantics of the code, the intricacies of the APIs used, and the relevant domain-specific concepts. Such information is often scattered across multiple sources, making it challenging for developers, especially novices or those working with unfamiliar APIs, to locate what they need. Furthermore, much of the relevant information is inadequately documented or spread across different formats and mediums, where it often becomes outdated.

With the growing popularity of large language model (LLM) based code generation tools [2, 4, 6], the need for information support for code understanding is arguably growing even higher. These tools can generate code automatically, even for developers with limited coding skills or domain knowledge. This convenience comes at a cost, however – developers may receive code they don't understand [106, 285]. Indeed, early research on LLM code generation tools has found that developers have a harder time debugging code generated by the LLM and easily get frustrated [138, 261].

Fortunately, LLMs also provide an opportunity in this space, namely by offering on-demand *generation-based information support* for developers faced with unfamiliar code. Compared to

[20]This chapter is adapted from Nam *et al.* [176]

general web search queries [270], LLM prompts can allow developers to provide more context, which can enable them to receive information that more precisely aligns with their specific needs, potentially reducing the time spent on sifting through the information obtained from the web to suit their particular requirements. Developers have indeed taken to web-hosted conversational LLM tools, such as ChatGPT, for programming support en masse, but this setup requires them to both context switch and copy the relevant context from their IDEs into the chat system for support.

To explore the potential for generation-based information support directly in the developer's programming environment, we developed a prototype in-IDE LLM information support tool, GILT (Generation-based Information-support with LLM Technology). GILT is capable of generating on-demand information while considering the user's local code context, which we incorporate into the prompts provided to the LLM behind the scenes. This way, we also introduce a novel interaction method with the LLM, *prompt-less interaction*. This option aims to alleviate the cognitive load associated with writing prompts, particularly for developers who possess limited domain or programming knowledge.

As there is still little knowledge about how to best use an LLM for information support (as opposed to just code generation), we evaluate the effectiveness of our prototype tool in an exploratory user study with 32 participants tasked with comprehending and extending unfamiliar code that involves new domain concepts and Python APIs for data visualization and 3D rendering – a challenging task. Our study quantitatively compares task completion rates and measures of code understanding between two conditions – using the LLM-backed assistant in-IDE versus directly searching the web in a browser – and qualitatively investigates how participants used the tools and their overall satisfaction with this new interaction mode. Concretely, we answer three research questions:

- RQ7.1: To what extent does GILT affect developers' understanding, task completion time, and task completion rates when faced with unfamiliar code?

- RQ7.2: How do developers interact with GILT, and to what extent does that differ between the participants?

- RQ7.3: How do developers perceive the usefulness of GILT?

Our results confirm that there are statistically significant gains in task completion rate when using GILT, compared to a web search, showing the utility of generation-based information support. However, we did not find the utility gains in terms of time and understanding level, leaving room for further improvement. We also discovered that the degree of the benefit varies between students and professionals, and investigated potential reasons behind this.

7.2 The GILT Prototype Tool

We iteratively designed GILT to explore different modes of interaction with an LLM for information support. GILT is a plugin for the VS Code IDE (Figure 7.1) that considers user context (the code selected by the user) when querying an LLM for several information support applications.

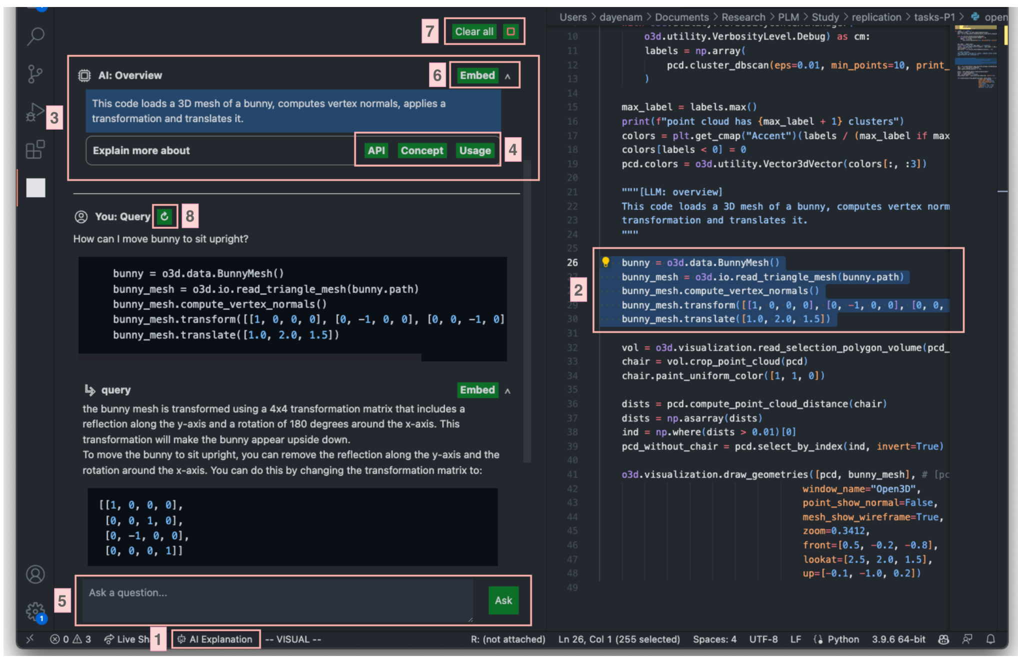


Figure 7.1: Overview of our prototype. (1) A trigger button; (2) code used as context when prompting LLM; (3) code summary (no-prompt trigger); (4) buttons for further details; (5) an input box for user prompts; (6) options to embed information to code (Embed) and a hide/view button; (7) options to clear the panel (Clear all) and an abort LLM button; (8) a refresh button.

7.2.1 Interacting with GILT

There are two ways to interact with the plugin. First, users can select parts of their code (Figure 7.1-②) and trigger the tool by clicking on "AI Explanation" on the bottom bar (Figure 7.1-①), or using "alt/option + a" as a shortcut, to receive a summary description of the highlighted code (`Overview`). They can then explore further by clicking on buttons (Figure 7.1-④) for API (`API`), domain-specific concepts (`Concept`), and usage examples (`Usage`), which provide more detailed explanations with preset prompts. The `API` button offers detailed explanations about the API calls used in the code, the `Concept` button provides domain-specific concepts that might be needed to understand the highlighted code fully, and the `Usage` button offers a code example involving API calls used in the highlighted code.

Users can also ask a specific question directly to the LLM via the input box (Figure 7.1-⑤). If no code is selected, the entire source code is used as context (`Prompt`); alternatively, the relevant code highlighted by the user is used (`Prompt-context`). The model will then answer the question with that code as context. GILT also allows users to probe the LLM by supporting conversational interaction (`Prompt-followup`). When previous LLM-generated responses exist, if a user does not highlight any lines from the code, the LLM generates a response with the previous conversion as context. Users can also reset the context by triggering the tool with code highlighted, or with the Clear all button.

7.2.2 Our Design Process and Decisions

Focus on understanding. We intentionally did not integrate a code generation feature in the prototype as we wanted to focus on how developers *understand* code.

In-IDE extension. Besides anticipating a better user experience, we designed the prototype as an in-IDE extension to more easily provide the code context to the LLM – participants could select code to use as part of the context for a query.

Pre-generated prompts. We designed buttons that query the LLM with pre-generated prompts (*prompt-less* interaction) to ask about an API, conceptual explanations, or usage examples, as shown in Figure 7.1-④. We chose these based on API learning theory [62, 126, 158, 245], expecting this may particularly assist novice programmers or those unfamiliar with the APIs/domains or the LLM, as writing efficient search queries or prompts can be difficult for novices [59, 64, 116]. At the same time, we also expected that this could reduce the cognitive burden of users in general in formulating prompts.

For `Overview` and the buttons `API`, `Concept`, `Usage`, we developed prompt templates after a few iterations. To more efficiently provide the context to LLM, we used the library names and the list of API methods included in the selected code, such as "Please provide a [library name] code example, mainly showing the usage of the following API calls: [list of API methods]" for `Usage`.

Unrestricted textual queries. Users can also directly prompt the LLM (Figure 7.1-⑤), in which case GILT will automatically add any selected code as context for the query. Internally, the tool adds the selected code as part of the user prompt using pre-defined templates, and requests the LLM to respond based on the code context.

Need-based explanation generation. The tool is pull-based, i.e., it generates an explanation only when a user requests it. Similar to many previous developer information support tools, we wanted to reduce information overload and distraction. We expect that if and when enough context can be extracted from the IDE, hybrid (pull + push) tools will be possible, but this would require more research.

Iterative design updates. We ran design pilot studies and updated our prototype accordingly. For example, we made the code summary as the default action for the tool trigger with code selection, after seeing pilot participants struggling to find parts of code to work on due to their unfamiliarity with libraries and domains. We updated the prompt-based interaction with the LLM to support a conversational interface, based on the pilot participants' feedback that they wanted to probe the model based on their previous queries to clarify their intent or ask for further details. Finally, we opted to use GPT-3.5-turbo instead of GPT-4 as planned, after discovering that the response time was too slow in the pilot studies.

7.3 Human Study Design

Participants. We advertised our IRB-approved study widely within the university community (through Slack channels, posted flyers, and personal contacts) and to the public (through Twitter, email, and other channels). We asked each participant about their programming experience and screened out those who reported their experience level as "not at all". We did not ask about their professional programming experience, as the target users of our information support tools are not limited to professional developers. To minimize the possibility of participants knowing solutions, we specifically sought out participants who had not used the libraries included in our study. We accepted participants into the study on a rolling basis to capture a range of programming experience and existing domain knowledge. We compensated each participant with a $25 Amazon Gift card. We recruited 33 participants and conducted 33 studies in total. However, we had to exclude data from one participant from the analysis because they did not follow the instructions for using the extension. Therefore, we report on 9 women and 23 men participants. Among them, 16 participants identified themselves primarily as students, 1 as software engineer, 2 as data scientists, and 13 as researchers. In the analysis, we divided the participants into two groups (students v.s., professionals) based on this. 24 participants had experience with ChatGPT, 15 with Copilot, 5 with Bard, while 7 participants reported no prior use of any LLM-based developer tools. In terms of familiarity with such tools, 14 participants stated that they have either used AI developer tools for their work or always use them for work.

Tasks. The tasks were designed to simulate a scenario in which developers with specific requirements search the web or use existing LLMs to generate code and find similar code that does not precisely match their needs. For each task, we provided participants with a high-level goal of the code, start and goal outputs, and a starter code file loaded into the IDE. In this way, participants had to understand the starter code we provided and make changes to it so that the modified code met the goal requirements. Each task consisted of 4 sub-tasks, to help reduce participant's overhead in planning, as well as to measure the task completion ratio in the analysis. There were some subtle dependencies between the sub-tasks, so we advised participants to follow the order we gave, but they were free to skip. Completing each sub-task required a small change, ranging from a single parameter value update to an addition of a line of code. The difficulty levels of the sub-tasks varied, but we intentionally designed the first sub-task to be easy so that participants could have at least one success easily. Sub-tasks also came with start and goal outputs and descriptions (see Figure 7.2). We did not include tasks that required strong programming

Table 7.1: An example code context and the information that can be provided by our tool given the code.

Code Context	
Selected code	bunny = o3d.data.BunnyMesh() bunny_mesh = o3d.io.read_triangle_mesh(bunny.path) bunny_mesh.compute_vertex_normals() bunny_mesh.transform([[1, 0, 0, 0], [0, −1, 0, 0], [0, 0, −1, 0], [0, 0, 0, −1]]) bunny_mesh.translate([2.5, 2.1, 1.2])
One-click Prompting	
Overview	This code loads a mesh of a bunny, computes its vertex normals, applies a transformation matrix to invert and translate it.
API	- The first line of code loads the bunny mesh from the open3d data module. - The second line reads the triangle mesh of the bunny from the file path. - The third line computes the vertex normals of the bunny mesh, which are used to determine the orientation of each vertex with respect to the surface of the mesh. - The fourth line applies a transformation matrix to the bunny mesh using the `transform` function. The matrix used in this case is a 4x4 matrix that performs a reflection along the x-axis, followed by a reflection along the y-axis, and finally a reflection along the z-axis. This effectively inverts the bunny mesh. - The fifth line applies a translation to the bunny mesh using the `translate` function. The translation vector used in this case is [2.5, 2.1, 1.2], which moves the bunny mesh 2.5 units along the x-axis, 2.1 units along the y-axis, and 1.2 units along the z-axis.
Concept	Here are 3 domain concepts that might be useful to know. - Mesh: A mesh is a collection of vertices, edges, and faces that define the shape of a 3D object. In open3d, a mesh is represented by a TriangleMesh object. - Vertex Normals: Vertex normals are vectors that are perpendicular to the surface of a mesh at each vertex. They are used to determine how light interacts with the surface of the mesh. - Transformation Matrix: A transformation matrix is a mathematical matrix that is used to transform a 3D object. In open3d, a transformation matrix is represented by a 4x4 matrix, where the first three columns represent the rotation and scaling of the object, and the last column represents the translation.

Table 7.1: An example code context and the information that can be provided by our tool given the code (cont.).

<table>
<tr><td>Usage</td><td>Here is an open3d code example.

<pre>
import open3d as o3d

# Load a mesh of a bunny
bunny = o3d.data.BunnyMesh()
bunny_mesh = o3d.io.read_triangle_mesh(bunny.path)

# Compute vertex normals
bunny_mesh.compute_vertex_normals()

# Apply transformation matrix to invert and translate the bunny mesh
transformation_matrix = [[1, 0, 0, 0], [0, −1, 0, 0], [0, 0, −1, 0], [0, 0, 0, −1]]
bunny_mesh.transform(transformation_matrix)
bunny_mesh.translate([2.5, 2.1, 1.2])

# Visualize the transformed bunny mesh
o3d.visualization.draw_geometries([bunny_mesh])
</pre>

This code loads a mesh of a bunny, computes its vertex normals, applies a transformation matrix to invert and translate it, and finally visualizes the transformed bunny mesh.</td></tr>
<tr><td colspan="2">Query Prompting</td></tr>
<tr><td>Query</td><td>How can I move bunny to sit upright?</td></tr>
<tr><td>Response</td><td>In this code, the bunny mesh is transformed using a 4x4 transformation matrix. To move the bunny to sit upright, you can apply a rotation transformation to the mesh. For example, you can rotate the mesh 90 degrees around the x-axis by using the following transformation matrix:

<pre>
[[1, 0, 0, 0],
 [0, 0, 1, 0],
 [0, −1, 0, 0],
 [0, 0, 0, 1]]
</pre>

You can apply this transformation to the bunny mesh using the <code>transform</code> method:

<pre>
bunny_mesh.transform([[1, 0, 0, 0], [0, 0, 1, 0], [0, −1, 0, 0], [0, 0, 0, 1]])
</pre>

This will rotate the bunny mesh 90 degrees around the x-axis, making it sit upright.</td></tr>
</table>

knowledge, because our goal was to assess how well participants could understand the code. For the same reason, we provided participants with a starter code that was runnable and bug-free.

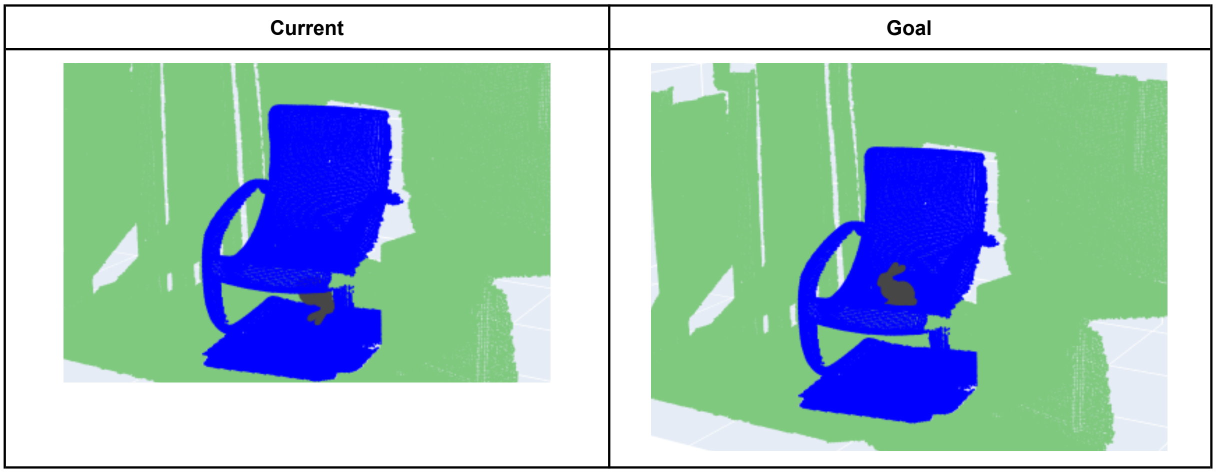


Figure 7.2: A 3D-rendering example sub-task (open3d-3). With these start and goal outputs, we asked the participants to "Make the bunny sit upright on the chair." See Figure 7.1 for the corresponding starter code and the tool output.

Our tasks cover both a common and less common domain that a Python developer might encounter in the wild. We chose two domains: data visualization and 3D rendering. These two tasks also allowed participants to easily check their progress, as they produced visual outputs that were comparable with the given goal. For the data visualization task, we used the Bokeh [1] library and asked participants to edit code that visualizes income and expenses data in polar plots. Understanding this code required knowledge of concepts related to visualizing data formats, marks, and data mapping. In the 3D rendering task, we used the Open3d [5] library. This task required knowledge of geometry and computer graphics. Participants were asked to edit code that involved point cloud rendering, transformation, and plane segmentation.

When selecting libraries, we intentionally did not choose the most common ones in their respective domains, to reduce the risk of participants knowing them well. Choosing less common libraries also helped reduce the risk of an outsized advantage of our LLM-powered information generation tool. Responses for popular libraries can be significantly better than those for less commonly used ones, as the quality of LLM-generated answers depends on whether the LLM has seen enough relevant data during training.

The Bokeh starter code consisted of 101 LOC with 11 (6 unique) Bokeh API calls, and the starter code for the Open3D task consisted of 43 LOC with 18 (18 unique) Open3D API calls. The tasks were designed based on tutorial examples in each API's documentation. In the starter codes, we did not include any comments in the code to isolate the effects of the information provided by our prototype or collected from search engines. All necessary API calls were in the starter code so participants did not need to find or add new ones.

In the task descriptions, we tried to avoid using domain-specific or library-specific keywords that could potentially provide participants with direct answers from either search engines or GILT. For instance, we opted to use "make the bunny...", instead of "transform the bunny..." which may have steered participants towards the function `transform` without much thought.

The full task descriptions, starter code, solution for the demo, and the actual tasks are available in replication package [178].

Experimental Design. We chose a within-subjects design, with participants using both GILT (treatment) and a search engine (control) for code understanding, but they did so on different tasks. This allowed us to ask participants to rate both conditions and provide comparative feedback about both treatments.

The control-condition participants were not allowed to use our prototype, but they were free to use any search engine to find the information they needed. The treatment-condition participants were encouraged to primarily use our prototype for information support. However, if they could not find a specific piece of information using our prototype, we allowed them to use search engines to find it. This was to prevent participants from being entirely blocked by the LLM. We expected that this was a realistic use case of any LLM-based tool, but it rarely happened during the study. Only 2 participants ended up using search engines during the treatment condition, but they could not complete the tasks even with the search engines.

We counterbalanced the tasks and the order they were presented to participants to prevent carryover effects, resulting in four groups (2 conditions x 2 orders). We used random block assignments when assigning participants to each group. Participants were assigned to each group to balance the self-reported programming and domain experience (data visualization and 3D rendering). For every new participant, we randomly assigned them to a group that no previous participant with the same experience level had been assigned. If all groups had previous participants with the same experience level, we randomly assigned the participant to one of them.

Study Protocol. We conducted the study via a video conferencing tool and in person, with each session taking about 90 minutes; in-person participants also used the video conferencing tool, for consistency. At the beginning of the study, we asked participants to complete a pre-study survey, collecting their demographic information, background knowledge, and experience with LLMs. We also estimated their general information processing and learning styles using a cognitive style survey [92] categorizing participants into two groups per dimension: comprehensive / selective information processing and process-oriented learning / tinkering. The participants were then asked to join our web-based VS Code IDE hosted on GitHub CodeSpaces [3], which provided a realistic IDE inside a web browser with edit, run, test, and debug capabilities, without requiring participants to install any software locally [58]. We then showed them a demo task description and explained what they would be working on during the real tasks. Before their first task in the treatment condition, we provided a tutorial for our plugin using the demo task, introducing each feature and giving three example prompts for the LLM. For the control condition, we did not provide any demo, as we expected every participant to be able to use search engines fluently. For each task, we gave participants 20 minutes to complete as many sub-tasks as they could. During the task, we did not use the think-aloud protocol because we wanted to collect timing data. Instead, we collected qualitative data in the post-survey with open-ended questions. We also collected extensive event and interaction logs during the task. After each task, we asked participants to complete a post-task survey to measure their understanding of the provided code and the API calls therein. At the very end, we asked them to complete a post-study survey where we asked them to evaluate the perceived usefulness and perceived ease of use of each code understanding approach and each feature in GILT. We based our questionnaire on the Technology Acceptance Model (TAM) [130], NASA Task Load Index (TLX) [95], and pre- and post-study questionnaires that were previously used in similar studies [219, 245].

We conducted 33 studies in total, with 33 participants. The initial 18 studies were conducted on a one-on-one basis, while some studies in the latter half (involving 15 participants) were carried out in group sessions, with two to five participants simultaneously and one researcher serving as the moderator. We took great care to ensure that participants did not interrupt each other or share their progress. As mentioned before, we excluded one participant's data and used 32 participants' for the analysis. We discovered this issue after the study, as this participant was part of the largest group session (with five participants).

7.4 RQ 7.1: Effects of GILT

In this section, we report on the effectiveness of using GILT in understanding unfamiliar code.

7.4.1 Data Collection

Code understanding. To evaluate the effectiveness of each condition, we used three measurements: (1) Task completion time: to complete each sub-task; (2) Task progress: we rated the correctness of the participants' solution to each sub-task and measured how many sub-tasks they correctly implemented; and (3) Understanding level: we cross-checked participants' general understanding of the starter code by giving them sets of quiz questions about the APIs included in the starter code. Each set contained three questions, requiring an in-depth understanding of the functionalities of each API call and the application domains. To measure the effect of using GILT and search engines, we excluded the sub-tasks data if participants *guessed* the solution without using the tool (i.e., zero interaction with the tool) or search engines before completing it (i.e., no search queries).

Prior knowledge. To control for prior knowledge, we used self-reported measures of participants' programming and domain experience. We expected more programming experience, especially in the specific domain, to lead to a faster understanding of code.

Experience in AI developer tools. Crafting effective prompts for LLM-based tools requires trial and error, even for NLP experts [59, 280]. Therefore, we asked participants about their experience with LLM-based developer tools. We expected participants' familiarity with other AI tools to affect their usage of the LLM-based information support tool, especially the use of free-form queries, and lead to more effective use of the extension than participants without such experience.

7.4.2 Methodology

To answer RQ7.1, we compared the effectiveness of using a GILT with traditional search engines for completing programming tasks by estimating regression models for three outcome variables. For task progress and code understanding, we used quasi-Poisson models because we are modeling count variables, and for the task completion time, we used a linear regression model.

To account for potential confounding factors, we included task experience, programming experience, and LLM knowledge as control variables in our models. Finally, we used a dummy variable (`uses_GILT`) to indicate the condition (using GILT vs. using search engines). We considered mixed-effects regression but used fixed effects only, since each participant and task appear only once in the two conditions (with and without GILT). For example, for the task completion time response, we estimate the model:

```
completion_time ~ domain_experience + programming_experience
                                    + AI_tool_familiarity + uses_GILT
```

Table 7.2: Summaries of regressions estimating the effect of using the prototype. Each column summarizes the model for a different outcome variable. We report the coefficient estimates with the standard errors in parentheses.

	Progress	*Time(s)*	*Underst.*	Progress	
	(1)	(2)	(3)	*Pros*	*Students*
Constant	0.41	312.65	−1.81**	−0.38	1.82**
	(0.49)	(185.33)	(0.89)	(0.68)	(0.83)
Domain experience	0.13*	23.14	0.41***	0.16	0.04
	(0.07)	(25.40)	(0.12)	(0.09)	(0.11)
Programming experience	−0.10	−23.67	0.20	0.01	−0.37*
	(0.12)	(43.53)	(0.22)	(0.17)	(0.21)
AI tool familiarity	−0.01	7.70	−0.09	0.07	−0.10
	(0.07)	(27.04)	(0.14)	(0.11)	(0.10)
Uses GILT	0.47***	−9.10	0.29	0.57**	0.29
	(0.16)	(57.26)	(0.28)	(0.22)	(0.25)
R^2	0.173	0.022	0.202	0.341	0.137
Adj. R^2	0.117	−0.046	0.148	0.243	0.010

Note: *p <0.1; **p <0.05; ***p <0.01.

The estimated coefficient for the `uses_GILT` variable indicates the effect of using GILT while holding fixed the effects of programming experience, domain experience, and LLM knowledge.

7.4.3 Results

Table 7.2 columns (1)-(3) display the regression results for three response variables. The task progress model (Table 7.2-(1)) shows a significant difference between the two conditions, with participants in the GILT condition completing statistically significantly more sub-tasks (0.47 more, $p < 0.01$) than those who used search engines, controlling for experience levels and AI tool familiarity. This indicates that GILT may assist users in making more progress in their tasks compared to search engines.

On the other hand, models (2) and (3) fail to show any significant difference in completion time and code understanding quiz scores between conditions. This suggests that users in the GILT condition do not complete their tasks at a sufficiently different speed or have a sufficiently different level of understanding than those in the control group, given the statistical power of our experiment.

In summary, the results suggest that GILT may help users make more progress in their tasks without changing, for better or worse, their speed and code understanding abilities.

7.4.4 Additional Analysis

After observing the significant effect of GILT on task progress, we dove deeper to examine whether all participants benefited equally from the tool. To do this, we divided the participants into two distinct groups based on their self-reported occupations (professionals and students) and estimated the effects of GILT usage in each group.[21] We opted for these groups as we did not have any prior theoretical framework to guide our grouping choices, and it provided a simple yet effective approach to group participants with multiple dimensions, including programming experience, skills, and attitude toward programming.

Although both groups were more successful when using the tool, there were notable differences in their performance gains. To better understand these variations, we estimated coefficients for each group (Table 7.2-Pros and -Students) and observed that the impact of GILT was significant only in the Pros group model. Specifically, professionals completed 0.57 more sub-tasks with GILT support compared to when they used search engines, whereas students did not experience significant gains. These findings suggest that the degree of benefit provided by GILT may vary depending on participants' backgrounds or skills.

7.5 RQ 7.2: GILT Usage

In this section, we focus on how participants interacted with GILT, their perception of the importance of different features, and how different factors correlate with the feature usage.

7.5.1 Usage of Features

To analyze in more detail how participants actually used the tool, we instrumented GILT and recorded participants' event and interaction logs. The logs allowed us to count the number of times participants triggered each feature, and in what order. To supplement the usage data, participants were asked to rate the importance of each feature in a post-task survey. We used these ratings to triangulate our findings from the usage data.

Figure 7.3 summarizes the sequences of GILT features used by participants in the treatment condition. On average, to complete their tasks in this condition, participants interacted with the LLM via GILT 15.34 times. The number of interactions per participant ranged from a minimum of 5 to a maximum of 23. The `Overview` feature was the most frequently used method to interact with the LLM, with an average of 4.76 activations per participant. Many participants also used `Overview` as their first feature, possibly because it requires minimal effort, with just a single click, in contrast to other features that necessitated the formulation of queries by participants, and perhaps also because some of the buttons (e.g., `Concept`) required first using the `Overview` feature. Participants also frequently used `Prompt-context` (4.12 times) and `Prompt-followup` (2.88 times). General prompting without code context was used less frequently (1.27 times). While participants generally used buttons less frequently, some used them more frequently than queries (e.g., P29), indicating personal preferences in prompt-based and prompt-less interactions. Specifically, the `API` button was used 1.24 times, the `Concept` button 0.45 times, and the `Usage` button 0.24 times on average.

The reported importance of the features by participants (see Figure 7.4) generally corresponds to the observed usage data. Most of the participants (97%) responded that the ability to directly ask questions to the LLM was extremely/very important, whereas their reported usefulness of the buttons varied. The reported importance of the overview feature (53% extremely/very important)

[21]We considered but decided against modeling interaction effects as they would have required more statistical power.

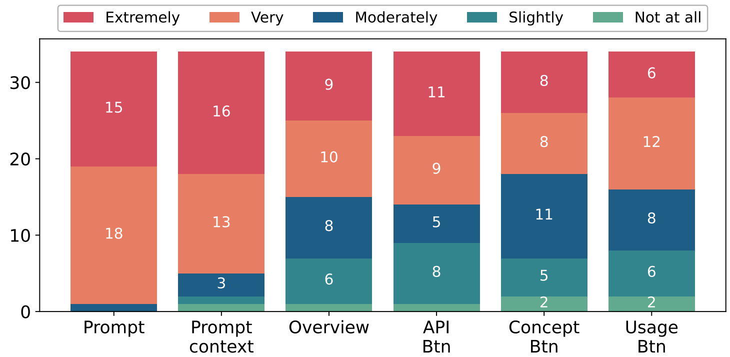
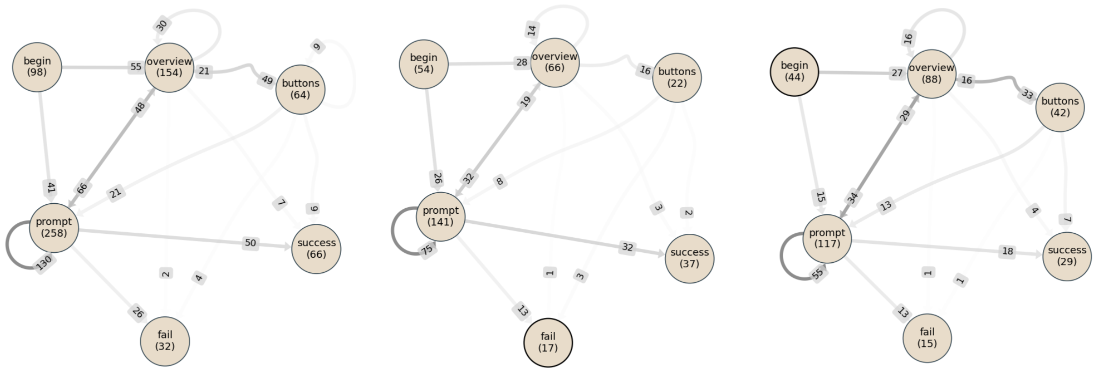


Figure 7.4: Participants' report on the importance of GILT features.



(a) All (b) Professionals (c) Students

Figure 7.5: Transition Graphs for User Interaction. Each node displays the number of times users interacted with respective features, and each edge indicates the counted number of transitions between the connected features. For space and readability reasons, `Prompt`, `Prompt-Context`, and `Prompt-Followup` are merged into `prompt`, and `API`, `Concept`, and `Usage` are merged into `buttons`. Counts lower than 5 are omitted except for the edges connected to the 'Success' and 'Fail' nodes.

was relatively low compared to the actual use, suggesting that participants may not have used the summary description provided by the overview but instead used it as context for further prompting or to activate buttons.

To further investigate participants' interaction with the tool, we created transition graphs (Figure 7.5) using sequences of feature-use events for each sub-task, using both the sub-tasks successfully completed by participants and those that resulted in failure (due to incorrect answers or timeouts). Out of the potential total of 128 sub-tasks (32 participants $\times$ 4 sub-tasks), 98 sub-tasks were started before the time ran out. In understanding the transition graph, we focused on the last feature in each participant's sequence, with an assumption that when a participant completes a task, it is likely that the information from the last interaction satisfied their information needs. Among the sub-tasks they successfully completed, a substantial majority (75%) originated from prompt-based interactions. At the same time, 83% of the failed tasks were also preceded by prompt-based interactions, so prompt-based interactions were not particularly likely to result in successful information seeking.

7.5.2 Professionals v.s., Students

To better understand the experiences of professionals and students (see Section 7.4.4), we compared the transition graphs for both groups (Figure 7.5 (b) and (c)). Notable distinctions emerged in terms of the features more likely influencing the success and failure of sub-tasks. Specifically, for professionals, a majority (86%) of successful sub-tasks originated from `prompt`, whereas for students, this percentage (62%) was statistically significantly lower ($\chi^2(1, 66), p < .05$). The success rate of prompt-based interaction was also higher among professionals (71%: 32 out of 45) compared to students (58%: 18 out of 31). Conversely, the success rate of the overview and buttons for professionals (56%: 5 out of 9) was lower than that of students (85%: 11 out of 13). These results may indicate that students, possibly with less experience in information seeking for programming, encounter challenges in formulating effective prompts compared to the professionals, and rely more on prompt-less interaction. However, we can also infer that prompt-less interaction is still not sufficient to compete with the benefits of prompt-based interaction with the current design, as they only accounted for less than 40% of the completed tasks.

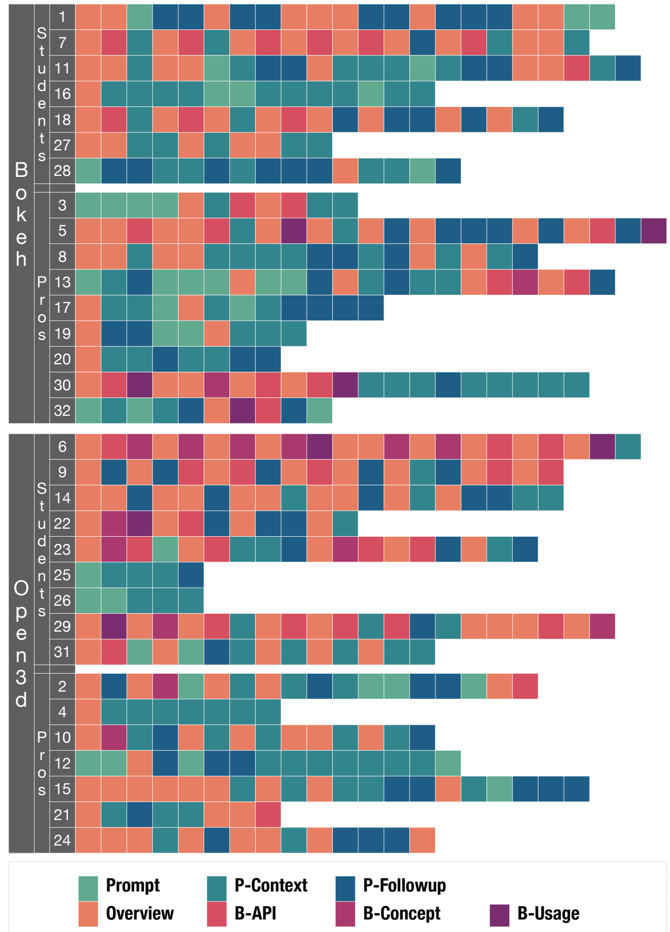


Figure 7.3: The sequences of feature usage in GILT. Each row corresponds to an individual participant, and the color cells are arranged chronologically.

To further investigate the differences in the two groups' prompt engineering, we analyzed the text of the prompts they wrote, by comparing the frequencies of bi-, tri-, and quadrigrams in the prompts. Table 7.3 presents the list of n-grams that showed divergent usage between the two groups. One notable observation is that the n-grams used by the professional group include more effective keywords, or they revise the prompts to incorporate such keywords. For instance, in the bokeh-3 sub-task, none of the participants in the student group used the critical keyword "annular wedge," which is essential for generating the information needed to solve the task, although it was used multiple times in the provided starting code. Instead, students tended to use more general keywords or keywords that had a different concept in the library (e.g., "legend") and faced difficulties in effectively revising the prompts. In addition, more participants in the professional group demonstrated proficiency in refining their prompts by providing further specifications. For example, one participant revised the prompt from "How to change the position of the bunny to 180 degrees" to "How to transform the bunny_mesh to 180 degrees." We infer

Table 7.3: Frequencies of n-grams used differently in prompts by professionals and students. For clarity, we only include n-grams used uniquely by one of the two groups, with a frequency difference of more than 2. If multiple n-grams share the same longer n-gram, we report only the superset.

Sub-task	n-gram	Pro.	Stu.
bokeh-2	(‘align’, ‘text’)	3	0
	(‘flip’, ‘label’)	0	3
bokeh-3	(‘annular’, ‘wedge’)	6	0
	(‘grid’, ‘annular’, ‘wedge’)	3	0
	(‘first’, ‘pie’)	3	0
	(‘pie’, ‘chart’)	3	0
	(‘add’, ‘legend’)	0	4
	(‘tell’, ‘line’, ‘need’, ‘change’)	0	3
o3d-3	(‘sit’, ‘upright’, ‘chair’)	4	0
	(‘make’, ‘bunny’, ‘sit’, ‘upright’, ‘chair’)	3	0

that the difference in the benefit received from GILT by the two groups can be at least partially attributed to their proficiency in prompt engineering.

7.5.3 Other Factors Associated with Feature Use

During the pilot studies, we observed that participants approached the tasks differently depending on their familiarity with other LLM-based tools, and styles of information processing and learning as observed in many previous studies on software documentation and debugging [29, 158, 159]. Thus, we tested whether the GILT feature use correlates with factors other than their experience.

Hypotheses. Out of two information processing styles [39, 56, 88], people who exhibit a “selective information processor” tendency focus on the first promising option, pursuing it deeply before seeking additional information. On the other hand, people who are “comprehensive information processors” tend to gather information broadly to form a complete understanding of the problem before attempting to solve it. Based on these processing styles, we hypothesized that selective processors would utilize GILT’s `Prompt-followup`, as they would prefer to use a depth-first strategy.

In terms of learning styles [39, 193], “process-oriented learners” prefer tutorials and how-to videos, while “tinkerers” like to experiment with features to develop their own understanding of the software’s inner workings. Consequently, we hypothesized that tinkerers would use GILT less often, as they would prefer to tinker with the source code rather than collect information from the tool.

We also expected that participants who were already familiar with LLM-based tools would use prompt-based interaction in general (`Prompt`), especially the chat feature, more frequently, as they would already be accustomed to using chat interfaces to interact with LLMs. Conversely,

Table 7.4: Summaries of regressions testing for associations between the user factors and the feature usage counts. Each column summarizes a regression modeling different outcome variables. We report the coefficient estimates with their standard errors in parentheses.

	Prompt (1)	Followup (2)	All (3)
Constant	1.39***	−0.82	2.43***
	(0.31)	(0.69)	(0.27)
AI tool	0.19**	0.38**	0.11
familiarity	(0.07)	(0.15)	(0.06)
Infomation	−0.04	0.44	−0.04
Comprh.	(1.15)	(0.30)	(0.13)
Learning	0.19	0.60**	−0.12
Process	(1.14)	(0.29)	(1.13)
R^2	0.262	0.283	0.165
Adj. R^2	0.184	0.206	0.075

Note: *p <0.1; **p <0.05; ***p <0.01.

we posited that participants with less experience with such tools would use the buttons more, as prompt engineering might be less familiar to them and place greater cognitive demands on them.

Methodology. To test for associations between GILT features used and the factors above we again used multiple regression analysis. We estimated three models, each focused on one particular feature. For each model, the dependent variable was the feature usage count, while participants' information processing style, learning style, and familiarity with AI developer tools were modeled as independent variables to explain the variation in usage counts.

Results. Table 7.4 presents the results of the regression analysis conducted for three response variables. The first model (Prompt (1)), which uses the total count of prompt-based interactions (`prompt` + `prompt-context` + `prompt-followup`), reveals that developers who are more familiar with other AI developer tools are more likely to prompt the LLMs using natural language queries. This result confirms our hypothesis that the AI tool familiarity level influences developers' use of queries. The familiarity level also has a statistically significant impact on `prompt-followup`, as shown in the Followup model (2). However, we did not find any significant impact of participants' information processing style on their use of GILT. This means that selective processors and comprehensive processors probed the LLMs similarly, as far as we can tell. The model, however, shows a statistically significant correlation between participants' learning styles and `prompt-followup` feature usage. Specifically, process-oriented learners tend to probe LLMs more frequently than tinkerers. This result might indicate that process-oriented learners are more likely to learn thoroughly before proceeding to the next step, while tinkerers tend to tinker with the code after getting the minimum amount of direction from GILT. Finally, the All model (3), which uses the total count of all GILT interactions, indicates that there is no statistically significant difference between the information styles, learning styles, and familiarity levels in terms of overall feature usage counts.

7.6 RQ 8.3: User Perceptions

In this section, we investigate how participants perceived their experience of using GILT. Specifically, we examine their perceived usefulness, usability, and cognitive load in comparison to search-based information seeking. Additionally, we explore the pros and cons participants reported, and suggestions for improving the tool.

7.6.1 Comparison with Web Search

We employed two wildly-used standard measures, TLX and TAM, in our post-task survey and compared them using two-tailed paired t-tests. TAM (Technology Acceptance Model) [130] is a widely used survey that assesses users' acceptance and adoption of new technologies, and TLX (Task Load Index) [95] is a subjective measure of mental workload that considers several dimensions, including mental, physical, and temporal demand, effort, frustration, and performance. The summaries of TAM and TLX comparisons can be found in Appendix D.

The average scores for the [perceived usefulness, perceived ease of use] in TAM scales were [27.3, 29.75] for the control condition, and [33.49, 34.2] for the treatment condition. The paired t-tests on the TAM scores indicated that there were significant differences in perceived usefulness and perceived usability scores between the two conditions ($p < 0.001$). Specifically, participants rated GILT higher on both dimensions than they did search engines.

For TLX items [mental demand, physical demand, temporal demand, performance, effort, frustration], the average scores were [3.8, -2.1, 4.0, 1.6, 3.4, -0.1] for the control condition and [3.3, -2.5, 2.6, 3.3, 3.3, 1.0] for the treatment condition. Paired t-tests on the TLX scores revealed statistically significant differences between the tool and search engines in temporal demand ($p < 0.05$) and performance ($p < 0.05$) but not in other items. These results indicate that the participants felt less rushed when using GILT than when using search engines, and they felt more successful in accomplishing the task with the tool than with search engines, but there were no significant differences in other dimensions.

7.6.2 User Feedback

In the post-task survey, we asked open-ended questions regarding (i) their general experience with using GILT, (ii) the tool's fit with the participants' workflow, (iii) comparison with other tools, and (iv) opportunities for improvement. Two researchers conducted a thematic analysis [50] to analyze the answers. Initially, two researchers separately performed open coding on the same set of 8 responses (25% of the entire data), and convened to discuss and merge the codes into a shared codebook. I coded the rest of the responses and discussed with the rest of the researchers whenever new codes needed to be added. The codebook is available in Appendix D, and we discuss some of them here.

The participants in this study reported several positive aspects of the tool, with the most notable being context incorporation. Participants valued the ability to prompt the LLM with their code as context, which allowed them to tailor the LLM's suggestions to their specific programming context, e.g., "the extension generated code that could easily be used in the context of the task I was performing, without much modification." (P5) Participants also found it extremely useful to prompt the LLM with just code context, as it allowed them to bypass the need to write proficient queries, a well-known challenge in search-based information seeking [116, 270]. P15 mentioned

"*It's nice not to need to know anything about the context before being effective in your search strategy.*"

Many participants reported that using the tool helped them speed up their information seeking, by reducing the need to forage for information, e.g., "Stack Overflow or a Google search would require more time and effort in order to find the exact issue and hence would be time-consuming." (P27)

Some participants, however, reported having a hard time finding a good prompt that could give them the desired response. Combined with the need for good prompts and the limitations of LLM, this led some participants to report that the responses provided by the tool were occasionally inaccurate, reducing their productivity. P28 summarized this issue well: "[*prototype*] *was not able to give me the code that I was looking for, so it took up all my time* (*which I got very annoyed about*). *I think I just didn't word the question well.*"

Participants had mixed opinions on the different features of the tool, especially the buttons. Some preferred to use "*different buttons for different types of information so I didn't have to read a lot of text to find what I was looking for*" (P7), while others thought that was overkill and mentioned "*a simpler view would be nice.*" (P8)

Compared to ChatGPT, 17 participants (out of 19 who answered) mentioned advantages of GILT, with the `Prompt- context` feature being one of the main ones. Participants expressed positive feelings about CoPilot but acknowledged that the tool had a different role than CoPilot and that they would be complementary to each other, e.g.: "*Copilot is a tool that I can complete mechanical works quickly, but* [*GILT*] *offers insight into more challenging tasks.*" (P29)

Many participants reported that the tool would be even more useful when combined with search engines, API documentation, or CoPilot,[22] as they provide different types of information than the tool. Having the ability to choose sources based on their needs would enhance their productivity by giving them control over the trade-offs, such as speed, correctness, and adaptability of the information.

7.7 Threats to Validity

One potential concern with our study design is the task and library selection. We only used tasks that show visible outputs, which might have led participants to detect potential errors more easily, compared to other tasks, such as optimization or parallel programming. However, we believe that the tasks we chose are representative of common programming errors that would need to be identified in real-world programming situations. Indeed, when we asked the participants in the post-task survey, both data visualization and 3D rendering tasks were reported to very or extremely closely resemble real-world tasks by 82% and 73% of the participants.

Similarly, the selection of libraries might have biased the study results. However, in selecting libraries for our study, we avoided using popular libraries that could unintentionally give an advantage to LLMs. We believe that the libraries we chose are of medium size and quality, and therefore represent a fair test of the LLM tools. However, it is possible that different libraries or larger codebases could produce different results.

Despite our efforts to create a controlled experience, several factors differentiate our in-IDE extension from search engines, aside from the inclusion of LLMs. For example, although previous

[22]Notably, GitHub independently announced these enhancements to Copilot already, after we conducted our study: `https://www.theverge.com/2023/7/20/23801498/github-copilot-x-chat-code-chatbot-public-beta`

research investigating the incorporation of search into IDE did not find a statistically significant difference between the control and treatment groups [35, 134], the in-IDE design itself may have been more helpful than access to LLMs, as it potentially reduced context-switching. Thus, further studies are needed to gain a better understanding of the extent to which each benefit of our prototype can be attributed to these differences.

Additionally, the laboratory setting may not fully capture the complexity of real-world programming tasks, which could impact the generalizability of our findings. Also, the time pressure participants could have felt, and the novelty effect in a lab setting could have changed how users interact with LLMs. Our sample size, 32, was relatively small and skewed towards those in academia. This may also limit the generalizability of our findings to more professional programmers. Thus, future research with larger, more diverse samples is necessary to confirm and expand upon our results.

Our analysis also has the standard threats to statistical conclusion validity affecting regression models. Overall, we took several steps to increase the robustness of our estimated regression results. First, we removed outliers from the top 1% most extreme values. Second, we checked for multicollinearity using the Variation Influence Factor (VIF) and confirmed that all variables we used had VIF lower than 2.5 following Johnston et al. [111].

Another potential threat to the validity of our findings is the rapid pace of technological development in the field of LLM tools. Despite our efforts to use the most up-to-date LLM available at the time of the submission, it is possible that new breakthroughs could render our findings obsolete before long.

7.8 Discussion and Implications

Comprehension outsourcing. Our analysis revealed an intriguing finding regarding participants' behavior during the study, where some of them deferred their need for code comprehension to the LLM, which was well described by one participant as *comprehension outsourcing*. These participants prompted the model at a higher level directly and did not read and fully comprehend the code before making changes. As one participant commented, "*I was surprised by how little I had to know about (or even read) the starter code before I can jump in and make changes.*" This behavior might be attributed to developers' inclination to focus on task completion rather than comprehending the software, as reported in the literature [151]. Or, participants may have also weighed the costs and risks of comprehending code themselves, and chosen to defer their comprehension efforts to the language model. While this behavior was observed in the controlled setting of a lab study and may not fully reflect how developers approach code comprehension in their daily work, it does raise concerns about the potential impact of such a trend (or over-reliance on LLMs [261]) on code quality. This highlights the importance of preventing developers who tend to defer their comprehension efforts to the LLM from being steered in directions that neither they nor the LLM are adequately equipped to handle. Studies showing developers' heavy reliance on Stack Overflow, despite its known limitations in accuracy and currency [269, 281], further emphasize the need for caution before widely adopting LLM-based tools in code development. Research on developers' motivations and reasons for code comprehension when LLMs are available will be valuable in informing future tool designs.

Utilize more context. One of the main advantages of GILT reported by the participants is its ability to prompt the LLM with the code being edited as context. We believe that additional types of context can be leveraged to improve the tool's utility, including project context (e.g., project scale and domain), system context (e.g., programming languages and target deployment environments), and personal context (e.g., programming expertise in libraries, and domains). By combining these contexts with proper back-end engineering, we believe that GILT, or other LLM-powered developer tools, will be able to provide relevant information to developers with even less prompt engineering efforts of the users.

Need further studies in real-world settings. One possible explanation for some of the models with null results from RQ7.1 and RQ7.2 is the artificial setting of the lab study, where participants were encouraged to focus on small, specific task requirements instead of exploring the broader information dimension. For example, participants prioritized completing more tasks rather than fully understanding the code, as reported by participant P18 in their survey response: " [*GILT*] ..., *which could definitely help one to tackle the task better if there weren't under the timed-settings.*" Thus, although our first study shed some light on the potential challenges and promises, to fully understand the implications of deploying this tool into general developer pipelines, it is necessary to observe how programmers use it in real-world settings with larger-scale software systems, less specific goals, and over a longer time frame. Given that GitHub recently launched CopilotX [83], a tool that offers a comparable set of features to our prototype to enhance developer experience, such research is urgently needed. We believe that our findings are a timely contribution and a good first step for researchers and tool builders in designing and developing developer assistants that effectively use LLMs.

7.9 Summary

The user study results have demonstrated that GILT significantly enhances developers' ability to complete tasks compared to traditional information-seeking methods. Participants also expressed positive experiences while using GILT, especially on the ability to incorporate their task context.

Chapter 8

Towards Building Personalized Information Support

In evaluating user-centered information support tools, we have demonstrated that user-centered design can meaningfully improve programmers' information seeking. Specifically, the qualitative analysis of user responses in Chapter 7 has shown that incorporating the code context improves their information-seeking. In this final chapter, we test the potential value of incorporating an even broader user context, and eventually *personalized* generation-based information support by automatically augmenting the user context.

8.1 Introduction

Developing a software system requires developers to consider numerous dimensions, ranging from its architecture to implementation bugs. Each dimension demands information of varying scopes and abstractions, making software engineering highly information-intensive. Thus, developers spend 35% of their time on the foraging itself [117], and 50% when other aspects of foraging are taken into account [197]. This intensive search for information is driven by the need to evaluate multiple viable solutions, taking into account their constraints and the trade-offs involved. Inadequate or missing information can lead to suboptimal design decisions, adversely affecting the quality of the software product. Therefore, timely access to all necessary information is crucial for enhancing developers' productivity and, consequently, the quality of the final software products.

However, information seeking is still not easy in software engineering [14, 205, 270]. Much information about software and systems is not properly documented [255] or it is spread across varying mediums in different formats, often getting outdated as software evolves [55]. Developers have used search engines as a major way of information seeking (e.g., they issue more than 20 search queries every day [270]), which many developers find challenging because they may not know, or may forget to include, the right keywords [65, 116, 147]. Even when they retrieve a promising page, they still need to evaluate the relevance of the information [80, 116], which is itself challenging, especially to those who are new to the API, or programming in general [116].

The emergence of generative models and large language models (LLMs) has introduced new avenues for satisfying programmers' information needs, offering some advantages over traditional methods [138, 176, 219]. However, the effectiveness of these models is still contingent upon the ability to write effect prompts, a task that often requires trial and error, even for NLP experts [59, 280]. While LLMs can process complex queries, allowing users to provide rich clues for search, as users often receive a response from LLMs instead of having the opportunity to browse through multiple potential solutions as in conventional information retrieval approaches, it poses a challenge for discovering suitable solutions when queries fail to direct the LLM effectively.

In this chapter, we posit that considering developer *context* can alleviate the challenges associated with software information seeking, as it can explain some of the programmers' information needs, even when not explicitly expressed in their queries. Context is defined as "everything that affects a computation except the explicit input and output" [139], or, more specifically for the information retrieval system, as "everything that a perfect information retrieval system would need to know about the user, their situation, the domain, and anything else in order to return exactly the results relevant to the user' s stated query" [67]. Leveraging context has proven beneficial in other domains, such as search engines and assistance systems, which have successfully utilized spatiotemporal context and interaction history to provide personalized results [48, 67, 82]. Although mostly imagined to operate without considering the software development context, thus serving as a "one size fits all" solution, there is already a vision in our community that contextual search would be useful in supporting developers' information seeking [13, 215].

We argue that the time is ripe to begin developing *personalized* information support. The personalization of information support for programming might have not seemed to be feasible, as most of the general information seeking was done at search engines, which can only retrieve information from existing documents, providing a limited level of personalization. However, since LLMs generate information, they can further personalize the information support, by suggesting or skipping information at a granularity finer than a document level, which retrieval mechanisms like search engines are limited to, and LLMs can even format the information as the programmer would want it. Furthermore, LLMs can accept a much richer request than a single search query that programmers often use for information retrieval with search engines, enabling the user to provide even broader contextual information.

As an initial step towards providing personalized information support, this chapter focuses on identifying the categories of context that may enhance information support and testing the feasibility of measuring the impact of including such contextual factors on developers' productivity, especially when working with unfamiliar code. Through the examination of ChatGPT prompts in software engineering artifacts and an experimental ablation study, this research aims to address the following questions:

- RQ8.1: What are the contextual factors programmers use in information seeking?
- RQ8.2: Does providing contextual factors enhance the quality of generated information?

8.2 RQ8.1: Understanding Context for Information Support

We conducted an empirical study to understand what type of contextual factors can be useful in satisfying the information needs of programmers in the context of LLM-based information support.

8.2.1 Study Design

We analyzed the DevGPT dataset [271]. DevGPT comprises mentions of ChatGPT conversations in software development artifacts, including GitHub commits, issues, pull requests, discussions, files, and Hacker News. The dataset [23] contains 4,733 shared ChatGPT links, a total of 29,778 prompt/answer-pairs, sourced from 3,559 GitHub or Hacker News references.

First, as we have little prior understanding of how programmers express their information needs to LLM-powered generation-based information support tools, we performed open coding. For the initial analysis, we randomly selected 10 conversations per source type (e.g., GitHub issues), resulting in 60 conversations. We excluded the ChatGPT conversations that contained dead links (i.e., 404), or the ones containing prompts written in languages other than English. We also excluded the conversations that are not relevant to Software Engineering tasks, such as "what's the real Netflix idea origin story?" The open coding was conducted at a phrase level, where a phrase can be a sentence (e.g., "What is immutability?", or a part of a sentence (..."expert python programmers"...).

The open coding process yielded 39 distinct codes, encompassing various information needs, relevant contextual information provided, and some guidelines for directing LLMs. A significant observation from the initial coding phase was the substantial overlap among many codes with pre-existing categories and dimensions identified in the studies of programmers' information-seeking behavior [79].

Overlap with existing knowledge on information needs. The initial coding revealed that the types of contextual information programmers supply are very similar to what they have provided as part of their Stack Overflow questions [146]. Specifically, we observed that the information needs we coded overlap with the seven-question types from Beyer et al. [31]: API usage, Conceptual, Discrepancy, Errors, Review, API change, and Learning. Beyer et al. [31] studied and merged the taxonomies of prior studies [16, 252] on information needs and created a new taxonomy of information needs shown in Stack Overflow questions. As the taxonomy is broad enough to cover the general information seeking in software engineering, although the focus was API-related questions, we found that it can explain more general information needs expressed in the DevGPT dataset.

Overlap with existing knowledge on contextual factors. The initial coding also revealed some overlap with the varied contextual factors influencing developers' information seeking behavior in conventional setting, studied by Freund [79]: Personal factors, Project factors, Work task factors, and Information task factors. The list was created before the advent of LLM-powered information seeking, from a semi-structured interview study conducted between 2004 and 2005, where programmers relied heavily on conventional information seeking, using search engines, documentation, and communication with colleagues. Although numerous studies have explored programmers' information foraging strategies and how these strategies vary with different programmer backgrounds [126, 151]), the differences were mostly discussed as implications from studies on documentation or program comprehension, providing more in-depth, but not extensive lists of contextual factors influencing programmers' information-seeking behaviors.

[23] 20231012 snapshot

8.2.2 Results: Contextual Factors Used by Programmers

We conducted a second round of coding, using the two taxonomies discussed above (information needs, and contextual factors). The closed-coding round resulted in 6 codes for information needs (Table 8.1), and 5 codes for contextual factors (Table 8.2). The codes from the existing two taxonomies could cover multiple codes, reducing the number of codes, from 39 to 11. For example, both phrases coded as "context: repo" and "context: environment" in the previous open coding fell under the "Project Factors" in this round of coding.

Table 8.1: Descriptions and examples of information needs types.

Code	Description & Representative Quote
Impleme-ntation	Needs information to implement specific functionality "*How can I use cef to make Chrome devtools open on selected screen?*"
Conceptual	Needs information about conceptual knowledge, such as design patterns or architectural styles "*What is immutablity?*"
Discrepancy	Needs information to understand and resolve unexpected behavior "*Why is my redirect not working?*"
Error	Needs information to resolve exceptions and errors "*that produces this error, how do we fix it?*"
Review	Needs information to find for better solutions, best practice approaches, or verification of the program. "*Is there anything you suggest as an alternative?*"
Learning	Needs resources to learn a tool, language, or concept. "*Could you generate a description of how it works that a 10-year old might understand.*"

The information needs articulated in the prompts were closely aligned with the taxonomy of Beyer et al. [31]. Our analysis of the sampled ChatGPT prompts revealed the presence of six out of the seven categories in the original taxonomy.

Many prompts contained phrases such as how to implement something or ways of using something, initially suggesting an alignment with the "API usage" category. However, to more accurately encapsulate the broader scope of information needs beyond just API knowledge expressed in these how-to inquiries, we revised this category to "Implementation". This adjustment better reflects the varied implementation-related questions found in the prompts. We observed an absence of prompts explicitly seeking information on "API changes", indicating a potential area of lesser concern or different information-seeking behavior among ChatGPT users.

Out of the four categories identified by Freund [79], "Personal factors", "Project factors", "Work task factors", and "Information task factors", "Information task factors" were excluded in our final codebook as they cover the same dimension as the information needs, which was covered in our previous codebook. "Work task factors" was further divided into "Task Requirement", "Status Description", and "Software Artifact" as the task factors were specified in detail in the majority of the prompts. In the end, contextual information in the prompts resulted in five codes:

Table 8.2: Descriptions and examples quotes of contextual factors

Code	**Description & Example**
Task Requirement	Descriptions of the functional and non-functional requirements of the task "*I want to develop an extension that needs to keep track of all tabs, even when the browser restarts.*"
Status Description	Descriptions on the current status of the task, including the functionality that has been implemented or the actual undesirable results obtained "*It seems the script is not finding C column in Dashboard sheet*"
Software Artifact	Products or by-products of software systems, including source code, error messages, or runtime outputs "*TypeError: Cannot read properties of undefined* (*reading* '*importKey*')"
Project Factors	Factors relevant to the software engineering projects, including a range of characteristics of the technical infrastructure or project scope, plan, or application domain "*Packages: mplayer, vlc, all the packages from base-devel ...*"
Personal Factors	Programmers' personal backgrounds including expertise in programming, familiarity with application domains, and roles - familiarity: "*Yes, I am familiar with statically typed languages, and the hazards of javascript loosely typed variables*" - role: "*We are expert python programmers and data analysts/data scientists/researchers*"

Task Requirement describes the goal and the constraints of the software engineering tasks programmers are working on. This code includes the desired functionality, type of functional improvement (e.g., "*I never want any downtime which would cause the images to not display.*") or non-functional improvement to make (e.g., "*Provide several suggestions for potential indexes that might speed up the query...*"), and some constraints like "*All divs appended to 'theGridItself' must be organized such that each row consists of 'squareSideSize' number of divs, no more and no less.*"
Status Description describes the less-desired current status, such as implemented functionality that requires further work or that does not meet certain functional or non-functional requirements, such as "*Now, it ignores your default browser and opens links in Edge by default.*"
Software Artifact includes source code, error messages, runtime outputs, and test cases that are generated during the software development life cycle. This is provided to incorporate a detailed context of the software systems and the status.
Project Factor includes meta-information of the software engineering projects programmers work on, including the technical infrastructure of the system they are working on, the programming languages, platforms, tools, packages, operating systems, and devices.
Personal Factor is related to the individuals, describing their professional expertise (e.g., "*I am a beginner*"), familiarity with application domains (e.g., "*I don't have much experience working with google drive.*"), libraries, frameworks, or programming languages, and programmers' roles (e.g., software architect). This is often provided to set the depth of the information generated by LLMs.

8.3 RQ8.2: Experimental approach to measuring the value of providing additional contextual factors

From our qualitative analysis, we identified various contextual factors that programmers consider when seeking information using large language models (LLMs). This observation aligns with empirical evidence from our previous study detailed in Chapter 7, which indicated a preference among programmers to integrate their context into such systems. Despite these findings, the direct impact of these contextual factors on the effectiveness of information generation remains uncertain, primarily due to challenges in quantitatively assessing the benefits of context integration. To address this gap, we have designed a novel experimental ablation study. This experimental study aims to quantitatively measure and compare the influence of different contextual factors, thereby elucidating their specific contributions to enhancing the quality of information support provided by LLMs.

8.3.1 Study Design

LLM-as-a-judge

The most straightforward way of measuring the usefulness of adding contextual factors is to compare the quality of information generated with and without specific contextual factors. Ideally, evaluating the quality of responses would involve consultations with human experts or conducting A/B tests with actual programmers. However, such evaluations can be very expensive, particularly because evaluating the quality of information for real-world software engineering tasks demands extensive technical expertise.

Table 8.3: Summary statistics of LLM coding

# Prompt	Task Requirement	Status Description	Software Artifact	Project Factor	Personal Factor	Total
458	364	66	255	179	47	911

To mitigate this challenge, we propose an experimental evaluation approach, using to the "LLM-as-a-judge" framework [284]. This method offers a cost-effective and scalable alternative for evaluating various LLMs. In this setup, a robust base LLM, serving as the "judge," is presented with corresponding outputs from two different LLMs, and a specific problem description to be used to ground its evaluation. The judge LLM is then tasked with selecting the preferred answer based on certain criteria, and with providing rationale behind the comparison.

For our study, we compare responses from the same LLM, GPT-4, generated with and without a particular contextual factor. We utilize a prompt template designed by Zheng et al. [284], which instructs the LLM to evaluate responses based on criteria such as helpfulness, relevance, accuracy, depth, creativity, and detail. In this experiment, we choose GPT-4 as the judge, following findings by Zheng et al. [284] that it exhibits the highest agreement level with human evaluations at 85%, surpassing even the between-human agreement rate of 81%.

Data Collection

To compare the responses generated with and without specific contextual factors, it is essential to include user prompts that incorporate additional contextual factors identified previously. As manually coding the dataset is impractical, especially given the technical complexity, and thus the long conversations, included in DevGPT dataset, we decided to employ an LLM for automatic coding. We utilized a GPT model, following Chew et al. [47], which involves instructing an LLM to perform deductive coding using a predefined codebook. We adopted their prompt template, utilized the codebook from the previous section, and employed the GPT-4-turbo-preview model for this task.

To accurately assess the impact of contextual factors, we only included conversations that consisted of a single user prompt and a single response, with the prompt length limited to 4,096 characters. The LLM coder was allowed to apply the same code (e.g., "project factor") multiple times within a single prompt, but each segment of text was assigned only one code.

The results of LLM coding are detailed in Table 8.3. In our analysis, the LLM coder processed 458 ChatGPT prompts from the DevGPT dataset, assigning a total of 911 codes. A substantial majority of the prompts (79.47%) included task requirements, and over half (55.68%) contained software artifacts such as code snippets or error messages. The most common of the other contextual factors were project factors (e.g., programming languages in use), while personal factors (e.g., programmers' experience level) were the least common.

Although LLM coding facilitated large-scale analysis, the accuracy of the coding was insufficient for direct use in further steps. For instance, the model struggled with personal factors, often confusing context related to the prompt writer (e.g., "*We are expert python programmers and data analysts/data scientists/researchers*") with hypothetical scenarios (e.g., "*Imagine being a developer advocate with 5 years of experience*").

We also discovered that some contexts, particularly task requirements or status descriptions, were critical to the prompts, which posed challenges for our ablation experiment. In many cases, prompts were entirely coded as task requirements or status descriptions, such as "*Create a python script to send a DNS packet using scapy with a secret payload.*" As excluding these contextual factors would leave insufficient information in the prompts to generate meaningful responses for the experiment, we refined our focus to software artifacts, project factors, and personal factors, which were less intertwined with the essential content. We also encountered prompts where software artifacts were pivotal, such as those containing solely an error message, but decided to keep software artifacts as one of the context categories for ablation study, because there were still many prompts adding software artifacts to enhance LLM responses even when they were not pivotal.

To address issues arising from coding accuracy and to filter cases where the context was critical in the prompts, we conducted a manual review of the LLM coding before proceeding with the ablation experiment. Starting with personal factors, due to their fewer instances, we identified 12 suitable for the ablation experiment. We then randomly sampled instances of software artifacts and project factors and continued filtering until we had 12 suitable instances from each of the three other categories to balance the category distribution in the experiment.

Experiment Protocol

For an ablation study, for each prompt, we requested LLM to generate responses for the original prompt (e.g., "... Assume I am a beginner and have no git and node installed. ...") in the dataset, and the prompt after excluding the contextual factors (e.g., "... Assume I have no git and node installed."). We then provided the two responses to the LLM evaluator to attain verdicts. The prompt template for the LLM evaluator consisted of a problem description, explaining the user's request, and two responses to compare. For both LLM generator and LLM evaluator, we used the same model, GPT-4. This consistency helps minimize any potential self-enhancement bias, where evaluators might prefer responses from the same model used for generating answers.

To quantify the impact of context on response quality, we introduced two metrics: score-original and score-removed. The score-original was calculated by providing the evaluator with the original, context-rich prompt as the problem description alongside the two different responses to compare. Conversely, the score-removed was derived by providing the modified, context-reduced (i.e., de-contextualized) prompt, with the two responses. These metrics are intended to reveal the importance of contextual information at different perspectives, with score-original highlighting the importance of *not missing* certain context, and score-removed assessing the effectiveness of augmenting *additional* context in prompt.

To compensate for randomness and evaluator bias, each evaluation was repeated four times. We also counterbalanced the presentation order of the responses to remove positional bias, where the evaluator might favor the first-presented answer. We computed the scores by counting how many times the response generated with the original, context-rich prompt wins, making the possible range of both scores -4 to 4. A positive score means that the response generated with the prompt with contextual information was judged to be better, a negative score means the opposite, and 0 means the two responses are similar (i.e., tie).

The example original prompt, de-contextualized prompt, LLM generator's responses generated with the two different prompts, the LLM evaluator's comparisons with the rationales, and how we calculated the scores can be found in Table 8.4.

Table 8.4: An example the user prompt and the LLM-as-a-judge evaluation result including the comparison result and the rationale behind it. A/B in the parenthesis indicates which Assistant generated the response for the prompt *with* contextual information.

User Prompt	
Mention Link	`https://github.com/Vandivier/ai-snacks/blob/c732a2bd53da134adfc5eb1fcd0e972e9d35ec53/time-to-learn-coding-twitter-thread/README.md`
Org. ChatGPT Conv.	`https://chat.openai.com/share/df755e01-51c1-4f76-8adb-8b3e03656995`
Prompt (**context** removed for prompt-removed)	**ROLE: We are expert python programmers and data analysts/ data scientists/researchers** CONTEXT: the attached raw-thread-text.txt file contains a raw and unstructured text dump based on a Twitter thread. In this thread, individuals discuss how long it took them to learn to code. ...
LLM Responses	
Response: original	To achieve the task of analyzing the time it took for individuals to learn coding as a skill and then land a professional programming job from a raw ...
Response: removed	Given the task at hand, we'll need to execute a few steps to efficiently create a structured dataset from the raw, unstructured text dump...
LLM-as-a-judge Evaluation & Rationale - Score-original: 4	
Problem Desc.	ROLE: We are expert python programmers and data analysts/data scientists/researchers CONTEXT: the attached raw-thread-text.txt file...
org-1 (A)	... Assistant A's inclusion of a anticipatory guidance on potential challenges related to data extraction and the need for advanced NLP techniques, ... Final Verdict: **[[A]] (+1)**
org-2 (A)	... Final Verdict: **[[A]]** **(+1)**
org-3 (B)	... Final Verdict: **[[B]] (+1)**
org-4 (B)	... Final Verdict: **[[B]]** **(+1)**
LLM-as-a-judge Evaluation & Rationale - score-removed: 2	
Problem Desc.	CONTEXT: the attached raw-thread-text.txt file contains a raw and unstructured text dump based on a Twitter thread. ...
rm-1 (A)	... Assistant A has a slight edge due to its detailed technical guidance and realistic acknowledgment of the complexity involved in dealing with unstructured text data. Therefore, **[[A]] (+1)**
rm-2 (A)	... **[[A]] (+1)**
rm-3 (B)	... **[[C]] (0)**
rm-4 (B)	... **Verdict** **[[C]]** for a tie. **(0)**

Table 8.5: Summary statistics of the comparisons between LLM response generated with prompts, with and without a specific contextual information category. *avg.* columns show the average score, and CI columns indicate the confidence intervals.

Context	n	Score-Original		Score-Removed		Score Difference	
		avg.	CI	avg.	CI	avg.	CI
Artifact	12	3.17	[2.27, 4.06]	0.92	[-0.37, 2.20]	2.25	[0.87, 3.63]
Personal	12	0.92	[-0.40, 2.23]	0.08	[-1.28, 1.45]	0.83	[-1.15, 2.82]
Project	12	2.08	[0.56, 3.60]	0.00	[-1.65, 1.65]	2.08	[0.74, 3.42]
All	36	2.06	[1.33, 2.78]	0.33	[-0.43, 1.09]	1.71	[0.87, 2.58]

8.3.2 Results: Impact of Including Contextual Factors in Response Quality Enhancement

Table 8.5 summarizes the evaluation results. The average score-original, the score measured with the original, context-rich prompt as the problem description for the LLM evaluator, of 2.06 out of 4 indicates that including appropriate context information, or not missing necessary context in the prompts, can help generate better-quality LLM responses. The average score-removed, the score measured with de-contextualized prompt as the problem description, of 0.33 indicates that additional contextual information leads to answers that are perceived as slightly better, although the effect is not significant. The difference between the score-original and the score-removed (score-original - score-removed) was 1.71 on average, indicating that the benefit of including additional contextual information may be less significant, compared to not missing important, critical context.

Software artifact

Both scores suggest that including software artifacts in the prompt is beneficial: the score-original of 3.17 and the score-removed of 0.92 are the highest among the three context categories. The LLM evaluator preferred responses that included software artifacts, noting they "*provide a more accurate, relevant, and detailed response to the user's requirements,*" and adhere better to "*the output format.*" However, there are indications that including additional software artifacts can sometimes confuse the LLM evaluator, as evidenced by the disparity between the score-original and score-removed. For example, when LLM evaluator is asked to compare two responses with the de-contextualized prompt as the criteria, the LLM evaluator found the response generated with the original prompt to provide unnecessary information, mentioning "*Despite* [*response generated with the original prompt*] *providing valuable additional advice,* [*response generated with de-contextualized prompt*]*'s response is more aligned with the user's immediate needs.*" This suggests that while adding software artifacts may enhance responses in general, it is most effective when the artifacts are directly related to the developer' s current task. It also indicates that indiscriminately including every code snippet and error message might not necessarily enhance the quality of an LLM' s response.

Project factor

Responses that are generated with prompts including project-specific factors achieved an average score-original of 2.08. This suggests that integrating these factors allows the LLM to generate more relevant responses tailored to the specific system a programmer is working on, thereby providing solutions that are better contextualized to the task at hand. For example, the LLM evaluator noted, "[*A response generated with contextual information*] *provided a comprehensive answer, addressing various architectures and project requirements.*"

However, when evaluated in a more general setting, with the de-contextualized prompt as the problem description, the inclusion of additional project factors did not improve the score-removed, which remained at 0.00. In some instances, adding project context refined the responses, which proved beneficial. For instance, one response was described as "[*providing*] *a slightly more technical and in-depth analysis.*" Conversely, for certain project factors (e.g., "*I' m writing an iPhone Swift meditation app*"), this additional information led to responses being overly specific, such as in cases where "*the response was too detailed for a user who did not specify their development platform and asked a broad question.*"

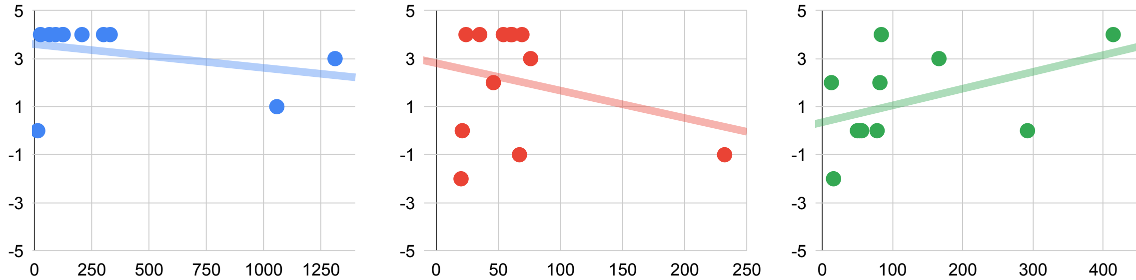
Personal factor

The average score-original of 2.06 out of 4 suggests that including relevant personal characteristics in the prompts (e.g., user's experience level) can enhance the LLM's responses by allowing it to adjust the complexity and provide more detailed explanations. For instance, the LLM evaluator noted that "[*response generated with the original prompt including the personal context*] *adds a layer of depth to the advice that could be especially beneficial for data scientists and researchers.*" However, the differences between score-original and score-removed were the highest among the three context categories, as indicated by wide confidence intervals, 3.97, in score differences. This variability may be because personal factors influence the tone and depth of answers generally, so the evaluation heavily changes based on the problem description. This may indicate that although including the personal context might support LLMs generating better responses in general, but not consistently.

Length analysis

Previous research has found that LLM evaluators tend to prefer longer responses due to a length bias [284]. Given that LLMs often produce longer responses when prompted with longer inputs, this could have influenced our results, as the original prompts were designed to be longer than the de-contextualized prompts. To investigate the presence of a length bias in our evaluation, we conducted a lightweight sanity check on whether a comparative quality benefit—specifically, higher scores—was attributable to the length (number of characters) difference between the original and de-contextualized prompts.

The results, shown in Figure 8.1, suggest that the influence of length bias might not be significant. Despite the challenges of conducting statistical tests due to the small sample size, there is no clear trend indicating that scores increase with the length difference; in fact, the opposite trend was observed for software artifacts and project factors. This finding suggests that augmenting user prompts with overly lengthy context may not always contribute to the response quality enhancement, as it could potentially confuse the LLM or lead to overly specific responses, diminishing the quality of the output.



(a) Software Artifact (b) Project Factor (c) Personal Factor

Figure 8.1: Scatter plots showing the trends between the character length difference between the original and de-contextualized prompts (x-axis) and the score-original (y-axis).

8.4 Discussion

By analyzing the information needs and contextual factors shown in LLM prompts, we have identified 6 types of programmer information needs and 5 types of contextual factors programmers provide as part of their prompts. In addition, through an experimental ablation study gauging the usefulness of the contextual factors, we saw promising findings indicating that adding these contextual factors can enhance the quality of LLM-generated information.

8.4.1 Study Design Limitation

However, although the ablation study to measure the importance of contextual factors is novel, and utilizing LLM-as-a-judge idea allowed us to automate the experiment, it also still holds many limitations.

Multiple layers of LLM usage

The study design involves multiple layers of LLM usage: we first generate responses using LLMs, with and without contextual factors in the prompt, and then evaluate these responses also using an LLM. While this methodology enables the automatic quality evaluation of different contexts—potentially facilitating larger-scale studies in the future—it also introduces significant noise into our experiment. Specifically, we observed inconsistencies in the criteria used by the LLMs during response generation and evaluation. Even though we employed the same model, GPT-4-turbo-preview, for all tasks, the necessity to initiate each chat anew made it challenging to maintain a consistent rationale and evaluation criteria between the generating and evaluating LLMs. For instance, the LLM generator provided a response to the prompt including a personal factor "I am a beginner" by omitting complex troubleshooting details present in the de-contextualized version, possibly because beginners might not be able to understand or need the troubleshooting details. However, when the LLM judge compared the two responses for score-original, it favored the de-contextualized response for including detailed troubleshooting information, noting: "*This small detail could significantly aid users facing execution issues, making it a valuable inclusion.*" Inconsistencies, and unclear rationale behind LLM responses of each layer, necessitated the manual investigation to interpret the results, which we discuss more next.

Need for human verification

In our study, we encountered significant limitations stemming from the need for extensive manual verification, contradicting our initial goal of using LLM evaluations to facilitate a large-scale study. This need arose due to inaccuracies in the coding step (see Section 8.3.1-Data collection), inconsistencies between the LLM generators and evaluators, and challenges in interpreting LLM evaluations. Specifically, manual qualitative coding limited our dataset to only 60 ChatGPT conversations for code identification. Furthermore, the need for preliminary manual verification of LLM-generated codes restricted our analysis to a sampled set of prompts for the ablation study. Expanding human resources could potentially aid in scaling the experiment; however, the expensive nature of human evaluation, particularly for technically complex content, inherently limits the practical expansion of the study. Therefore, we believe that developing mechanisms to reduce the reliance on human verification or to streamline the human evaluation process will be crucial for effectively scaling this experiment in future research.

DevGPT dataset

The inherent limitations of conducting experiments with a pre-generated dataset, such as DevGPT, are also significant. For example, DevGPT, collected from public, open-source interactions with ChatGPT in October 2023, does not include data from proprietary software systems. Thus, the contextual factors identified in the DevGPT data may not be the extensive set of all possible factors users might use, as we might have missed them in coding. Moreover, the dataset's temporal scope means that many conversations may have been exploratory in nature, as the snapshot we used in the study was collected in October 2023, not too long after ChatGPT's public release in November 2022. Despite our efforts to manually filter out irrelevant interactions, such as philosophical inquiries or tests of malicious intent, the broader evaluation of contextual factors' value might still inadvertently include such data. Thus, we do not claim that we identified the extensive set of contextual factors or our finding can be generalized, and believe that similar studies with different populations, ideally with larger-scale, are needed to better reflect the evolving real-world use of LLMs by programmers.

8.4.2 Potential for automatic contextualization

Despite the limitations, the experimental study provides some promising preliminary evidence on the benefit of not missing important contextual factors, or even including additional ones. Thus, we argue that it might be worth investigating ways of automatically augmenting the user prompts by adding contextual factors, to enhance the programmers' information seeking in general. This may alleviate the programmers' challenges in forming effective prompts for LLM due to their lack of expertise in application domains or programming. In this section, we provide some ideas on how we can automate this context augmentation of the user prompts. Specifically, we examine each contextual factors we discovered focusing on where they can be found (source) and whether they change over time (variability).

Sources. Not all contextual factors are observable. For example, the programming language might be easily identifiable from the file names, but it will be harder to identify the quality requirements of the software system. General contextual search mainly uses observable variables (e.g., search history) as it is difficult to make direct use of unobservable variables (e.g., demographic data). However, as most of the software engineering artifacts are archived, we envision that some of the commonly unobservable factors that will be useful in developer information support can be identified from other sources. For example, the history of software systems, issue logs, continuous integration logs, or code reviews in version control systems (e.g., GitHub) will be useful in inferring both the history of the developer and what they work on. From the developer's IDE, the current project, code-base, and interaction traces can be collected to understand the developer's current context. For contextual variables that are not easily inferred from any of the sources, it might also be possible to ask users directly.

Variability. All software evolves, as do its developers, so most contextual factors change over time. Thus, the context might also be re-inferred or updated as the software and developer change. Knowing how often they get updated is necessary to properly update the context, as they can vary by context factors. For example, the job title of a developer rarely changes, but their familiarity with a certain API method will be updated fairly frequently.

Software Artifact ← IDE logs, search queries, prompts

Software artifacts, including code snippets, test cases, or error messages can be extracted from the local file systems, online source repositories, continuous integration tools, and input/output of terminals. In addition to the above, the developers' IDE interaction history will also be useful in understanding the developer's interest (e.g., focal points in code) and clarify the relevancy of software artifacts to the current task of the developer. However, software artifacts can frequently change, as developers work on multiple tasks [85]. Sometimes, developers might work on multiple tasks concurrently [114], so inferring the relevant software artifacts to include will be more challenging than identifying appropriate context of other types.

Project Factor ← design documents, backlogs

Project context, such as the length of the project or technical infrastructure, does not change often or changes slowly, especially when the project is well-planned. Technical infrastructure, like programming languages, libraries, their versions in use, operating systems, and devices, will be easier to extract. Many of the technical infrastructure can be inferred from the configuration and build files, such as `pom.xml` for Maven or `package.json` or `tsconfig.json` for Typescript. Technical infrastructure does not often change after the software design phase, unless there are large refactorings or significant changes in requirements. Project scale, size, and domain can be easily inferred if there are written requirements or design documents, and the status can be inferred from the backlogs or task management tools (e.g., Kanban boards). However, when the project plan is not well documented (e.g., when it is initiated in a casual setting and grows organically), it is harder to infer the project scale or size, and explicit user inputs will be necessary.

Personal Factor ← user input, IDE logs, personal repo.

The simplest way of extracting personal factors, including the developer's expertise in programming, application domains, libraries, tools, and roles, is to explicitly ask the developer to provide the information. As backgrounds or preferences will only rarely change and can be easily answered by users, it might be worth asking them when they start using the information support, as many recommendation systems (e.g., Apple Music) do when users sign up. The full personal context may not be obvious if not provided by the developers, but parts are still inferable from external sources. Personal repositories in a version control system or professional networking platforms will be useful in inferring the expertise levels, skills, and backgrounds. They contain the history of the developer's programming works from which their use of programming languages, libraries, design patterns, and API methods can be easily inferred. In capturing developer backgrounds as well as their skills and expertise, LinkedIn or resumes will be great sources.

8.5 Summary

In this chapter, we identified the information needs of programmers and the contextual factors programmers use in the LLM-powered information support setting by analyzing the ChatGPT prompts used in the software engineering context. We further conducted an experimental ablation study to measure the importance of including such contextual factors in prompts, in enhancing the LLM response quality. The study provided promising preliminary evidence that adding contextual factors can be useful in providing better information support based on criteria such as helpfulness, relevance, accuracy, depth, creativity, and detail. We believe that this chapter motivates the appropriate augmentation of diverse contextual factors in providing LLM-powered information support for programmers, and provides a good starting point for the future generation of information support that is more contextualized and, thus, more effective.

Chapter 9

Conclusion & Future Work

9.1 Summary of Contributions

This thesis makes a number of major contributions to enhancing information support for programmers' learning, including:

- A thorough review of the background and related work on programmers' information seeking in general, and the various information support approaches (Chapter 2).
- Evidence that programmers' different use of documentation correlates with their user characteristics (e.g., experience level with the API), and with their future API adoption (Chapter 3).
- A novel approach for documentation design review using page-view log analysis (Chapter 3).
- MARBLE, an automatic approach to identify boilerplate code from API client code (Chapter 4).
- A dataset of boilerplate code for 13 popular Java libraries (Chapter 4).
- Evidence that push-based comparable API methods information support can enhance programmers' understanding of API design space (Chapter 5).
- A prototype information support tool that presents comparable API methods within Chrome browser (Chapter 5).
- SOREL, a learning-based model extracting comparable API methods and the support evidence from Stack Overflow posts (Chapter 5).
- A dataset of comparable API methods including 587 Stack Overflow answers, with 198 pairs of comparable API methods and 737 sentences summarizing the relations (Chapter 5).
- Showing the feasibility of providing programming by example support for real-world library users to mitigate the issue of the lexical gap (Chapter 6).
- A language model, which can predict a sequence of API methods given input and output value pairs (Chapter 6).

- Evidence that in-IDE, context-aware, prompt-less support using an LLM can help programmers working with new APIs complete more tasks (Chapter 7).

- A prototype in-IDE information support tool called GILT, that generates on-demand information using an LLM while considering the user' s local code context (Chapter 7).

- A novel experimental approach to measure the value of adding different contextual factors into LLM prompts in the response quality (Chapter 8).

- A preliminary evidence that including contextual factors are useful in enhancing the LLM response quality (Chapter 8).

9.2 Discussion & Future Work

Efficient and effective information seeking for programming is still a largely unresolved problem, requiring lots of research. *How* different programmers seek information is still an open problem, especially with the introduction of LLM-based information seeking, which totally changes the strategies and behaviors of programmers. There are many practical future work items, including extending the prototype tools to support other languages or libraries, but in this section, I focus on future directions that require collaborative efforts.

More Personalized and Broader Information Support

In this thesis, I have concentrated on providing information support for professional programmers dealing with code that involves unfamiliar APIs and concepts. This work represents an initial exploration into the design of intelligent information support tools. Moving forward, it is imperative to consider more diverse scenarios (e.g., software design, refactoring) and users (e.g., end-user programmers and software architects). To support more diverse programmers in information seeking, it will be necessary to first understand the specific challenges they face with the intelligent information support tools. In-depth analysis with end-user programmers or programmers with specific roles can be conducted, like those reported in Chapter 7. To support broader tasks, information categories that have not covered in this thesis should also be prepared, beyond learning resources like documentation or usage examples. The various information needed for tasks like system design, debugging, refactoring, and testing, often involving multiple source files, may be prepared with program analysis techniques, in addition to the learning-based approaches trained with code data.

To provide support for broader tasks and users, further research is required to determine the extent and nature of the additional context needed, as its impact on response quality was highlighted in Chapters 7 and 8. Particularly, there is a need to develop a better understanding of the appropriate scope of context, to build sophisticated methods that can accurately predicting user intent, task scope, and relevant context for effective augmentation. For this, we advocate for a larger-scale version of our experimental ablation study, as detailed in Chapter 8, using more recent data that mirrors real-world usage patterns rather than exploratory usage of LLMs. Such a study would be invaluable in identifying common strategies among programmers and understanding the relationship between the necessary context and their information needs.

Smooth Transitions of Programming Tools with New Intelligent Solutions

Research on integrating LLMs within the framework of existing programming tools, originally developed under more traditional paradigms, presents numerous opportunities to facilitate smoother transitions for programmers. Adopting interfaces with which programmers are already familiar is a good strategy, as it will leverage existing user comfort while introducing advanced capabilities. For instance, the introduction of prompt-less interactions in Chapter 7 in GILT proved effective, as programmers were already familiar with retrieving some information with a shortcut or a button, by using static analysis tools within IDE. Continuing this exploration by further integrating context-sensitive and user-friendly interfaces can significantly reduce the cognitive load on programmers as they adopt new programming tools.

Moreover, incorporating ideas used in more conventional programming tools into LLM-powered tools can enhance the experience for programmers familiar with chat-based interfaces. For example, although the SOREL system, described in Chapter 5, targets programmers who primarily seek information through search engines, the concept of push-based information delivery can be adapted for chat-based interfaces. This adaptation could involve suggesting follow-up queries or offering additional information proactively, even if the user has not explicitly requested it. Ideas like programming-by-example support in Chapter 6, and Chapter 4' s discussion on the utility of presenting boilerplate code, are also transferable across various interface designs. Reusing such strategies will provide opportunities to improve the usability and effectiveness of existing chat-based LLM tools.

User interfaces for intelligent information support systems also require research. Through our analysis, we observed that how and when information is presented can influence the benefit (e.g., Chapter 5), and can even benefit programmers differently (e.g., Chapter 3 and 7). Although some researchers have explored different design options in designing LLM-powered programming tools [21, 275], I believe more exploration is needed to support diverse populations. Specifically, the chat interface, which is commonly used for LLM-powered tools, is known to introduce challenges when the users are not already familiar with the application domains. Given that programming tools often have access to many contextual factors, more natural interfaces can be built by using such additional factors to infer users' information needs and intents. We believe that further research is needed, exploring various interaction options to support a diverse programmer population.

Understanding evolving information-seeking strategies among programmers using generation-based information support

Recent developments in LLM-powered programming tools have introduced a new paradigm for programmer information seeking, allowing for the generation rather than retrieval of information. Despite this, as discussed in Chapters 7 and 8, programmers still employ and need traditional information-seeking strategies similar to those used in conventional settings, including skills like query revision to elicit useful responses from LLMs.

We hypothesize that the continued reliance on traditional strategies may be attributed to the novelty of LLM-powered tools, with programmers still in the early stages of adapting to these new technologies. As prompt engineering techniques improve and guidelines for effective interaction with LLMs become more widespread, we anticipate that programmers might adopt entirely new strategies. Moreover, we predict that these strategies could vary significantly across different generations of programmers, ranging from senior software engineers accustomed to retrieval-based searches to junior engineers who are more familiar with generation-based approaches. Thus, we believe that understanding how these strategies evolve will be crucial for developing LLM tools that provide effective support for programmers. This will not only enhance the productivity of current programmers but also shape the training and integration of future professionals in the field.

Next-generation SE with Programmer-AI partnership

In addition to the aforementioned future research directions, it is also important for us to understand what *can* be automated by intelligent solutions, and what *should* still be done by programmers, because, eventually, the software systems will be maintained and used by people, and it is important for programmers to have an understanding of and control over the systems they build. At the same time, it will be important to design and build the next-generation software engineering processes, by answering questions like “What processes should we use to ensure the safety, robustness, and privacy of software systems built with AI tools?” Programmer-AI alignment should also be studied further, by answering “How can we design better evaluation and monitoring approaches to align AIs with SE experts?” and “How can we prevent misspecification and misgeneralization?” Finally, to guide the advancement of intelligent solutions to be aligned with what programmers need, further research is needed on how we evaluate the models. Specifically, building benchmarks that are more grounded towards real-world software engineering will be beneficial in extending the programming benchmarks, like HumanEval [44] or MBPP [22], that are widely used to evaluate foundation models and provide evaluations that are better aligned with real-world tasks.

Inter-community Collaborations for Future Programming Paradigms

Beyond the goals above, it is necessary for NLP, HCI, and SE research to forge future programming tools that are not only accurate but also highly usable. Given that the forthcoming years will witness a surge in tool development, potentially reshaping the programming landscape, it is imperative that NLP researchers incorporate actual user needs into their model designs, and that tool creators actively address assumptions and challenges posed by machine learning models.

9.3 Concluding Remarks

Software engineering is an information-intensive discipline. While building and maintaining software systems, programmers face a broad spectrum of questions ranging from implementation specifics to architectural concerns. However, satisfying their information needs is not easy, because the relevant information is often scattered across varying mediums in different formats. It becomes even more challenging when a programmer needs to work with unfamiliar code or libraries, without the necessary knowledge and experience to search for information effectively.

To mitigate some of the challenges in information seeking for programming, my research has investigated designs, algorithms, and evaluation approaches for intelligent information support for programmers. By combining multiple research methods at the intersection of Software Engineering, Human-Computer Interaction, and Natural Language Processing, I gained a richer understanding of the information seeking of programmers as users and also found many insights on how we can build better intelligent techniques to provide information support. For example, Chapter 5 not only demonstrated that the push-based interaction support can help programmers understand the API design space, but also built a learning-based model that can automatically extract comparable API methods information from Stack Overflow. Or, in Chapter 7, by building GILT and evaluating it with programmers, we could understand how programmers use LLM-powered information support tools for code understanding, and showed that utilizing context can be useful in information support, which led to the follow-up study of contextual factor investigation. The series of projects in this dissertation, involving iterations of user studies and tool building, demonstrates the importance of having user-centered tool support that helps information seeking for programming. It has been shown that user-centered intelligent support not only yields valuable insights for designing more useful and usable tools but also improves the performance of intelligent techniques used in these tasks.

As technology evolves at an unprecedented pace, there is a substantial effort being made to create intelligent programming tools. However, there is still a disconnect in understanding how programmers will use these tools and the implications of using them in real-world software engineering. Like what I have demonstrated with this dissertation, the synergy that could be achieved by studying both users and tools together holds immense potential. Thus, it is my hope that more intelligent technique development efforts will adopt a user-centered approach. By placing the users, here programmers, at the heart of tool creation, we can ensure that the technological advancements we make are not only innovative but also practical and impactful for the programmers who will use them, and eventually, the users of the software systems built by the programmers.

Appendix A

More Results for Logs Analysis

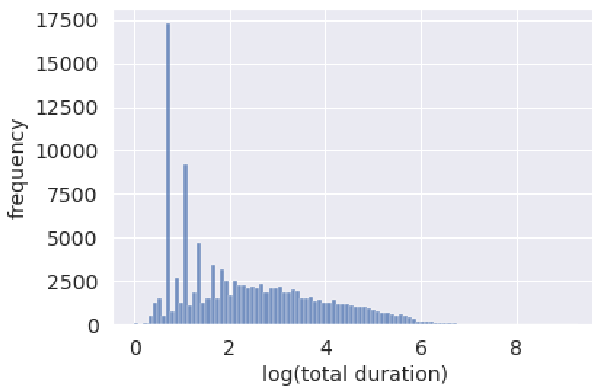


Figure A.1: Distribution of the log-transformed total dwell time (in minutes) on documentation.

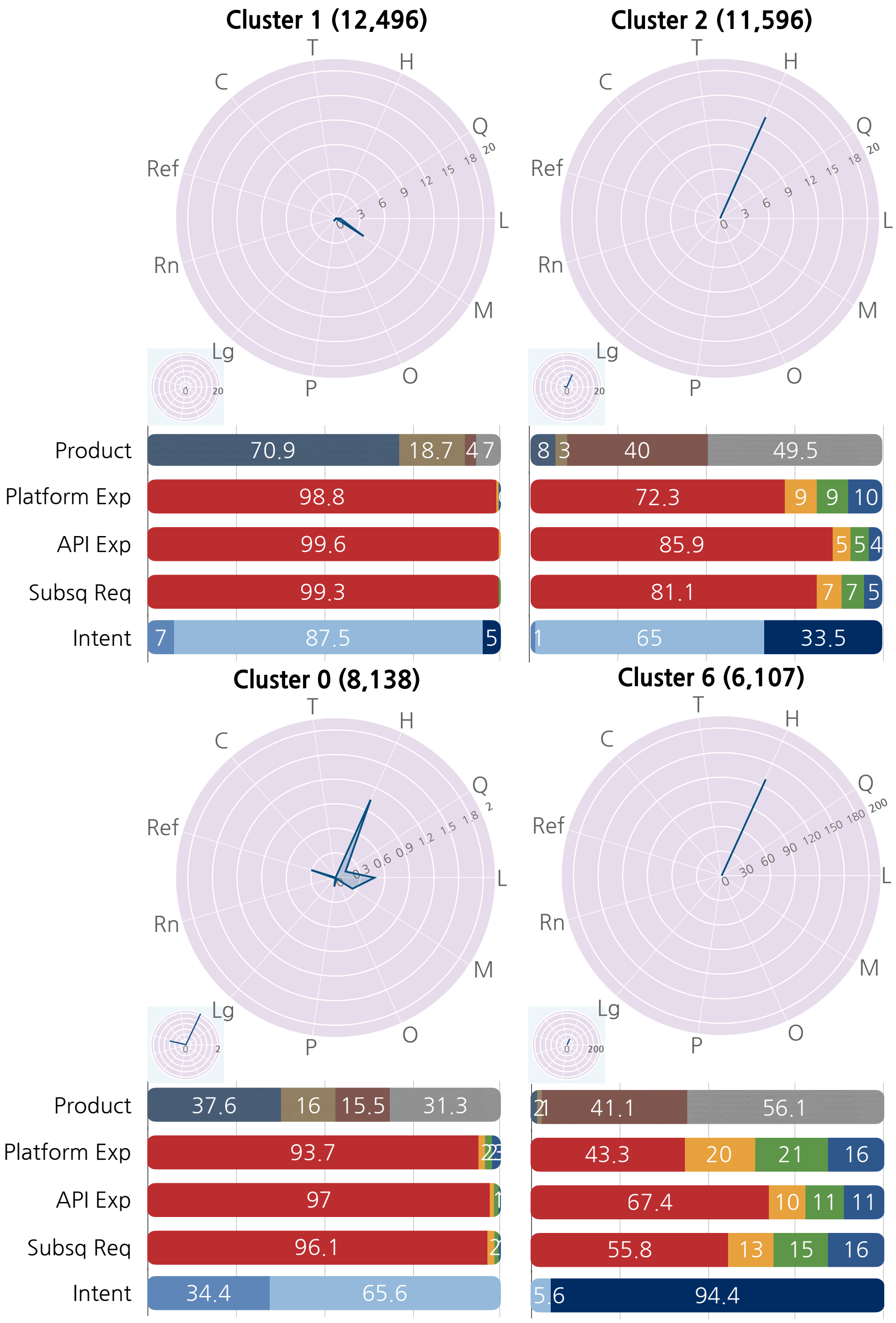


Figure A.2: Large clusters with more than 500 users sorted by the number of users (shown in parentheses). Each polar plot displays the average time spent on each type of documentation (T: Tutorial, H: How-to, Q: Quickstart, L: Landing, M: Marketing, O: Other, P: Pricing, Lg: Legal, Rn: Release note, Ref: Reference, C:Concept). The small polar plots show the average dwell time in the previous three months. Note that the ranges of the axes of the plots vary. Bar charts below the polar plots show the proportions (%) of each group in the cluster.

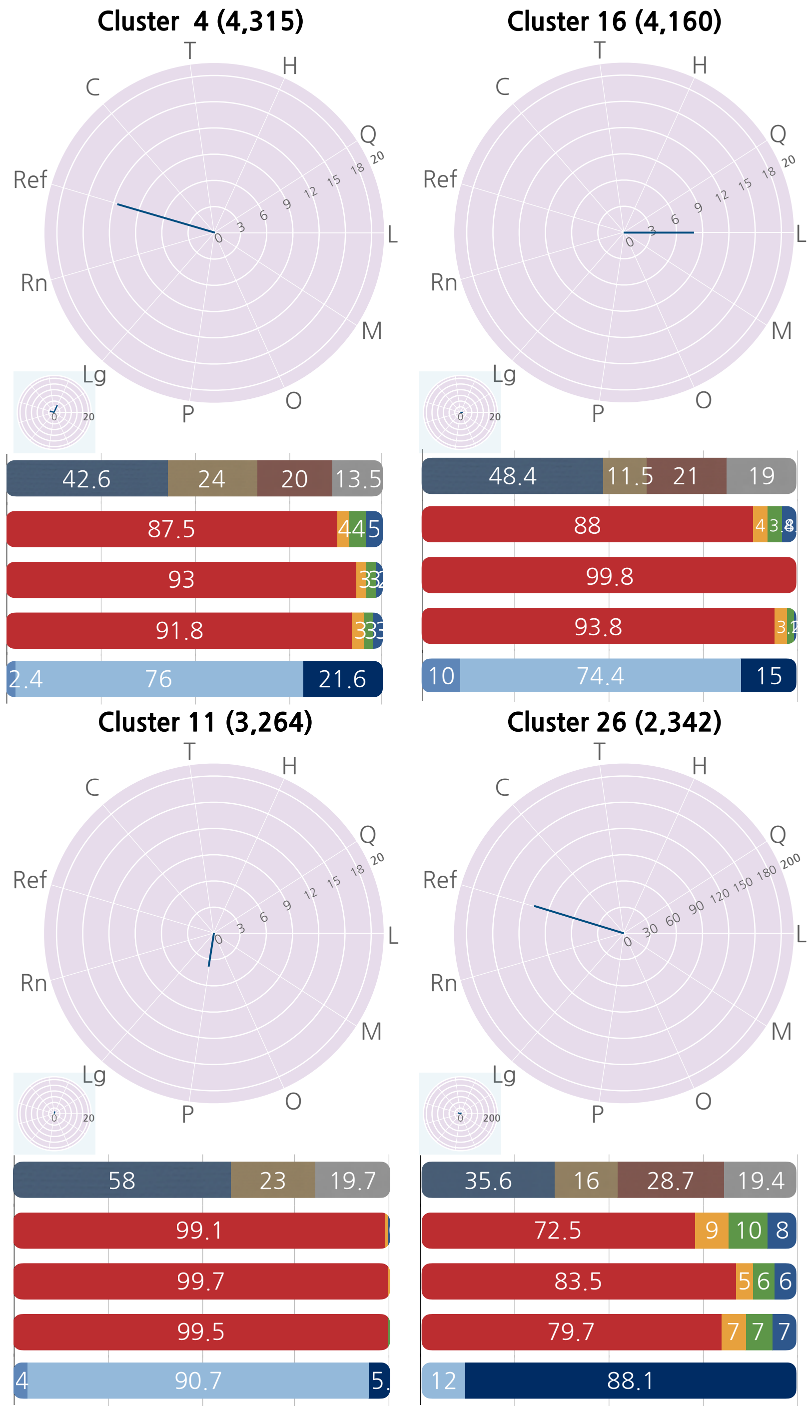


Figure A.3: Large clusters with more than 500 users sorted by the number of users (shown in parentheses). Each polar plot displays the average time spent on each type of documentation (T: Tutorial, H: How-to, Q: Quickstart, L: Landing, M: Marketing, O: Other, P: Pricing, Lg: Legal, Rn: Release note, Ref: Reference, C:Concept). The small polar plots show the average dwell time in the previous three months. Note that the ranges of the axes of the plots vary. Bar charts below the polar plots show the proportions (%) of each group in the cluster.

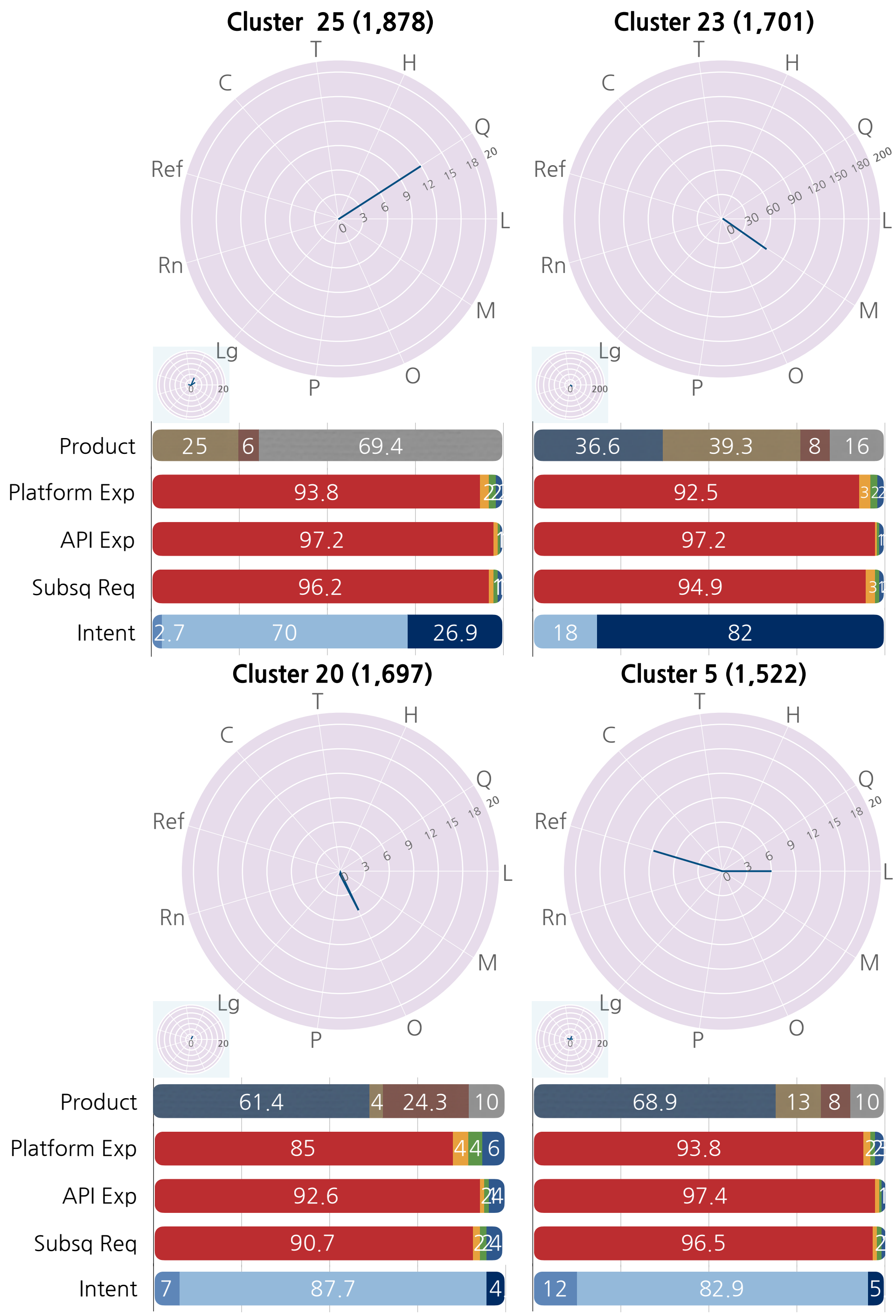


Figure A.4: Large clusters with more than 500 users sorted by the number of users (shown in parentheses). Each polar plot displays the average time spent on each type of documentation (T: Tutorial, H: How-to, Q: Quickstart, L: Landing, M: Marketing, O: Other, P: Pricing, Lg: Legal, Rn: Release note, Ref: Reference, C:Concept). The small polar plots show the average dwell time in the previous three months. Note that the ranges of the axes of the plots vary. Bar charts below the polar plots show the proportions (%) of each group in the cluster.

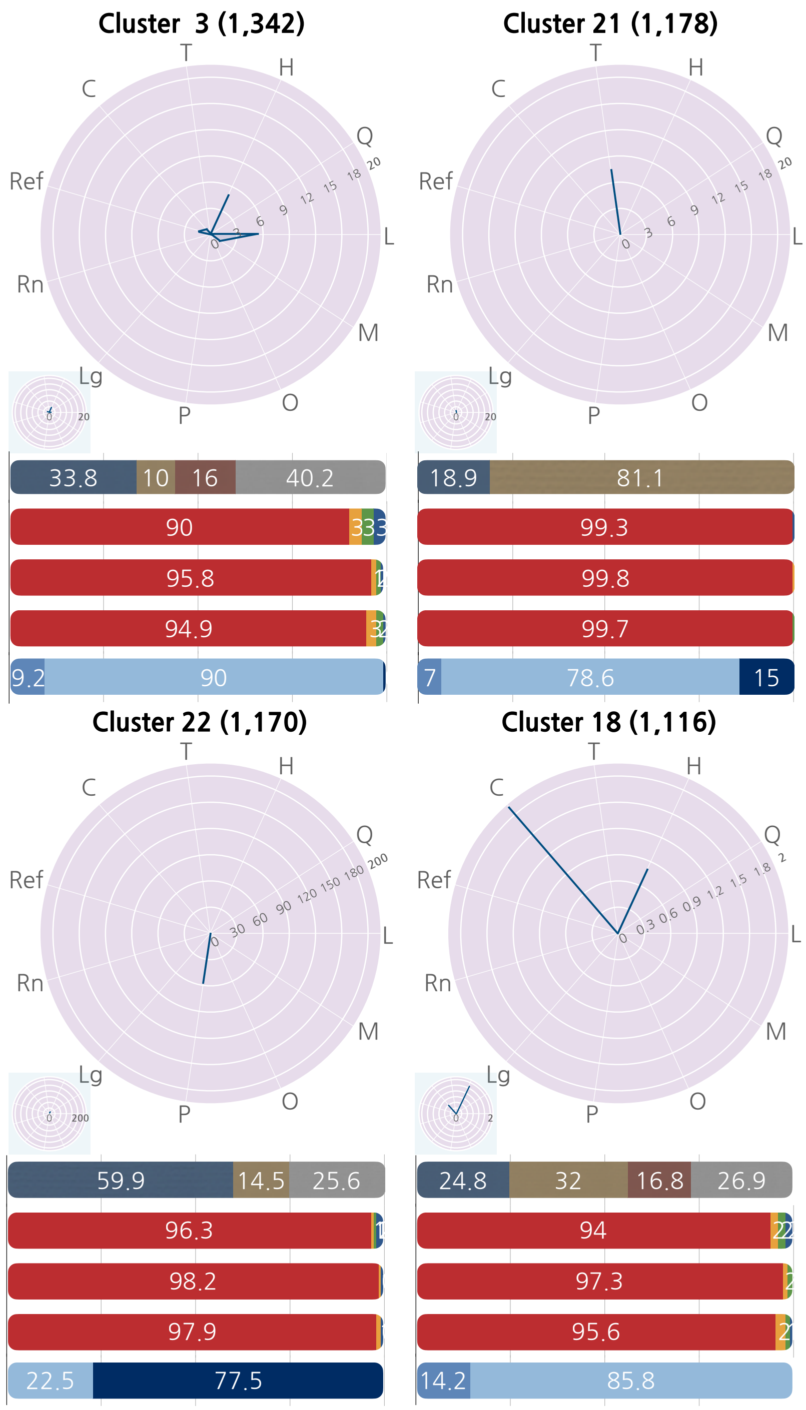


Figure A.5: Large clusters with more than 500 users sorted by the number of users (shown in parentheses). Each polar plot displays the average time spent on each type of documentation (T: Tutorial, H: How-to, Q: Quickstart, L: Landing, M: Marketing, O: Other, P: Pricing, Lg: Legal, Rn: Release note, Ref: Reference, C:Concept). The small polar plots show the average dwell time in the previous three months. Note that the ranges of the axes of the plots vary. Bar charts below the polar plots show the proportions (%) of each group in the cluster.

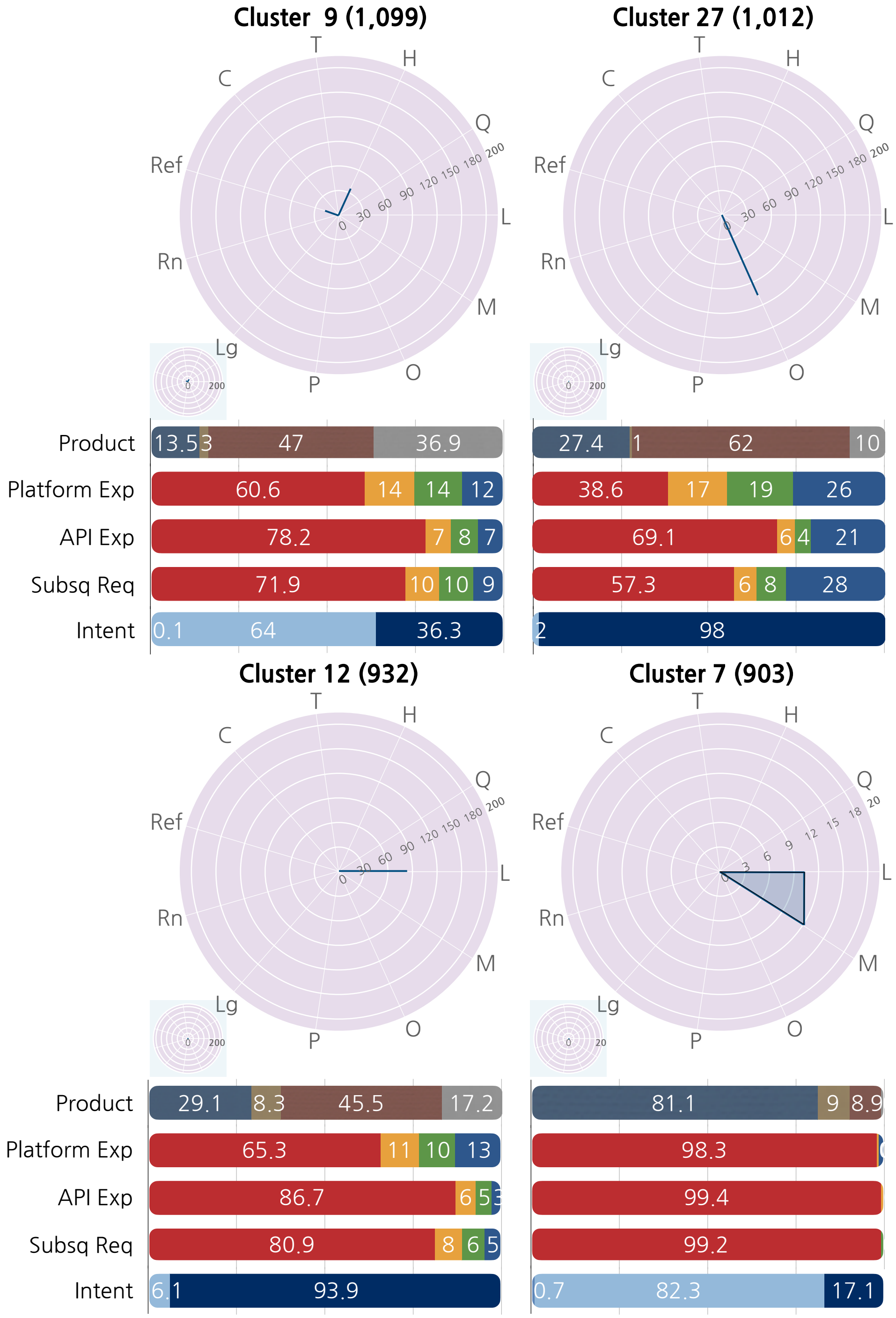


Figure A.6: Large clusters with more than 500 users sorted by the number of users (shown in parentheses). Each polar plot displays the average time spent on each type of documentation (T: Tutorial, H: How-to, Q: Quickstart, L: Landing, M: Marketing, O: Other, P: Pricing, Lg: Legal, Rn: Release note, Ref: Reference, C:Concept). The small polar plots show the average dwell time in the previous three months. Note that the ranges of the axes of the plots vary. Bar charts below the polar plots show the proportions (%) of each group in the cluster.

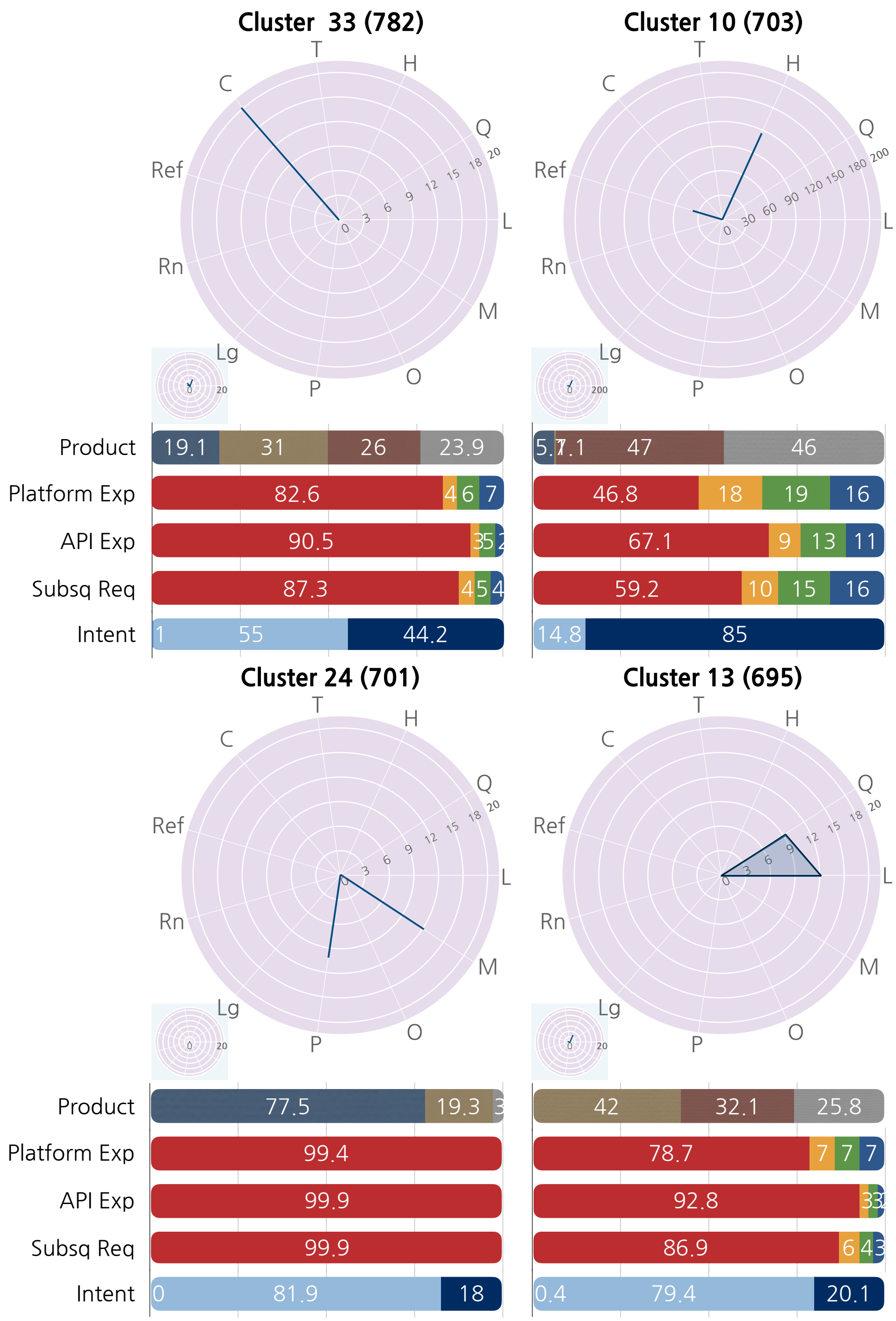


Figure A.7: Large clusters with more than 500 users sorted by the number of users (shown in parentheses). Each polar plot displays the average time spent on each type of documentation (T: Tutorial, H: How-to, Q: Quickstart, L: Landing, M: Marketing, O: Other, P: Pricing, Lg: Legal, Rn: Release note, Ref: Reference, C:Concept). The small polar plots show the average dwell time in the previous three months. Note that the ranges of the axes of the plots vary. Bar charts below the polar plots show the proportions (%) of each group in the cluster.

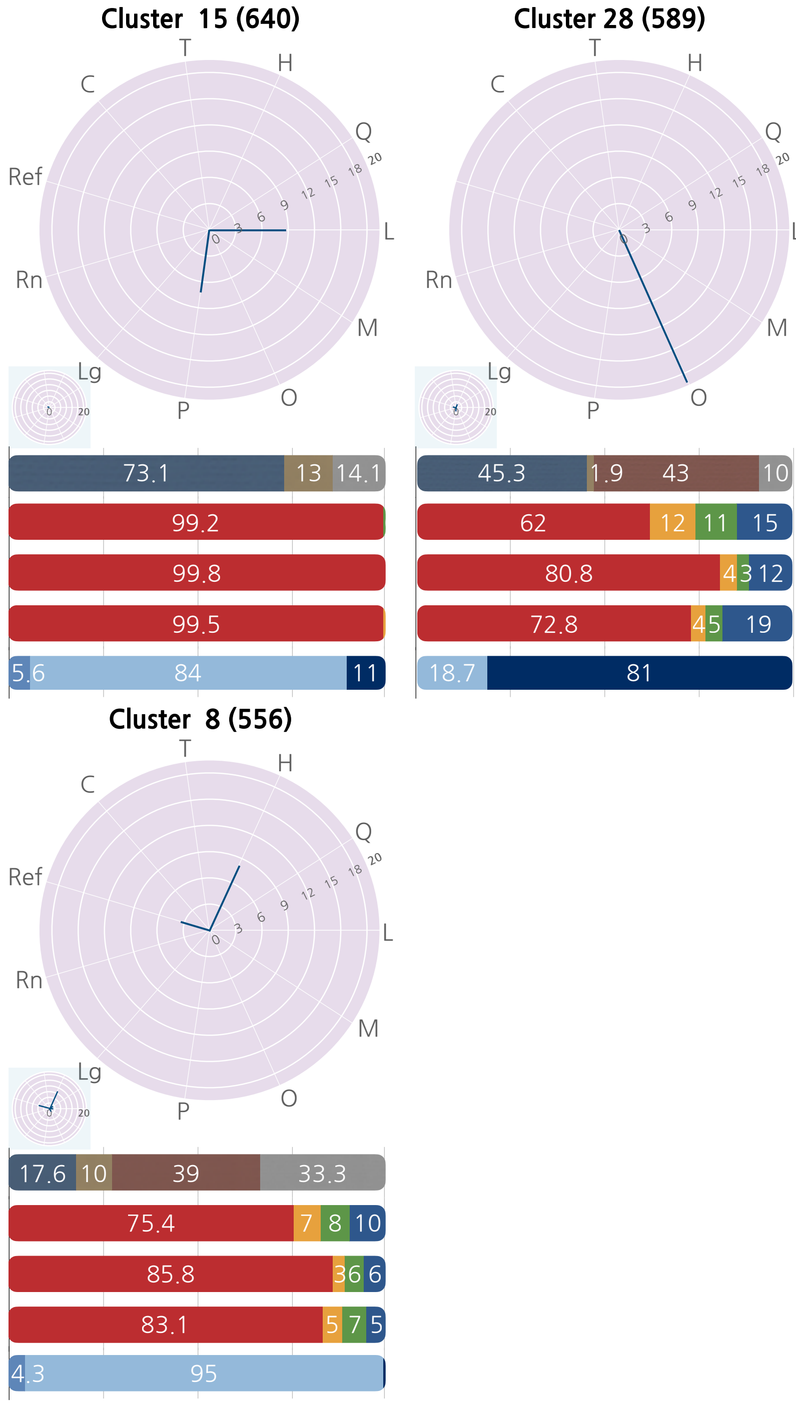


Figure A.8: Large clusters with more than 500 users sorted by the number of users (shown in parentheses). Each polar plot displays the average time spent on each type of documentation (T: Tutorial, H: How-to, Q: Quickstart, L: Landing, M: Marketing, O: Other, P: Pricing, Lg: Legal, Rn: Release note, Ref: Reference, C:Concept). The small polar plots show the average dwell time in the previous three months. Note that the ranges of the axes of the plots vary. Bar charts below the polar plots show the proportions (%) of each group in the cluster.

Appendix B

Experiment Details for Predictive Synthesis of API-Centric Code

B.1 Supported operations of PyTorch

Below is the list of 33 PyTorch operations. 16 operations used in the original dataset (described in Section 6.6.1) are highlighted.

- **add**
- **any**
- arange
- argmax
- **bincount**
- cdist
- div
- **eq**
- **expand**
- eye
- gather
- **gt**
- **lt**
- **masked_select**
- **matmul**
- max

- minimum
- **mul**
- **ne**
- one_hot
- repeat_interleave
- reshape
- roll
- searchsorted
- square
- squeeze
- **stack**
- sum
- **tensordot**
- tile
- **transpose**
- **unsqueeze**
- **where**

B.2 Stack Overflow Benchmarks

As mentioned in Section 6.6.1, we adapted the Stack Overflow benchmarks created for TF-Coder [229]. The examples were collected from Stack Overflow posts and the benchmarks were inspired by those posts. However, the input/output values were replaced by the TF-Coder authors for licensing reasons. The input/output values created by TF-Coder authors, and we updated some values to fit into our scope.

B.2.1 Input/output and Desired Code

	Inputs	Output	Code & Function Sequence	Expected Sequence
SO01	in1 = [[5, 2], [1, 3], [0, -1]]	[[[5, 5], [1, 1], [0, 0]], [[2, 2], [3, 3], [-1, -1]]]	torch.transpose(in1, expand((2, 3, 2)), 0, 2)	expand, transpose
SO02	in1 = [5, 1, 0, 3, 0, 0, 2, 0, 2] in2 = 1	[1, 1, 0, 1, 0, 0, 1, 0, 1]	torch.where(torch.lt(in1, 1), in1, 1)	lt, where
SO05	in1 = [[4, 3, 1], [6, 5, 2]] in2 = [[[5, 5]], [[1, 5]], [[6, 0]]]	[[[29, 35]], [[47, 55]]]	torch.tensordot(in1, in2, 1)	tensordot
SO06	in1 = [3, 5, 0, 2, 3, 3, 0]	[[1, 0, 0, 0, 1, 1, 0], [0, 1, 0, 0, 0, 0, 0], [0, 0, 1, 0, 0, 0, 1], [0, 0, 0, 1, 0, 0, 0], [1, 0, 0, 0, 1, 1, 0], [1, 0, 0, 0, 1, 1, 0], [0, 0, 1, 0, 0, 0, 1]]	torch.eq(in1, torch.unsqueeze(in1, 1))	unsqueeze, eq
SO07	in1 = [[[8, 4, 6], [2, 12, 3]], [[11, 12, 5], [9, 12, 12]], [[9, 2, 13], [7, 0, 7]], [[2, 10, 5], [7, 1, 2]]],	[[[8, 4, 6], [11, 12, 5], [9, 2, 13], [2, 10, 5]], [[2, 12, 3], [9, 12, 12], [7, 0, 7], [7, 1, 2]]]	torch.transpose(in1, 0, 1)	transpose
SO08	in1 = [1, 0, 0, 2, 1, 3, 5, 0, 1, 2, 10], in2 = [12, 3, 45, 6, 7, 8, 9, 87, 65, 4, 32], in3 = 1	[6, 8, 9, 4, 32]	torch.masked_select(in2, torch.gt(in1, 1))	gt, masked_select
SO11	in1 = [4, 0, 1, 1, 0, 4, 0, 0, 3, 4, 1]	[4, 3, 0, 1, 3]	torch.bincount(in1)	bincount
SO13	in1 = [[3, 5], [10, 2]], in2 = [[[1, 0], [5, 4]], [[3, 10], [2, 0]]]	[[[28, 20], [19, 30]], [[20, 8], [34, 100]]]	torch.transpose(torch.matmul(in1, in2), 0, 1)	matmul, transpose
SO14	in1 = [[[0, 0, 1], [0, 0, 0], [1, 0, 1], [0, 1, 0], [0, 0, 0], [1, 1, 1], [1, 1, 0]]]	[[1, 0, 1, 1, 0, 1, 1]]	torch.any(in1, -1)	any
SO15	in1 = [3, 1, 2, 0, 1, 0, 10, 1, 0]	3, 0, 2, 0, 0, 0, 10, 0, 0]	torch.mul(in1, torch.ne(in1, 1))	ne, mul
SO16	in1 = [[2, 5], [3, 0], [8, 7]], in2 = [4, 10, 6]	[[8, 20], [30, 0], [48, 42]]	torch.mul(in1, torch.unsqueeze(in2, 1))	unsqueez, mul
SO17	in1 = [17, 32, 99]	[[17, 17], [32, 32], [99, 99]]	torch.stack((in1, in1), 1)	stack
SO18	in1 = [[[1, 1, 1], [1, 0, 1]], [[1, 2, 3], [4, 5, 6]]], in2 = [[1, 1, 1, 1], [1, 2, 3, 4], [5, 6, 7, 8]], in3 = [100, 200, 300, 400],	[[[107, 209, 311, 413], [106, 207, 308, 409]], [[118, 223, 328, 433], [139, 250, 361, 472]]]	torch.add(in3, torch.matmul(in1, in2))	matmul, add
SO20	in1 = [[7, 2, 1], [4, 5, 1], [4, 4, 2], [3, 4, 3], [0, 0, 1]]	[[1, 0, 0], [0, 1, 0], [1, 0, 0], [0, 1, 0], [0, 0, 1]]	torch.nn.functional.one_hot(torch.argmax(in1, 1), 3)	argmax, one_hot
SO21	in1 = [[2], [0], [1], [0]], in2 = [[2, 5, 3], [1, 3, 6], [1, 6, 3], [7, 0, 3]]	[[3], [1], [6], [7]]	torch.gather(in2, 1, in1)	gather
SO22	in1 = [3, 1, 10], in2 = [[6, 4], [5, 1], [3, 4]]	[53, 53]	torch.tensordot(in1, in2, 1)	tensordot
SO23	in1 = [[0, 5, 2], [3, 1, 4], [5, 1, 5]],	[[1, 0, 1, 0, 0, 1, 0, 0, 0], [0, 1, 0, 1, 1, 0, 0, 0, 0], [0, 1, 0, 0, 0, 1, 0, 0, 0]]	torch.sum(torch.nn.functional.one_hot(in1, 9), 1)	one_hot, sum

SO24	in1 = [3, 1, 4, 5, 2, 8, 6, 7], in2 = [1, 0, 2, 0, 1, 1, 0, 2], in3 = 0	[3, 1, 2, 5, 2, 8, 6, 3.5]	torch.where(torch.ne(in2, in3), torch.div(in1, in2), in1)	div, where
SO25	in1 = 3 in2 = 4	[[1, 0, 0], [0, 1, 0], [0, 0, 1], [1, 0, 0], [0, 1, 0], [0, 0, 1], [1, 0, 0], [0, 1, 0], [0, 0, 1], [1, 0, 0], [0, 1, 0], [0, 0, 1]]	torch.tile(torch.eye(in1), (in2, 1))	eye, tile
SO26	in1 = [[[[3, 4], [1, 2]], [[5, 2], [10, 3]], [[10, 20], [4, 7]]]]	[10, 20, 41]	torch.sum(torch.flatten(in1, 1), 1)	sum, sum
SO27	in1 = [0, 3, 5, 6], in2 = 8	[1, 0, 0, 1, 0, 1, 1, 0]	torch.sum(torch.nn.functional.one_hot(in1, in2), 0)	one_hot, sum
SO29	in1 = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21], in2 = [12, 0, 10, 23, 16],	[6, 0, 5, 11, 8]	torch.searchsorted(in1, in2)	searchsorted
SO30	in1 = [[1., 2.], [3., 4.], [5., 6.]], in2 = [[9., 4.], [8., 5.], [7., 6.]],	[[math.sqrt(68), math.sqrt(58), math.sqrt(52)], [math.sqrt(36), math.sqrt(26), math.sqrt(20)], [math.sqrt(20), math.sqrt(10), math.sqrt(4)]]	torch.cdist(in1, in2)	cdist
SO32	[[1, 6, 2, 1], [3, 1, 4, 2], [2, 1, 2, 5]]	[1.3, 1.5, 2.0]	torch.tensordot(in1, torch.arange(4), 1)	arange, tensordot
SO34	in1 = [[[1, 2], [3, 4]], [[5, 6], [7, 8]], [[10, 20], [30, 40]]], in2 = [3, 5, 10],	[[128, 236], [344, 452]]	torch.tensordot(in2, in1, 1)	tensordot
SO36	in1 = [1, 0, 1, 1, 0, 1, 0, 1],	[1., 0., 0.333333, 0.25, 0., 0.166667, 0., 0.125]	torch.div(in1, torch.add(in1, torch.arange(8)))	arange, add, div
SO37	in1 = [[[[10, 20, 30], [40, 50, 60]], [[12, 34, 56], [78, 98, 76]]]], in2 = [5, 10, 20],	[[[850, 1900], [1520, 2890]]]	torch.tensordot(in1, in2, 1)	tensordot
SO39	in1 = [[15, 10, 9, 20], [11, 0, 1, 9], [10, 1, 11, 25]]	[[225, 100, 81, 400], [121, 0, 1, 81], [100, 1, 121, 625]]	torch.square(torch.mul(in1, in1))	mul, square
SO41	in1 = [5, 2, 8, 2, 4, 1, 1, 0, 2, 1], in2 = 3	[5, 2, 8, 4, 1, 1, 0, 2, 1]	torch.masked_select(in1, torch.ne(torch.arange(10), in2))	arange, ne, masked_select
SO42	in1 = [4, 6, 2, 6, 7, 3, 3] in2 = 7	[0, 0, 0, 0, 1, 0, 0]	torch.eq(in1, 7)	eq
SO44	in1 = [[3, 5, 2], [6, 2, 3], [8, 7, 1], [0, 3, 5], [4, 7, 3], [2, 1, 6], [10, 20, 30], [4, 5, 6]]	[[9, 7, 5], [8, 19, 6], [6, 8, 9], [14, 25, 36]]	torch.sum(torch.reshape(in1, (-1, 2, 8)), 1)	reshape, sum
SO45	in1 = [[[12, 34], [56, 78], [23, 54], [76, 78], [42, 24]]], in2 = [1, 0, 1, 0, 1],	[[[34, 12], [56, 78], [54, 23], [76, 78], [24, 42]]]	torch.where(torch.unsqueeze(in2, 1), torch.roll(in1, 1, -1), in1)	roll, unsqueeze, where
SO46	in1 = [3, 4, 1],	[0, 0, 0, 1, 1, 1, 1, 2]	torch.repeat_interleave(torch.arange(3), in1, 0)	arange, repeat_interleave
SO48	in1 = [32, 53, 45, 38, 29, 89, 64, 23], in2 = [38, 53, 89, 38, 32, 64],	[3, 1, 5, 3, 0, 6]	torch.argmax(torch.eq(in1, torch.unsqueeze(in2, 1)).float(), 1)	unsqueez, eq, argmax

SO49	in1 = [[[[1, 2, 3], [4, 5, 6]]], [[[8, 10, 0], [6, 4, 2]]], [[[9, 8, 7], [1, 2, 3]]],], in2 = [20, 5, 10],	[[[[20, 40, 60], [80, 100, 120]]], [[[40, 50, 0], [30, 20, 10]]], [[[90, 80, 70], [10, 20, 30]]]]	torch.transpose(torch.mul(torch.transpose(in1, 0, 3), in2), 0, 3)	transpose, mul, transpose
SO50	in1 = [3]	[[0, 0, 0, 1, 0, 0], [0, 0, 0, 1, 0, 0], [0, 0, 0, 1, 0, 0], [0, 0, 0, 1, 0, 0], [0, 0, 0, 1, 0, 0]]	torch.nn.functional.one_hot(in1.expand(in1,)) in2)	expand, one_hot

B.2.2 Links to Original StackOverflow Posts

- https://stackoverflow.com/questions/40441503/tensorflow-tensor-reshape
- https://stackoverflow.com/questions/46408839/tensorflow-trim-values-in-tensor
- https://stackoverflow.com/questions/43067338/tensor-multiplication-in-tensorflow
- https://stackoverflow.com/questions/47816231/create-binary-tensor-from-vector-in-tensorflow
- https://stackoverflow.com/questions/38212205/swap-tensor-axes-in-tensorflow
- https://stackoverflow.com/questions/33769041/tensorflow-indexing-with-boolean-tensor
- https://stackoverflow.com/questions/45194672/how-to-count-elements-in-tensorflow-tensor
- https://stackoverflow.com/questions/50777704/n-d-tensor-matrix-multiplication-with-tensorflow
- https://stackoverflow.com/questions/35657003/aggregate-each-element-of-tensor-in-tensorflow
- https://stackoverflow.com/questions/39045797/conditional-assignment-of-tensor-values-in-tensorflow
- https://stackoverflow.com/questions/46240646/tensor-multiply-along-axis-in-tensorflow
- https://stackoverflow.com/questions/51761353/about-tensor-of-tensorflow
- https://stackoverflow.com/questions/38222126/tensorflow-efficient-way-for-tensor-multiplication
- https://stackoverflow.com/questions/44834739/argmax-on-a-tensor-and-ceiling-in-tensorflow
- https://stackoverflow.com/questions/51690095/how-to-gather-element-with-index-in-tensorflow
- https://stackoverflow.com/questions/43284897/how-can-i-multiply-a-vector-and-a-matrix-in-tensorflow-without-reshaping
- https://stackoverflow.com/questions/53414433/tensorflow-tensor-binarization
- https://stackoverflow.com/questions/53643339/tensorflow-overriding-tf-divide-to-return-the-numerator-when-dividing-by-0
- https://stackoverflow.com/questions/53602691/duplicate-a-tensor-n-times
- https://stackoverflow.com/questions/54294780/how-to-perform-reduce-op-on-multiple-dimensions-at-once

- https://stackoverflow.com/questions/54225704/how-do-i-get-a-tensor-representing-the-on-positions-in-the-original-tensor
- https://stackoverflow.com/questions/54155085/bucketing-continous-value-tensors-in-tensorflow
- https://stackoverflow.com/questions/54147780/tensorflow-how-to-calculate-the-euclidean-distance-between-two-tensor
- https://stackoverflow.com/questions/48659449/how-to-compute-the-weighted-sum-of-a-tensor-in-tensorflow
- https://stackoverflow.com/questions/49532371/compute-a-linear-combination-of-tensors-in-tensorflow
- https://stackoverflow.com/questions/43306788/divide-elements-of-1-d-tensor-by-the-corrispondent-index
- https://stackoverflow.com/questions/49206051/multiply-4-d-tensor-with-1-d-tensor
- https://stackoverflow.com/questions/37912161/how-can-i-compute-element-wise-conditionals-on-batches-in-tensorflow
- https://stackoverflow.com/questions/54499051/elegant-way-to-access-python-list-and-tensor-in-tensorflow
- https://stackoverflow.com/questions/54493814/binary-vector-of-max
- https://stackoverflow.com/questions/54402389/sum-the-columns-for-each-two-consecutive-rows-of-a-tensor-of-3-dimensions
- https://stackoverflow.com/questions/54337925/reverse-order-of-some-elements-in-tensorflow
- https://stackoverflow.com/questions/58652161/how-to-convert-2-3-4-to-0-0-1-1-1-2-2-2-2-to-utilize-tf-math-segment-sum
- https://stackoverflow.com/questions/58481332/getting-the-indices-of-several-elements-in-a-tensorflow-at-once
- https://stackoverflow.com/questions/58466562/given-a-batch-of-n-images-how-to-scalar-multiply-each-image-by-a-different-scal
- https://stackoverflow.com/questions/58537495/tensorflow-initialize-a-sparse-tensor-with-only-one-line-column-not-zero

B.3 Implementation of ML Models

We implement the model in Python using the PyTorch. We describe some implementation details here, but the code will be shared after anonymous period is over.

Encoding. Encoding of a tensor requires three encodings, separated by a separator: value, size, and type. The max sizes of the encodings are 150, 5, 3, respectively. Our approach supports up to 3 input tensors and one output tensor, and each tensor is separated by a separator, which makes the the size of input encoding to be 640 ($4 * (150 + 1 + 5 + 1 + 2 + 1)$).

Compositional Model. For the embedding, we use the feed forward network, that is identical to the classification model, which is trained jointly with a bi-RNN model. The embedded input-output pair is passed to bi-RNN, having 1 hidden layer. To evaluate the model, we use a beam size of 3. We use the compositional model in two modes: "full sequence" mode, which returns the predicted API function sequence, and a "first-of-sequence" mode that returns only the first API function from the predicted sequence.

Multi-label Classification Model. For the weighted enumerative search with prioritization (Figure 6.1 (b)), we trained a simple multi-label classification model following DeepCoder, instead of TF-Coder which uses manually defined features (e.g., whether a value is a primitive) which we found less generalizable. We used the same model architecture with the classification model, but changed the last activation function into *sigmoid* for the multi-label classification. We trained this model with input-output of *sequences* of API functions.

Evaluation Metrics. We evaluate our models on the accuracy of their predictions. For the model accuracy with synthetic data, we check whether the model correctly predict the ground-truth APIs, and for the Stack Overflow benchmarks evaluation, we measure the rank of the correct prediction and extract top-1, top-3 and top-10 accuracy metrics.

B.4 Algorithms

Algorithm 1 Weighted Enumerative Synthesis

Input: A task specification input/output, (I, O)
Output: A program P such that $P(I) = O$

```
B ← {I, 0, −1, 1, ...}                                  ▷ Base values
E ← B                                                   ▷ Pool of values
Ops ← AssignOpCost(I, O)
for all v ∈ C do
    v.cost ← AssignValCost(v)
for C = 1 → max_cost do                                 ▷ Cost Budget
    for all op ∈ Ops do
        c ← op.cost
        n ← op.arity
        ▷ Partition cost budgets into n arguments
        for all [c_1, ..., c_n] ∈ partition(C − c, n) do
            for i = 1, ..., n do
                ▷ Collect values satisfying i-th arg cost budget
                A_i ← {e ∈ E | e.cost = c_i}
            for all args ∈ Π_i A_i do
                V ← Execute(op, args)                   ▷ Run op w/ args
                if V = O then return V.expr
                if V ∉ E then
                    V.cost ← C
                    E ← E ∪ {V}
return "Fail: reached maximum cost"
```

Algorithm 2 AssignOpCost

Input: A task specification input/output, (I, O)
Output: List of operations with costs Ops

```
for all op ∈ Ops do
    op ← preset_cost
if doModelPrioritization then
    candidate_ops ← MultiClassificationModel(I, O)
    for all op ∈ candidate_ops do
        op.cost ← op.cost ∗ reweight_multiplier
return Ops
```

Algorithm 3 Compositional Model - Full-Sequence

Input: A task specification input/output, (I, O)
Output: A program P such that $P(I) = O$

$B \leftarrow \{I, 0, -1, 1, ...\}$
$op\_seq \leftarrow CompositionalModel(I, O)$
$n \leftarrow \sum op_i.arity$
$args\_list \leftarrow \Pi_n B$
for all $args \in args\_list$ **do**
    $V \leftarrow Execute(op\_seq, args)$
    if $V = O$ **then return** $V.expr$
return ”Fail” or **start** ”Enumerative Search”

Algorithm 4 Compositional Model - First-Of-Sequence

Input: A task specification input/output, (I, O)
Output: A program P such that $P(I) = O$

$B \leftarrow \{I, 0, -1, 1, ...\}$
for $i = 0 \rightarrow k$ **do** ▷ Sequence of k operations
    $op_i \leftarrow CompositionalModel(V_{i-1}, I_i, O)$
    $n \leftarrow op_i.arity$
    $args\_list \leftarrow \Pi_n B$
    for all $args \in args\_list$ **do**
        $V_i \leftarrow Execute(op_i, args)$
        if $V_i = O$ **then return** $V.expr$
return ”Fail” or **start** ”Enumerative Search”

Appendix C

Additional Study Results for SOREL

C.1 Outcome Variables for Quantitative Analysis.

Table C.1: Outcome variables for quantitative analysis.

Variable	Details
Task completion time (`time`)	We measured the time from when a participant first enters a search query to when they finish submitting the solution.
Number of search queries (`queries`).	We counted the number of search queries participants wrote until they submitted the final answer, as an approximation of how easy it was to retrieve necessary information.
Number of pages visited (`pages`).	We measured the number of web pages visited by a participant, to approximate the efforts needed in discovery.
Prior knowledge of the task (`prior`).	After each task, we asked participants whether they had implemented similar code in the past, and if so, whether that helped the search or not. We asked the level of prior knowledge in four levels: no experience (0), vaguely remembering that they have worked on similar tasks, but not enough to be helpful (1), experience with similar tasks helped them complete the task (2), and prior knowledge helped them recollect the exact function name (3).
The correctness of the solutions given the task definition (correctness).	We tested whether a participant submits an API method that matches with our ideal solution, which we determined based on our expert knowledge of the tasks. The tasks were purposefully designed to have multiple potential solutions, with varying implication and trade-offs, to test whether participants are aware of the alternative solutions. Thus, "incorrect" solutions may not necessarily be "bad" solutions, and they can still meet the task requirements with different quality implications, such as performance, readability, or scalability.

Table C.1: Outcome variables for quantitative analysis (cont.).

Variable	Details
Awareness of the comparable API methods (`awareness`).	In each post-task interview, we tested whether a participant was aware of comparable API methods given the task context. When participants answered that they did not know other relevant API methods or admitted that they *guessed* the difference, we considered them to not be aware of the comparable methods. If they could elaborate on the differences, we considered that they were aware of the comparable API methods.
Understanding of difference (`understanding`).	In the post-task interviews, we also asked the participants to explain the differences between the comparable API methods. This is independent of the *awareness*, as one might already know about comparable API methods but not consider using them during the task. We measured the level of understanding with two levels: did not know the difference or guessed the difference (0), and describe at least one difference between the methods(1).

C.2 User Study Results

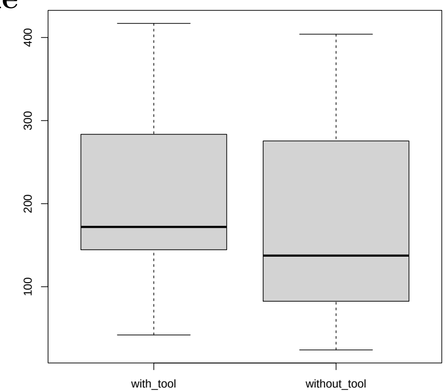
C.2.1 Quantitative Data Summary

Task ID	Our Ideal Solution	Comparable Methods	time		pages		queries		correctness	
			Tool	No	Tool	No	Tool	No	Tool	No
1	squeeze	reshape	153.50	69.75	2.00	1.00	1.25	1.00	0.75	1.00
2	einsum	[matmul, matmul]	171.75	223.00	3.00	2.75	1.75	2.25	0.00	0.00
3	concat	stack	139.25	123.75	1.25	2.25	1.25	1.00	1.00	1.00
4	boolean_mask	[where, gather]	282.00	120.75	2.50	2.00	1.25	2.00	1.00	0.75
5	floordiv	[divide, floor]	224.50	106.00	2.00	2.50	1.50	1.25	1.00	1.00
6	resize_with_pad	resize	222.75	158.25	2.50	1.75	1.00	1.25	1.00	0.75
7	conv1d	conv2d/convolution	239.25	298.75	3.50	2.50	1.50	1.50	0.50	0.25
8	sparse_softmax_cross-_entropy_with_logits	softmax_cross-_entropy_with_logits	217.00	255.50	2.00	2.25	1.00	1.00	0.75	0.50
Avg			206.25	169.47	2.34	2.13	1.31	1.41	0.75	0.66
SD			95.53	105.40	1.47	1.31	0.69	0.87	0.44	0.48

Task ID	Our Ideal Solution	Comparable Methods	awareness		understanding	
			Tool	No	Tool	No
1	squeeze	reshape	0.75	0.25	2.00	2.00
2	einsum	[matmul, matmul]	1.00	0.00	1.75	0.00
3	concat	stack	1.00	0.75	2.00	1.00
4	boolean_mask	[where, gather]	0.75	0.00	0.75	0.50
5	floordiv	[divide, floor]	1.00	0.50	1.75	1.50
6	resize_with_pad	resize	1.00	0.75	2.00	1.75
7	conv1d	conv2d/convolution	0.75	0.00	1.75	0.50
8	sparse_softmax_cross-_entropy_with_logits	softmax_cross-_entropy_with_logits	0.75	0.75	1.50	1.50
Avg			0.88	0.38	1.69	1.09
SD			0.34	0.49	0.64	0.96

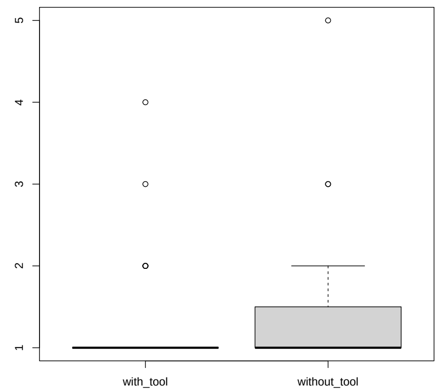
C.2.2 Statistical Test Results

Time



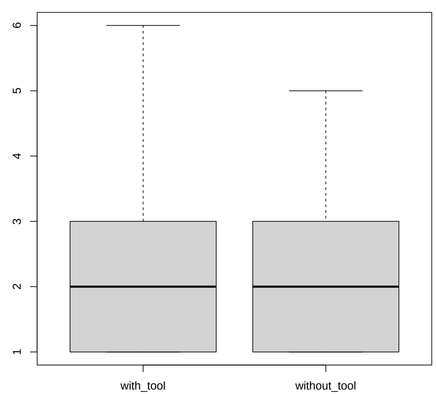
	Coeffs (Errors)
(Intercept)	198.69 (30.01)***
prior	-19.08 (14.35)
tool	37.97 (20.93)
AIC = 775.1; BIC = 788.1; LogLik = -381.5 $R2^m = 0.06$. $R2^c = 0.30$	

Queries



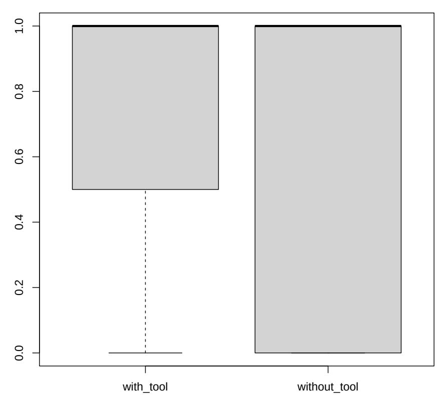
	Coeffs (Errors)
(Intercept)	1.41 (0.23)***
prior	-0.01 (0.12)
tool	-0.09 (0.18)
AIC = 160.1; BIC = 173.0; LogLik = -74.0 $R2^m = 0.00$. $R2^c = 0.14$	

Pages



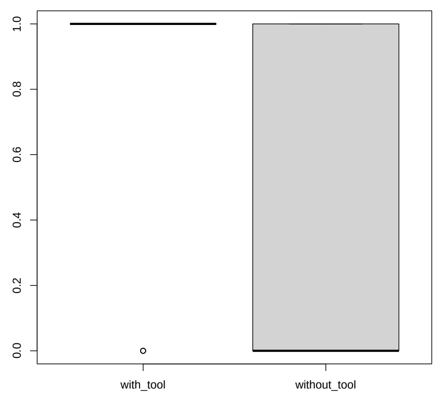
	Coeffs (Errors)
(Intercept)	2.36 (0.40)***
prior	-0.15 (0.20)
tool	0.23 (0.32)
AIC = 231.8; BIC = 244.8; LogLik = -109.9 $R2^m = 0.02$. $R2^c = 0.14$	

Correctness



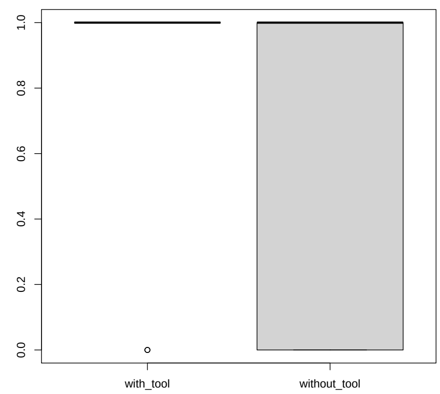
	Coeffs (Errors)
(Intercept)	-0.22 (1.50)
prior	0.96 (0.66)
tool	0.88 (0.87)
AIC = 66.2; BIC = 77.0; LogLik = -28.1 $R2^m = 0.07$. $R2^c = 0.74$	

Awareness



	Coeffs (Errors)
(Intercept)	-1.12 (0.92)
prior	0.30 (0.44)
tool	3.03 (0.95)**
AIC = 73.7; BIC = 84.5; LogLik = -31.9	
$R2^m = 0.35$. $R2^c = 0.53$	

Understanding



	Coeffs (Errors)
(Intercept)	-1.08 (1.10)
prior	1.11 (0.58).
tool	2.63 (0.95)**
AIC = 64.8; BIC = 75.6; LogLik = -27.4	
$R2^m = 0.31$. $R2^c = 0.63$	

Appendix D

Experiment Details for GILT Study

Table D.1: Completion time (s) of each subtask.

	All		Professionals		Students	
	GILT	Search	GILT	Search	GILT	Search
Bokeh-1	193.75	202.8	244.86	239.25	122.20	178.50
Bokeh-2	352.47	321.63	263.00	429.50	486.67	213.75
Bokeh-3	610.86	764.20	634.75	723.83	579.00	791.11
Bokeh-4	295.63	156.11	351.75	127.80	239.50	191.50
Open3d-1	483.47	503.80	458.38	663.00	505.78	265.00
Open3d-2	371.53	603.80	307.29	474.38	427.75	751.71
Open3d-3	408.83	326.67	454.67	267.67	363.00	444.67
Open3d-4	329.00	260.00	351.33	520.00	262.00	-

Table D.2: Completion rates of each subtask.

	All		Professionals		Students	
	GILT	Search	GILT	Search	GILT	Search
Bokeh-1	1.00	1.00	1.00	1.00	1.00	1.00
Bokeh-2	0.88	0.81	0.89	0.71	0.86	0.89
Bokeh-3	0.38	0.38	0.44	0.43	0.29	0.33
Bokeh-4	0.25	0.56	0.22	0.57	0.29	0.56
Open3d-1	0.81	0.69	1.00	0.67	0.67	0.71
Open3d-2	0.81	0.50	1.00	0.44	0.67	0.57
Open3d-3	0.50	0.13	0.57	0.11	0.44	0.14
Open3d-4	0.06	0.06	0.00	0.11	0.11	-

Table D.3: Understanding scores.

	All		**Professionals**		**Students**	
	GILT	Search	GILT	Search	GILT	Search
Bokeh	1.19	1.31	1.33	1.43	1.00	1.22
Open3d	0.75	0.125	1.43	0.11	0.22	1.14

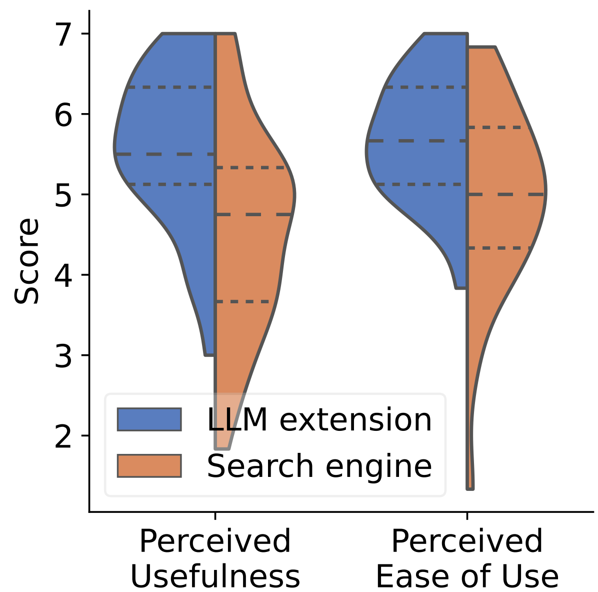


Figure D.1: Response to TAM items

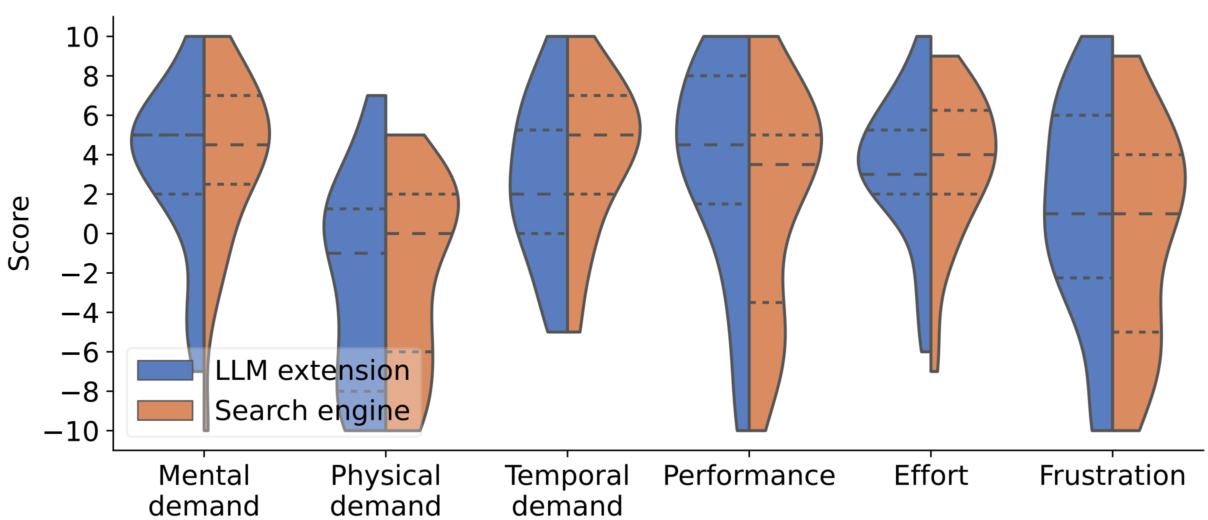


Figure D.2: Response to NASA TLX items

Table D.4: Participants' feedback codebook. The codes discussed in the paper are underlined. Initially, two authors separately performed open coding on the same set of 8 responses (25% of the entire data), and convened to discuss and merge the codes into a shared codebook. The first author coded the rest of the responses and discussed with the rest of the authors whenever new codes needed to be added.

	Code & Description	Representative Quote
Pros	**[Reduced initial apprehension effort]** The tool helps identify what to look for.	I was able to identify which sections of the code I need to modify much faster with the extension, even though I have no prior knowledge of 3D rendering. (P25)
	[Context-adjusted response] The tool provides code or explanation that is tailored to the user code.	Biggest difference using AI-based VS code extension is that the given solutions are specific to the code you are actually writing, whereas searching the web are usually more general solutions. (P32)
	[Amazement] General positive reaction for a new tool.	I was surprised how good the explanations were, how tailored it was to the code, how easily it could make the changes that were required, (P12)
	[Interactivity] Interactivity of the tool helps probing LLM.	The AI-based extension can be more interactive than search engines by maintaining conversation history. (P5)
	[Less context switching] The tool does not break the workflow and saves clicks as it is integrated in IDE.	··· I can accomplish everything within the IDE without having to context switch to other resources (P3)
	[Sufficient library details] The tool provides sufficient library information.	AI-based code extension was useful in a sense that it could also provide some details on concept, which could definitely help one to better tackle the task. (P28)
	[Context incorporation] The tool provides the ability to easily incorporate the context	(LLM-based AI programming tool is) better (than search engines). Since I don't need to come up with the good searching words (P20)
	[Varying granularity] The tool provides the ability to prompt about code at different granularity	Being able to change your level of insight was helpful (highlighting certain portions of code vs overall knowledge about the entire program (P15)
	[Less information foraging] The tool reduces the need for information foraging.	The main advantage for me is in circumventing the reading of the documentation. Official documentation is often lacking examples or short and concise explanations. For this reason, I often resort to stack overflow, in search of usage examples. (P24)

Table D.4: Participants' feedback codebook. The codes discussed in the paper are underlined. Initially, two authors separately performed open coding on the same set of 8 responses (25% of the entire data), and convened to discuss and merge the codes into a shared codebook. The first author coded the rest of the responses and discussed with the rest of the authors whenever new codes needed to be added (cont.).

	Code & Description	Representative Quote
Mixed	**[Required knowledge level for tool utility]** The tool is more useful when a user has / does not have existing knowledge.	It would be more useful when I learn a new library/PL. (P13) AI based are more helpful when you already know what the outcome should be (P11)
	[User interface] The user interface of the tool is usable / can be improved.	I liked the different buttons for different types of information so I didn't have to read a lot of text to find what I was looking for (P7) A simpler view would be nice, I felt like there were too many features which could get some time to get used to. (P8)
	[Pull-based interaction] Pull-based interaction is useful / not useful.	GitHub Copilot's autocomplete is faster as it automatically suggests what I'm supposed to type next. But I can't actually ask co-pilot follow up questions, which I can do in the ai-based vs code extension, and that makes it really helpful. (P1)
Cons	**[No correctness guarantee]** The lack of trust and correctness guarantee of the outputs.	... but there's always an underlying doubt in our mind regarding whether the information provided is 100% factual or useful. (P24)
	[Prompt dependency] The quality of the LLM-generated information depends on well-crafted prompts.	AI-based VS code extension was not able to give me the code that I was looking for, so it took up all my time (which I got very annoyed about). I think I just didn't word the question well. (P28)
	[Information diversity] The tool provides less diverse information compared to search engine results.	Stack overflow provides user historical comments and may be important for different approaches to the same problem. (P30)

Table D.4: Participants' feedback codebook. The codes discussed in the paper are underlined. Initially, two authors separately performed open coding on the same set of 8 responses (25% of the entire data), and convened to discuss and merge the codes into a shared codebook. The first author coded the rest of the responses and discussed with the rest of the authors whenever new codes needed to be added (cont.).

	Code & Description	Representative Quote
Sugge-stions	**[Other resources]** Add (links to) other relevant resources like documentation and Stack Overflow posts.	I hope it can be combined with a documentation of the library, which makes it more accurate when providing descriptions of a method. (P21)
	[LLM debugging capability] Ability to debug or auto-repair LLM outputs.	The ability to interactively debug incorrect AI-proposed solutions would be helpful. (P5)
	[**LLM performance**] Use more powerful and faster underlying LLM.	They were both quite verbose even when a single line would've sufficed. (P24)
	[Explainability] Add ability to provide the rationale behind the outputs.	I think it may be better if we could understand the why aspect of the code. (P10)

Bibliography

[1] Bokeh. `about:blank`. Retrieved: 2013-05-05. 7.3

[2] Chatgpt|openai. `https://chat.openai.com/`. Retrieved: 2013-05-05. (document), 5.3, 5.4.4, 5.4.4, 7.1

[3] Github codespaces. `https://github.com/features/codespaces`. Retrieved: 2013-05-05. 7.3

[4] Github copilot. `https://github.com/features/copilot`. Retrieved: 2013-05-05. 2.4, 7.1

[5] Open3d – a modern library for 3d data processing. `http://www.open3d.org/`. Retrieved: 2013-05-05. 7.3

[6] Tabnine: Ai assistant for software developers. `https://www.tabnine.com/`. Retrieved: 2013-05-05. 7.1

[7] Google privacy principles, 2023. URL `https://policies.google.com/privacy`. 3.2.3

[8] Google safety principles, 2023. URL `https://safety.google/principles/`. 3.2.3

[9] Google documentation style guide, 2023. URL `https://developers.google.com/style`. 3.2.2

[10] Rabe Abdalkareem, Emad Shihab, and Juergen Rilling. What do developers use the crowd for? a study using stack overflow. *IEEE Software*, 34(2):53–60, 2017. 2.1

[11] Mohamed Hussein Abdi, George Onyango Okeyo, and Ronald Waweru Mwangi. Matrix factorization techniques for context-aware collaborative filtering recommender systems: A survey. *Comput. Inf. Sci.*, 11(2):1–10, 2018. doi: 10.5539/cis.v11n2p1. URL `https://doi.org/10.5539/cis.v11n2p1`. 1.1.1, 3.5.3

[12] Deepak Agarwal, Bee-Chung Chen, Pradheep Elango, and Raghu Ramakrishnan. Content recommendation on web portals. *Commun. ACM*, 56(6):92–101, 2013. doi: 10.1145/2461256.2461277. URL `https://doi.org/10.1145/2461256.2461277`. 1.1.1, 3.5.3

[13] Emad Aghajani. Context-Aware Software Documentation. *2018 IEEE International Conference on Software Maintenance and Evolution* (*ICSME*), pages 727 – 731, 09 2018. doi: 10.1109/icsme.2018.00090. 8.1

[14] Emad Aghajani, Csaba Nagy, Olga Lucero Vega-Márquez, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, and Michele Lanza. Software documentation issues unveiled. In Joanne M. Atlee, Tevfik Bultan, and Jon Whittle, editors, *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 1199–1210. IEEE / ACM, 2019. doi: 10.1109/ICSE.2019.00122. URL `https://doi.org/10.1109/ICSE.2019.00122`. 2.1, 8.1

[15] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation. pages 2655–2668, 2021. doi: 10.18653/v1/2021.naacl-main.211. URL https://doi.org/10.18653/v1/2021.naacl-main.211. 6.2

[16] Miltiadis Allamanis and Charles Sutton. Why, when, and what: analyzing stack overflow questions by topic, type, and code. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 53–56. IEEE Press, 2013. ISBN 9781467329361. 2.1, 8.2.1

[17] Christoph Alt, Marc Hübner, and Leonhard Hennig. Fine-tuning pre-trained transformer language models to distantly supervised relation extraction. *arXiv preprint arXiv:1906.08646*, 2019. 2.3

[18] Matin Amoozadeh, David Daniels, Daye Nam, Stella Chen, Michael Hilton, Sruti Srinivasa Ragavan, and Mohammad Amin Alipour. Trust in generative ai among students: An exploratory study. In *The Technical Symposium on Computer Science Education*, 2024. 1, 2.4

[19] AmuletxHeart. How to avoid writing boilerplate code in java swing mvc? https://stackoverflow.com/questions/26154225/how-to-avoid-writing-boilerplate-code-in-java-swing-mvc, 2018. 4.2.2

[20] Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M. Dai, Anja Hauth, Katie Millican, David Silver, Slav Petrov, Melvin Johnson, Ioannis Antonoglou, Julian Schrittwieser, Amelia Glaese, Jilin Chen, Emily Pitler, Timothy P. Lillicrap, Angeliki Lazaridou, Orhan Firat, James Molloy, Michael Isard, Paul Ronald Barham, Tom Hennigan, Benjamin Lee, Fabio Viola, Malcolm Reynolds, Yuanzhong Xu, Ryan Doherty, Eli Collins, Clemens Meyer, Eliza Rutherford, Erica Moreira, Kareem Ayoub, Megha Goel, George Tucker, Enrique Piqueras, Maxim Krikun, Iain Barr, Nikolay Savinov, Ivo Danihelka, Becca Roelofs, Anaïs White, Anders Andreassen, Tamara von Glehn, Lakshman Yagati, Mehran Kazemi, Lucas Gonzalez, Misha Khalman, Jakub Sygnowski, and et al. Gemini: A family of highly capable multimodal models. *CoRR*, abs/2312.11805, 2023. doi: 10.48550/ARXIV.2312.11805. URL https://doi.org/10.48550/arXiv.2312.11805. 2.3

[21] Emily Judith Arteaga Garcia, João Felipe Nicolaci Pimentel, Zixuan Feng, Marco Gerosa, Igor Steinmacher, and Anita Sarma. How to support ml end-user programmers through a conversational agent. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400702174. doi: 10.1145/3597503.3608130. URL https://doi.org/10.1145/3597503.3608130. 9.2

[22] Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. Program synthesis with large language models. *CoRR*, abs/2108.07732, 2021. URL https://arxiv.org/abs/2108.07732. 9.2

[23] Shams Azad, Peter C Rigby, and Latifa Guerrouj. Generating api call rules from version history and stack overflow posts. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 25(4):1–22, 2017. 2.4

[24] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. 2017. URL https://openreview.net/

forum?id=ByldLrqlx. (document), 6.1, 6.1, 6.1, 6.1, 6.2, 6.5, 6.6.3

[25] Matej Balog, Rishabh Singh, Petros Maniatis, and Charles Sutton. Neural program synthesis with a differentiable fixer. *CoRR*, abs/2006.10924, 2020. URL https://arxiv.org/abs/2006.10924. 6.2

[26] Shraddha Barke, Michael B. James, and Nadia Polikarpova. Grounded copilot: How programmers interact with code-generating models. *Proc. ACM Program. Lang.*, 7(OOPSLA1), apr 2023. doi: 10.1145/3586030. URL https://doi.org/10.1145/3586030. 2.4

[27] Anton Barua, Stephen W Thomas, and Ahmed E Hassan. What are developers talking about? an analysis of topics and trends in stack overflow. *Empirical Software Engineering*, 19(3):619–654, 2014. 2.1

[28] Rohan Bavishi, Caroline Lemieux, Roy Fox, Koushik Sen, and Ion Stoica. Autopandas: neural-backed generators for program synthesis. *Proc. ACM Program. Lang.*, 3(OOPSLA): 168:1–168:27, 2019. doi: 10.1145/3360594. URL https://doi.org/10.1145/3360594. 6.1, 6.2, 6.8

[29] Laura Beckwith, Cory Kissinger, Margaret Burnett, Susan Wiedenbeck, Joseph Lawrance, Alan Blackwell, and Curtis Cook. Tinkering and gender in end-user programmers' debugging. In *Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 231–240, 2006. 7.5.3

[30] Stefanie Beyer and Martin Pinzger. A manual categorization of android app development issues on stack overflow. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 531–535, 2014. doi: 10.1109/ICSME.2014.88. 2.1

[31] Stefanie Beyer, Christian Macho, Martin Pinzger, and Massimiliano Di Penta. Automatically classifying posts into question categories on stack overflow. In *Proceedings of the 26th Conference on Program Comprehension*, ICPC '18, pages 211–221, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450357142. doi: 10.1145/3196321.3196333. URL https://doi.org/10.1145/3196321.3196333. 2.1, 8.2.1, 8.2.2

[32] Joshua Bloch. How to design a good API and why it matters. In *Companion to Conference on Object-oriented Programming Systems, Languages, and Applications*, pages 506–507. ACM, 2006. 4.1, 18, 4.1, 4.2, 4.2.1, 4.2.2

[33] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):P10008, 2008. 4.3.3

[34] Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In Dan R. Olsen Jr., Richard B. Arthur, Ken Hinckley, Meredith Ringel Morris, Scott E. Hudson, and Saul Greenberg, editors, *Proceedings of the 27th International Conference on Human Factors in Computing Systems, CHI 2009, Boston, MA, USA, April 4-9, 2009*, pages 1589–1598. ACM, 2009. doi: 10.1145/1518701.1518944. URL https://doi.org/10.1145/1518701.1518944. 3.1, 3.1, 3, 3.4.2, 3, 3.4.3

[35] Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R. Klemmer. Example-centric programming: integrating web search into the development environment. In *Proceedings of the 28th International Conference on Human Factors in Computing Systems,* (*CHI 2010*),

Atlanta, Georgia, USA, April 10-15, 2010, pages 513–522, New York, NY, 2010. ACM. doi: 10.1145/1753326.1753402. URL `https://doi.org/10.1145/1753326.1753402`. 7.7

[36] Broadcom. Spring framework. `https://spring.io`. 4.4.2

[37] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language Models are Few-Shot Learners. *arXiv*, 2020. doi: 10.48550/arxiv.2005.14165. 2.3

[38] Rudy Bunel, Matthew J. Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. Leveraging grammar and reinforcement learning for neural program synthesis. 2018. URL `https://openreview.net/forum?id=H1Xw62kRZ`. 6.2

[39] Margaret M. Burnett, Simone Stumpf, Jamie Macbeth, Stephann Makri, Laura Beckwith, Irwin Kwan, Anicia Peters, and Will Jernigan. Gendermag: A method for evaluating software's gender inclusiveness. *Interact. Comput.*, 28(6):760–787, 2016. doi: 10.1093/IWC/IWV046. URL `https://doi.org/10.1093/iwc/iwv046`. 7.5.3

[40] Open Source by greenrobot Logo Open Source by greenrobot Logo. Greendao. `http://greenrobot.org/greendao/documentation/introduction`, greenrobot . 4.4.2

[41] Marc Carpentier, Christophe Combescure, Laura Merlini, and Thomas V Perneger. Kappa statistic to measure agreement beyond chance in free-response assessments. *BMC medical research methodology*, 17(1):1–8, 2017. 5.2

[42] Preetha Chatterjee, Kostadin Damevski, and Lori Pollock. Automatic extraction of opinion-based q&a from online developer chats. In *International Conference on Software Engineering* (*ICSE*), pages 1260–1272. IEEE, 2021. 2.4

[43] Jie-Cherng Chen and Sun-Jen Huang. An empirical analysis of the impact of software development problem factors on software maintainability. *J. Syst. Softw.*, 82(6):981–992, 2009. doi: 10.1016/J.JSS.2008.12.036. URL `https://doi.org/10.1016/j.jss.2008.12.036`. 3.1

[44] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021. URL `https://arxiv.org/abs/2107.03374`. 1.1.2, 9.2

[45] Xinyun Chen, Chang Liu, and Dawn Song. Execution-guided neural program synthesis. 2019. URL https://openreview.net/forum?id=H1gfOiAqYm. 6.2

[46] Xinyun Chen, Dawn Song, and Yuandong Tian. Latent execution for neural program synthesis beyond domain-specific languages. pages 22196–22208, 2021. URL https://proceedings.neurips.cc/paper/2021/hash/ba3c95c2962d3aab2f6e667932daa3c5-Abstract.html. 6.2

[47] Robert F. Chew, John Bollenbacher, Michael Wenger, Jessica Speer, and Annice Kim. Llm-assisted content analysis: Using large language models to support deductive coding. *CoRR*, abs/2306.14924, 2023. doi: 10.48550/ARXIV.2306.14924. URL https://doi.org/10.48550/arXiv.2306.14924. 8.3.1

[48] Parmit K. Chilana, Amy J. Ko, and Jacob O. Wobbrock. Lemonaid: selection-based crowdsourced contextual help for web applications. pages 1549–1558, 2012. doi: 10.1145/2207676.2208620. URL https://doi.org/10.1145/2207676.2208620. 8.1

[49] Steven Clarke. What is an end user software engineer? In Margaret M. Burnett, Gregor Engels, Brad A. Myers, and Gregg Rothermel, editors, *End-User Software Engineering, 18.02. - 23.02.2007*, volume 07081 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2007. URL http://drops.dagstuhl.de/opus/volltexte/2007/1080. 2.2, 3.4.1, 2, 5.5

[50] Victoria Clarke and Virginia Braun. Teaching thematic analysis: Overcoming challenges and developing strategies for effective learning. *The psychologist*, 26(2):120–123, 2013. 7.6.2

[51] Wikipedia contributors. Boilerplate code definition of wikipedia. https://en.wikipedia.org/wiki/Boilerplate_code. 4.1, 4.2.2, 4.2.2, 4.2.2

[52] Carlos J Costa, Manuela Aparicio, and Robert Pierce. Evaluating information sources for computer programming learning and problem solving. In *Proceedings of the 9th WSEAS International Conference on APPLIED COMPUTER SCIENCE*, pages 218–223, 2009. 2.2, 3.1, 3.4.1

[53] Bill Curtis and Jakob Nielsen. Applying discount usability engineering. *IEEE Softw.*, 12 (1):98–100, 1995. doi: 10.1109/52.363161. URL https://doi.org/10.1109/52.363161. 3.1

[54] d370urn3ur. Reduce boilerplate for subclasses issue #172 parse-community/Parse-SDK-Android. https://github.com/parse-community/Parse-SDK-Android/issues/172. 4.1, 4.2

[55] Barthélémy Dagenais and Martin P Robillard. Creating and evolving developer documentation: understanding the decisions of open source contributors. *dl.acm.org*, page 127, 2010. doi: 10.1145/1882291.1882312. 8.1

[56] William K Darley and Robert E Smith. Gender differences in information processing strategies: An empirical test of the selectivity model in advertising response. *Journal of advertising*, 24(1):41–56, 1995. 7.5.3

[57] DatabaseGroup. Ap-ted implementation. https://github.com/DatabaseGroup/apted. 4.4.1

[58] Matthew C Davis, Emad Aghayi, Thomas D LaToza, Xiaoyin Wang, Brad A Myers, and Joshua Sunshine. What's (not) working in programmer user studies? *ACM Transactions on Software Engineering and Methodology*, 2022. 7.3

[59] Paul Denny, Viraj Kumar, and Nasser Giacaman. Conversing with copilot: Exploring prompt engineering for solving CS1 problems using natural language. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education, Volume 1, (SIGCSE 2023), Toronto, ON, Canada, March 15-18, 2023*, pages 1136–1142, New York, NY, USA, 2023. ACM. doi: 10.1145/3545945.3569823. URL `https://doi.org/10.1145/3545945.3569823`. 7.2.2, 7.4.1, 8.1

[60] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy I/O. 70: 990–998, 2017. URL `http://proceedings.mlr.press/v70/devlin17a.html`. 6.2

[61] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018. 5.4.1

[62] Ekwa Duala-Ekoko and Martin P Robillard. The information gathering strategies of api learners. Technical report, Technical report, TR-2010.6, School of Computer Science, McGill University, 2010. 7.2.2

[63] Ekwa Duala-Ekoko and Martin P Robillard. Using structure-based recommendations to facilitate discoverability in apis. In *European Conference on Object-oriented Programming*, pages 79–104. Springer, 2011. 5, 5.1

[64] Ekwa Duala-Ekoko and Martin P Robillard. Asking and answering questions about unfamiliar apis: An exploratory study. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 266–276. IEEE, 2012. 3.1, 3.1, 7.2.2

[65] Ashish Dwivedi, Steve Clarke, Brad A. Myers, Sae Young Jeong, Yingyu Xie, Jack Beaton, Jeff Stylos, Ralf Ehret, Jan Karstens, Arkin Efeoglu, and Daniela K. Busse. End-User Computing, Development, and Software Engineering. pages 81–102, 2012. doi: 10.4018/978-1-4666-0140-6.ch004. 8.1

[66] Ralph H. Earle, Mark A. Rosso, and Kathryn E. Alexander. User preferences of software documentation genres. In Kathie Gossett, Angie Mallory, and Dawn M. Armfield, editors, *Proceedings of the 33rd Annual International Conference on the Design of Communication, SIGDOC 2015, Limerick, Ireland, July 16-17, 2015*, pages 46:1–46:10. ACM, 2015. doi: 10.1145/2775441.2775457. URL `https://doi.org/10.1145/2775441.2775457`. 2.2, 3.1, 3.2.2, 3.4, 3.4.1, 1, 3.5.1

[67] Michael Ekstrand, Wei Li, Tovi Grossman, Justin Matejka, and George Fitzmaurice. Searching for software learning resources using application context. *Proceedings of the 24th annual ACM symposium on User interface software and technology - UIST '11*, page 195, 2011. doi: 10.1145/2047196.2047220. 8.1

[68] Kevin Ellis, Maxwell Nye, Yewen Pu, Felix Sosa, Joshua B. Tenenbaum, and Armando Solar-Lezama. Write, execute, assess: Program synthesis with a repl. 2019. 6.2

[69] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *International Conference on Automated Software Engineering*, pages 313–324. ACM, 2014. 4.3.3

[70] Umer Farooq and Dieter Zirkler. API peer reviews: A method for evaluating usability of application programming interfaces. In *Conference on Computer Supported Cooperative Work*, pages 207–210. ACM, 2010. 4.1

[71] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. Component-based synthesis of table consolidation and transformation tasks from examples. pages 422–436, 2017. doi: 10.1145/3062341.3062351. URL https://doi.org/10.1145/3062341.3062351. 6.2

[72] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. Program synthesis using conflict-driven learning. *ACM SIGPLAN Notices*, 53(4):420–435, 2018. 6.2

[73] James Finnie-Ansley, Paul Denny, Brett A. Becker, Andrew Luxton-Reilly, and James Prather. The robots are coming: Exploring the implications of openai codex on introductory programming. In *Australasian Computing Education Conference (ACE 2022) Virtual Event, Australia, February 14 - 18, 2022*, pages 10–19. ACM, 2022. doi: 10.1145/3511861.3511863. URL https://doi.org/10.1145/3511861.3511863. 2.4

[74] Felix Fischer, Konstantin Böttinger, Huang Xiao, Christian Stransky, Yasemin Acar, Michael Backes, and Sascha Fahl. Stack overflow considered harmful? the impact of copy&paste on android application security. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 121–136. IEEE, 2017. 5.5

[75] Scott D. Fleming, Chris Scaffidi, David Piorkowski, Margaret Burnett, Rachel Bellamy, Joseph Lawrance, and Irwin Kwan. An Information Foraging Theory Perspective on Tools for Debugging, Refactoring, and Reuse Tasks. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(2):14, 2013. ISSN 1049-331X. doi: 10.1145/2430545.2430551. 2.2

[76] Andrew Forward and Timothy Lethbridge. The relevance of software documentation, tools and technologies: a survey. In *Proceedings of the 2002 ACM Symposium on Document Engineering, McLean, Virginia, USA, November 8-9, 2002*, pages 26–33. ACM, 2002. doi: 10.1145/585058.585065. URL https://doi.org/10.1145/585058.585065. 3.1

[77] Jaroslav Fowkes and Charles Sutton. Parameter-free probabilistic API mining across GitHub. In *International Symposium on Foundations of Software Engineering*, pages 254–265. ACM, 2016. 4.1, 4.2.2, 4.3.1, 4.3.1, 4.4

[78] Steve Fox, Kuldeep Karnawat, Mark Mydland, Susan T. Dumais, and Thomas White. Evaluating implicit measures to improve web search. *ACM Trans. Inf. Syst.*, 23(2):147–168, 2005. doi: 10.1145/1059981.1059982. URL https://doi.org/10.1145/1059981.1059982. 3.2.2

[79] Luanne Freund. Contextualizing the information-seeking behavior of software engineers: Contextualizing the Information-Seeking Behavior of Software Engineers. *Journal of the Association for Information Science and Technology*, 66(8):1594–1605, 2014. ISSN 2330-1635. doi: 10.1002/asi.23278. 1.1.1, 3.1, 3.4.1, 1, 8.2.1, 8.2.2

[80] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. InCoder: A Generative Model for Code Infilling and Synthesis. *arXiv*, 2022. 8.1

[81] Golara Garousi, Vahid Garousi-Yusifoglu, Günther Ruhe, Junji Zhi, Mahmood Moussavi, and Brian Smith. Usage and usefulness of technical software documentation: An industrial case study. *Inf. Softw. Technol.*, 57:664–682, 2015. doi: 10.1016/J.INFSOF.2014.08.003. URL https://doi.org/10.1016/j.infsof.2014.08.003. 3.1

[82] M. Rami Ghorab, Dong Zhou, Alexander O'Connor, and Vincent Wade. Personalised Information Retrieval: survey and classification. *User Modeling and User-Adapted Interaction*, 23(4):381–443, 2013. ISSN 0924-1868. doi: 10.1007/s11257-012-9124-1. 8.1

[83] GitHub. Github copilot x: The ai-powered developer experience. `https://github.com/features/preview/copilot-x`, 2023. Retrieved: 2013-05-05. 7.8

[84] Elena L Glassman, Tianyi Zhang, Björn Hartmann, and Miryung Kim. Visualizing API usage examples at scale. In *Human Factors in Computing Systems*, pages 580:1–580:12. ACM, 2018. 4.1

[85] Victor M González and Gloria Mark. "Constant, constant, multi-tasking craziness": managing multiple working spheres. *Proceedings of the 2004 conference on Human factors in computing systems - CHI '04*, pages 113–120, 2004. doi: 10.1145/985692.985707. 8.4.2

[86] Google. Android API 26 release note. `https://developer.android.com/about/versions/oreo/android-8.0-changes#fvbi-signature`. 4.4.2

[87] Georgios Gousios and Diomidis Spinellis. GHTorrent: GitHub's data from a firehose. In *Internatioanl Conference on Mining Software Repositories*, pages 12–21. IEEE, 2012. 4.4.1

[88] Valentina Grigoreanu, Margaret M. Burnett, and George G. Robertson. A strategy-centric approach to the design of end-user debugging tools. In *Proceedings of the 28th International Conference on Human Factors in Computing Systems, (CHI 2010), Atlanta, Georgia, USA, April 10-15, 2010*, pages 713–722. ACM, 2010. doi: 10.1145/1753326.1753431. URL `https://doi.org/10.1145/1753326.1753431`. 7.5.3

[89] Tovi Grossman, Justin Matejka, and George W. Fitzmaurice. Chronicle: capture, exploration, and playback of document workflow histories. In Ken Perlin, Mary Czerwinski, and Rob Miller, editors, *Proceedings of the 23rd Annual ACM Symposium on User Interface Software and Technology, New York, NY, USA, October 3-6, 2010*, pages 143–152. ACM, 2010. doi: 10.1145/1866029.1866054. URL `https://doi.org/10.1145/1866029.1866054`. 2.1

[90] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. pages 317–330, 2011. doi: 10.1145/1926385.1926423. URL `https://doi.org/10.1145/1926385.1926423`. 6.2

[91] h5bp. html5-boilerplate. `https://github.com/h5bp/html5-boilerplate`. 4.2.1

[92] Md Montaser Hamid, Amreeta Chatterjee, Mariam Guizani, Andrew Anderson, Fatima Moussaoui, Sarah Yang, I Escobar, Anita Sarma, and Margaret Burnett. How to measure diversity actionably in technology. *Equity, Diversity, and Inclusion in Software Engineering: Best Practices and Insights*, 2023. 7.3

[93] Xiaoyu Han and Lei Wang. A novel document-level relation extraction method based on bert and entity information. *IEEE Access*, 8:96912–96919, 2020. 5.4.7

[94] Steven Hardy. Automatic induction of lisp functions. In *Proceedings of the 1st Summer Conference on Artificial Intelligence and Simulation of Behaviour*, AISB'74, pages 50–62, NLD, 1974. IOS Press. 6.2

[95] Sandra G Hart and Lowell E Staveland. Development of nasa-tlx (task load index): Results of empirical and theoretical research. In *Advances in psychology*, volume 52, pages 139–183. Elsevier, 1988. 7.3, 7.6.1

[96] Arto Hellas, Juho Leinonen, Sami Sarsa, Charles Koutcheme, Lilja Kujanpää, and Juha Sorva. Exploring the responses of large language models to beginner programmers' help requests. In *Proceedings of the 2023 ACM Conference on International Computing Education Research V.1* (*ICER 2023*). ACM, August 2023. doi: 10.1145/3568813.3600139. URL `http://dx.doi.org/10.1145/3568813.3600139`. 2.4

[97] Abram Hindle, Earl T. Barr, Mark Gabel, Zhendong Su, and Premkumar T. Devanbu. On the naturalness of software. *Commun. ACM*, 59(5):122–131, 2016. doi: 10.1145/2902362. URL `https://doi.org/10.1145/2902362`. 6.3

[98] Klaus Hinkelmann and Oscar Kempthorne. *Design and analysis of experiments, volume 1: Introduction to experimental design*, volume 1. John Wiley & Sons, 2007. 5.3.1

[99] Sture Holm. A simple sequentially rejective multiple test procedure. *Scandinavian Journal of Statistics*, 6(2):65–70, 1979. ISSN 03036898, 14679469. URL `http://www.jstor.org/stable/4615733`. (document), 3.4.4, 3.6, 3.7

[100] Amber Horvath, Sachin Grover, Sihan Dong, Emily Zhou, Finn Voichick, Mary Beth Kery, Shwetha Shinju, Daye Nam, Mariann Nagy, and Brad A. Myers. The long tail: Understanding the discoverability of API functionality. In Justin Smith, Christopher Bogart, Judith Good, and Scott D. Fleming, editors, *2019 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2019, Memphis, Tennessee, USA, October 14-18, 2019*, pages 157–161. IEEE Computer Society, 2019. doi: 10.1109/VLHCC.2019.8818681. URL `https://doi.org/10.1109/VLHCC.2019.8818681`. 1, 3.1, 3.1, 3.5.2, 5.1, 5.5

[101] Amber Horvath, Michael Xieyang Liu, River Hendriksen, Connor Shannon, Emma Paterson, Kazi Jawad, Andrew Macvean, and Brad A Myers. Understanding How Programmers Can Use Annotations on Documentation. *arXiv*, 2021. doi: 10.1145/3491102.3502095. 2.1

[102] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for nlp. In *International Conference on Machine Learning*, pages 2790–2799. PMLR, 2019. 5.4.1

[103] Jeremy Howard and Sebastian Ruder. Universal language model fine-tuning for text classification. *arXiv preprint arXiv:1801.06146*, 2018. 5.4.1

[104] Qiao Huang, Xin Xia, Zhenchang Xing, David Lo, and Xinyu Wang. Api method recommendation without worrying about the task-api knowledge gap. In *International Conference on Automated Software Engineering* (*ASE*), pages 293–304. IEEE, 2018. 2.4, 5.1

[105] Yintong Huo, Yuxin Su, Hongming Zhang, and Michael R. Lyu. ARCLIN: automated API mention resolution for unformatted texts. In *International Conference on Software Engineering* (*ICSE*), pages 138–149. IEEE, 2022. 5.4

[106] Saki Imai. Is github copilot a substitute for human pair-programming? an empirical study. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pages 319–321, 2022. 7.1

[107] Pankaj Jalote, Lionel Briand, André van der Hoek, Siddharth Subramanian, Laura Inozemtseva, and Reid Holmes. Live API documentation. *Proceedings of the 36th International Conference on Software Engineering*, pages 643–652, 2014. doi: 10.1145/2568225.2568313. 2.4, 5.1

[108] Sae Young Jeong, Yingyu Xie, Jack Beaton, Brad A. Myers, Jeffrey Stylos, Ralf Ehret, Jan Karstens, Arkin Efeoglu, and Daniela K. Busse. Improving documentation for esoa apis through user studies. In Volkmar Pipek, Mary Beth Rosson, Boris E. R. de Ruyter, and Volker Wulf, editors, *End-User Development, 2nd International Symposium, IS-EUD 2009, Siegen, Germany, March 2-4, 2009. Proceedings*, volume 5435 of *Lecture Notes in Computer Science*, pages 86–105. Springer, 2009. doi: 10.1007/978-3-642-00427-8_6. URL https://doi.org/10.1007/978-3-642-00427-8_6. 3.1, 3.1, 3.5.2, 3.5.3

[109] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. pages 215–224, 2010. doi: 10.1145/1806799.1806833. URL https://doi.org/10.1145/1806799.1806833. 6.2

[110] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why Don't Software Developers Use Static Analysis Tools to Find Bugs? *2013 35th International Conference on Software Engineering (ICSE)*, pages 672–681, 2013. doi: 10.1109/icse.2013.6606613. 4.4

[111] Ron Johnston, Kelvyn Jones, and David Manley. Confounding and collinearity in regression analysis: a cautionary tale and an alternative procedure, illustrated by studies of british voting behaviour. *Quality & Quantity*, 52:1–20, 07 2018. doi: 10.1007/s11135-017-0584-6. 3.4.4, 7.7

[112] Cory J. Kapser and Michael W. Godfrey. "cloning considered harmful" considered harmful: patterns of cloning in software. *Empirical Software Engineering*, 13(6):645, 2008. 4.2, 4.2.1, 4.2.2, 4.2.2

[113] Majeed Kazemitabaar, Justin Chow, Carl Ka To Ma, Barbara J Ericson, David Weintrop, and Tovi Grossman. Studying the effect of AI Code Generators on Supporting Novice Learners in Introductory Programming. *arXiv*, 2023. doi: 10.1145/3544548.3580919. 2.4

[114] Mik Kersten and Gail C. Murphy. Using task context to improve programmer productivity. *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering - SIGSOFT '06/FSE-14*, pages 1–11, 2006. doi: 10.1145/1181775.1181777. 8.4.2

[115] Jinhan Kim, Sanghoon Lee, Seung-Won Hwang, and Sunghun Kim. Enriching Documents with Examples: A Corpus Mining Approach. *ACM Transactions on Information Systems (TOIS)*, 31(1):1, 2013. ISSN 1046-8188. doi: 10.1145/2414782.2414783. 1.1.2

[116] Amy J. Ko and Yann Riche. The role of conceptual knowledge in API usability. In Gennaro Costagliola, Amy J. Ko, Allen Cypher, Jeffrey Nichols, Christopher Scaffidi, Caitlin Kelleher, and Brad A. Myers, editors, *2011 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2011, Pittsburgh, PA, USA, September 18-22, 2011*, pages 173–176. IEEE, 2011. doi: 10.1109/VLHCC.2011.6070395. URL https://doi.org/10.1109/VLHCC.2011.6070395. 1.1.1, 2.2, 3.4.1, 7.2.2, 7.6.2, 8.1

[117] Amy J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Trans. Software Eng.*, 32(12):971–987, 2006. doi: 10.1109/TSE.2006.116. URL https://doi.org/10.1109/TSE.2006.116. 1, 3.1, 3.1, 3.4, 3, 3.5.1, 7.1, 8.1

[118] Amy J. Ko, Robert DeLine, and Gina Venolia. Information needs in collocated software development teams. In *29th International Conference on Software Engineering (ICSE*

2007), Minneapolis, MN, USA, May 20-26, 2007, pages 344–353. IEEE Computer Society, 2007. doi: 10.1109/ICSE.2007.45. URL `https://doi.org/10.1109/ICSE.2007.45`. 1, 2.1, 3.1, 3.5.2, 7.1

[119] A Gunes Koru and Hongfang Liu. Building effective defect-prediction models in practice. *Software*, 22(6):23–29, 2005. 4.4

[120] Sandeep Kaur Kuttal, Se Yeon Kim, Carlos Martos, and Alexandra Bejarano. How end-user programmers forage in online repositories? An information foraging perspective. *Journal of Computer Languages*, 62:101010, 2021. ISSN 2590-1184. doi: 10.1016/j.cola.2020.101010. 2.2

[121] Dmitry Lagun and Mounia Lalmas. Understanding user attention and engagement in online news reading. In Paul N. Bennett, Vanja Josifovski, Jennifer Neville, and Filip Radlinski, editors, *Proceedings of the Ninth ACM International Conference on Web Search and Data Mining, San Francisco, CA, USA, February 22-25, 2016*, pages 113–122. ACM, 2016. doi: 10.1145/2835776.2835833. URL `https://doi.org/10.1145/2835776.2835833`. 3.1

[122] Ralf Lämmel and Simon Peyton Jones. *Scrap your boilerplate: a practical design pattern for generic programming*. ACM, 2003. 4.2

[123] J Richard Landis and Gary G Koch. The measurement of observer agreement for categorical data. *biometrics*, pages 159–174, 1977. 5.2

[124] Thomas D. LaToza and Brad A. Myers. Hard-to-answer questions about code. *Evaluation and Usability of Programming Languages and Tools on - PLATEAU '10*, page 8, 2010. doi: 10.1145/1937117.1937125. 1, 2.1

[125] Thomas D. LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: a study of developer work habits. In Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa, editors, *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, pages 492–501. ACM, 2006. doi: 10.1145/1134285.1134355. URL `https://doi.org/10.1145/1134285.1134355`. 1, 2.2, 3.4.1, 7.1

[126] Thomas D LaToza, David Garlan, James D Herbsleb, and Brad A Myers. Program comprehension as fact finding. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 361–370, 2007. 7.2.2, 8.2.1

[127] Joseph Lawrance, Rachel Bellamy, Margaret Burnett, and Kyle Rector. Using information scent to model the dynamic foraging behavior of programmers in maintenance tasks. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '08, pages 1323–1332, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605580111. doi: 10.1145/1357054.1357261. URL `https://doi.org/10.1145/1357054.1357261`. 2.2

[128] Joseph Lawrance, Christopher Bogart, Margaret M. Burnett, Rachel K. E. Bellamy, Kyle Rector, and Scott D. Fleming. How programmers debug, revisited: An information foraging theory perspective. *IEEE Trans. Software Eng.*, 39(2):197–215, 2013. doi: 10.1109/TSE.2010.111. URL `https://doi.org/10.1109/TSE.2010.111`. 2.2, 3.1

[129] Vu Le and Sumit Gulwani. Flashextract: A framework for data extraction by examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 542–553, 2014. 6.2

[130] Younghwa Lee, Kenneth A Kozar, and Kai RT Larsen. The technology acceptance model: Past, present, and future. *Communications of the Association for information systems*, 12 (1):50, 2003. 7.3, 7.6.1

[131] Juho Leinonen, Paul Denny, Stephen MacNeil, Sami Sarsa, Seth Bernstein, Joanne Kim, Andrew Tran, and Arto Hellas. Comparing Code Explanations Created by Students and Large Language Models. *arXiv*, 2023. 2.4

[132] Timothy C Lethbridge, Janice Singer, and Andrew Forward. How software engineers use documentation: The state of the practice. *IEEE software*, 20(6):35–39, 2003. 5.1

[133] Hongwei Li, Zhenchang Xing, Xin Peng, and Wenyun Zhao. What Help Do Developers Seek, When and How? *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 142–151, 2013. doi: 10.1109/wcre.2013.6671289. 2.2, 3.4.1

[134] Hongwei Li, Xuejiao Zhao, Zhenchang Xing, Lingfeng Bao, Xin Peng, Dongjing Gao, and Wenyun Zhao. amassist: In-ide ambient search of online programming resources. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 390–398. IEEE, 2015. 7.7

[135] Hongwei Li, Sirui Li, Jiamou Sun, Zhenchang Xing, Xin Peng, Mingwei Liu, and Xuejiao Zhao. Improving api caveats accessibility by mining api caveats knowledge graph. In *International Conference on Software Maintenance and Evolution (ICSME)*, pages 183–193. IEEE, 2018. 1.1.2, 2.4

[136] Jing Li, Aixin Sun, and Zhenchang Xing. To Do or Not To Do: Distill crowdsourced negative caveats to augment api documentation. *Journal of the Association for Information Science and Technology*, 69(12):1460–1475, 2018. ISSN 2330-1635. doi: 10.1002/asi.24067. 2.4

[137] Paul Luo Li, Amy J Ko, and Jiamin Zhu. What makes a great software engineer? In *International Conference on Software Engineering (ICSE)*, volume 1, pages 700–710. IEEE, 2015. 5.1

[138] Jenny T Liang, Chenyang Yang, and Brad A Myers. Understanding the usability of ai programming assistants. *arXiv preprint arXiv:2303.17125*, 2023. 2.4, 7.1, 8.1

[139] H Lieberman and T Selker. Out of context: Computer systems that adapt to, and learn from, context. *IBM Systems Journal*, 39(3.4):617–632, 2000. ISSN 0018-8670. doi: 10.1147/sj.393.0617. 8.1

[140] Bin Lin, Fiorella Zampetti, Gabriele Bavota, Massimiliano Di Penta, and Michele Lanza. Pattern-based mining of opinions in q&a websites. In *International Conference on Software Engineering (ICSE)*, pages 548–559. IEEE, 2019. 2.4

[141] Bin Lin, Nathan Cassee, Alexander Serebrenik, Gabriele Bavota, Nicole Novielli, and Michele Lanza. Opinion mining for software development: A systematic literature review. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(3):1–41, 2022. 2.4

[142] Mario Linares-Vásquez, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. How do api changes trigger stack overflow discussions? a study on the android sdk. In *International Conference on Program Comprehension (ICPC)*, pages 83–94, 2014. 2.1, 5.1

[143] Bing Liu. *Web Data Mining: Exploring Hyperlinks, Contents, and Usage Data. Second Edition*. Data-Centric Systems and Applications. Springer, 2011. ISBN 978-3-642-19459-7. doi: 10.1007/978-3-642-19460-3. URL `https://doi.org/10.1007/978-3-642-19460-3`. 3.2.2

[144] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation, 2023. Retrieved from `https://arxiv.org/abs/2305.01210`. 2.4

[145] Michael Xieyang Liu, Jane Hsieh, Nathan Hahn, Angelina Zhou, Emily Deng, Shaun Burley, Cynthia Taylor, Aniket Kittur, and Brad A. Myers. Unakite: Scaffolding developers' decision-making using the web. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*, UIST '19, pages 67–80, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368162. doi: 10.1145/3332165.3347908. URL `https://doi.org/10.1145/3332165.3347908`. 1

[146] Mingwei Liu, Xin Peng, Andrian Marcus, Shuangshuang Xing, Christoph Treude, and Chengyuan Zhao. Api-related developer information needs in stack overflow. *IEEE Transactions on Software Engineering*, 2021. 2.1, 5.1, 5.1, 5.4.6, 8.2.1

[147] Mingwei Liu, Xin Peng, Andrian Marcus, Christoph Treude, Jiazhan Xie, Huanjun Xu, and Yanjun Yang. How to Formulate Specific How-To Questions in Software Development? 2022. 8.1

[148] Yang Liu, Mingwei Liu, Xin Peng, Christoph Treude, Zhenchang Xing, and Xiaoxin Zhang. Generating concept based api element comparison using a knowledge graph. In *International Conference on Automated Software Engineering* (*ASE*), pages 834–845. IEEE, 2020. (document), 2.3, 5.1, 5.3, 5.4.4, 5.4.4

[149] Project Lombok. Project lombok. `https://projectlombok.org`. 4.2.4

[150] W. Maalej and M. P. Robillard. Patterns of Knowledge in API Reference Documentation. *IEEE Transactions on Software Engineering*, 39(9):1264–1282, 2013. ISSN 0098-5589. doi: 10.1109/tse.2013.12. 3.1

[151] Walid Maalej, Rebecca Tiarks, Tobias Roehm, and Rainer Koschke. On the comprehension of program comprehension. *ACM Trans. Softw. Eng. Methodol.*, 23(4):31:1–31:37, 2014. doi: 10.1145/2622669. URL `https://doi.org/10.1145/2622669`. 1, 2.2, 3.4.1, 7.1, 7.8, 8.2.1

[152] Stephen MacNeil, Andrew Tran, Arto Hellas, Joanne Kim, Sami Sarsa, Paul Denny, Seth Bernstein, and Juho Leinonen. Experiences from Using Code Explanations Generated by Large Language Models in a Web Software Development E-Book. *arXiv*, 2022. 2.4

[153] Andrew Macvean, Martin Maly, and John Daughtry. API design reviews at scale. In *Extended Abstracts on Human Factors in Computing Systems*, pages 849–858. ACM, 2016. 4.1

[154] Zohar Manna and Richard Waldinger. A deductive approach to program synthesis. pages 542–551, 1979. 6.2

[155] mast group. Probabilistic API mining implementation. `https://github.com/mast-group/API-mining`. 4.4.1

[156] Rob McCarney, James Warner, Steve Iliffe, Robbert Haselen, Mark Griffin, and Peter Fisher. The hawthorne effect: A randomised, controlled trial. *BMC medical research methodology*, 7:30, 02 2007. doi: 10.1186/1471-2288-7-30. 3.1

[157] Sahar Mehrpour, Thomas D. LaToza, and Rahul K. Kindi. Active Documentation: Helping Developers Follow Design Decisions. *2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 00:87–96, 2019. doi: 10.1109/vlhcc.2019.8818816. 3.1

[158] Michael Meng, Stephanie Steinhardt, and Andreas Schubert. Application programming interface documentation: What do software developers want? *Journal of Technical Writing and Communication*, 48(3):295–330, 2018. doi: 10.1177/0047281617721853. URL `https://doi.org/10.1177/0047281617721853`. 2.1, 3.1, 3.1, 3.4, 3.5.1, 3.5.2, 3.5.3, 5.1, 7.2.2, 7.5.3

[159] Michael Meng, Stephanie Steinhardt, and Andreas Schubert. How developers use API documentation: an observation study. *dl.acm.org*, 2018. doi: 10.1145/3274995.3274999. 2.2, 3.1, 3.1, 3.4, 3.4.1, 2, 3.5.1, 5.5, 7.5.3

[160] Michael Meng, Stephanie M. Steinhardt, and Andreas Schubert. Optimizing API documentation: Some guidelines and effects. In Josephine N. Walwema, Daniel L. Hocutt, and Stacey Pigg, editors, *SIGDOC '20: The 38th ACM International Conference on Design of Communication, Denton, TX, USA, October 5-9, 2020*, pages 24:1–24:11. ACM, 2020. doi: 10.1145/3380851.3416759. URL `https://doi.org/10.1145/3380851.3416759`. 3.1, 3.1

[161] Andre N. Meyer, Laura E. Barton, Gail C. Murphy, Thomas Zimmermann, and Thomas Fritz. The Work Life of Developers: Activities, Switches and Perceived Productivity. *IEEE Transactions on Software Engineering*, 43(12):1178–1193, 2017. ISSN 0098-5589. doi: 10.1109/tse.2017.2656886. 1, 7.1

[162] Tomás Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In Christopher J. C. Burges, Léon Bottou, Zoubin Ghahramani, and Kilian Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States*, pages 3111–3119, 2013. URL `https://proceedings.neurips.cc/paper/2013/hash/9aa42b31882ec039965f3c4923ce901b-Abstract.html`. 5.4.4

[163] Michela Montesi and Trilce Navarrete. Classifying web genres in context: A case study documenting the web genres used by a software engineer. *Inf. Process. Manag.*, 44 (4):1410–1430, 2008. doi: 10.1016/J.IPM.2008.02.001. URL `https://doi.org/10.1016/j.ipm.2008.02.001`. 1

[164] Todd K Moon. The expectation-maximization algorithm. *Signal Processing Magazine*, 13 (6):47–60, 1996. 4.3.1

[165] Eduardo Mosqueira-Rey, David Alonso-Ríos, Vicente Moret-Bonillo, Isaac Fernández-Varela, and Diego Álvarez-Estévez. A systematic approach to API usability: Taxonomy-derived criteria and a case study. *Information and Software Technology*, 97:46–63, 2018. 4.1, 4.1, 4.2

[166] Hussein Mozannar, Gagan Bansal, Adam Fourney, and Eric Horvitz. Reading Between the Lines: Modeling User Behavior and Costs in AI-Assisted Programming. *arXiv*, 2022. doi: 10.48550/arxiv.2210.14306. 2.4

[167] Lauren Murphy, Mary Beth Kery, Oluwatosin Alliyu, Andrew Macvean, and Brad A Myers. 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)*, pages 249–258. IEEE, 2018. 4.1

[168] Brad A Myers and Jeffrey Stylos. Improving API usability. *Communications of the ACM*, 59(6):62–69, 2016. 4.1, 4.1

[169] Varvana Myllärniemi, Sari Kujala, Mikko Raatikainen, and Piia Sevón. Development as a journey: factors supporting the adoption and use of software frameworks. *J. Softw. Eng. Res. Dev.*, 6:6, 2018. doi: 10.1186/S40411-018-0050-8. URL `https://doi.org/10.1186/s40411-018-0050-8`. 3.1

[170] Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. Jumping through hoops: Why do java developers struggle with cryptography apis? In *International Conference on Software Engineering (ICSE)*, pages 935–946, 2016. 2.1, 5.1

[171] N. J. D. NAGELKERKE. A note on a general definition of the coefficient of determination. *Biometrika*, 78(3):691–692, 09 1991. ISSN 0006-3444. doi: 10.1093/biomet/78.3.691. URL `https://doi.org/10.1093/biomet/78.3.691`. 3.4.5

[172] Daye Nam. API design implications of boilerplate client code. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*, pages 1253–1255. IEEE, 2019. doi: 10.1109/ASE.2019.00153. URL `https://doi.org/10.1109/ASE.2019.00153`. 1, 2, 3

[173] Daye Nam, Amber Horvath, Andrew Macvean, Brad Myers, and Bogdan Vasilescu. Marble source code and the result. `https://doi.org/10.5281/zenodo.3408715`, 2019. 4.1, 4.4.2

[174] Daye Nam, Amber Horvath, Andrew Macvean, Brad A. Myers, and Bogdan Vasilescu. MARBLE: mining for boilerplate code to identify API usability problems. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*, pages 615–627. IEEE, 2019. doi: 10.1109/ASE.2019.00063. URL `https://doi.org/10.1109/ASE.2019.00063`. 1, 2, 3, 3.4.5, 6, 5.1, 6.8

[175] Daye Nam, Baishakhi Ray, Seohyun Kim, Xianshan Qu, and Satish Chandra. Predictive synthesis of api-centric code. In Swarat Chaudhuri and Charles Sutton, editors, *MAPS@PLDI 2022: 6th ACM SIGPLAN International Symposium on Machine Programming, San Diego, CA, USA, 13 June 2022*, pages 40–49. ACM, 2022. doi: 10.1145/3520312.3534866. URL `https://doi.org/10.1145/3520312.3534866`. 2, 16

[176] Daye Nam, Andrew Macvean, Vincent J. Hellendoorn, Bogdan Vasilescu, and Brad A. Myers. In-ide generation-based information support with a large language model. *CoRR*, abs/2307.08177, 2023. doi: 10.48550/ARXIV.2307.08177. URL `https://doi.org/10.48550/arXiv.2307.08177`. 4, 20, 8.1

[177] Daye Nam, Brad A. Myers, Bogdan Vasilescu, and Vincent J. Hellendoorn. Improving API knowledge discovery with ML: A case study of comparable API methods. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pages 1890–1906. IEEE, 2023. doi: 10.1109/ICSE48619.2023.00161. URL `https://doi.org/10.1109/ICSE48619.2023.00161`. 2, 3, 4, 3.5.2, 8

[178] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. Artifacts for Using an LLM to Help With Code Understanding, January 2024. URL `https://doi.org/10.5281/zenodo.10461385`. 7.3

[179] Daye Nam, Andrew Macvean, Brad Myers, and Bogdan Vasilescu. Understanding documentation use through log analysis: A case study of four cloud services. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*, 2024. 1, 3, 1

[180] Anh Tuan Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. Statistical learning approach for mining API usage mappings for code migration. In Ivica Crnkovic, Marsha Chechik, and Paul Grünbacher, editors, *International Conference on Automated Software Engineering* (*ASE*), pages 457–468. ACM, 2014. 5.1

[181] Christina Niklaus, Matthias Cetto, André Freitas, and Siegfried Handschuh. A survey on open information extraction. In *International Conference on Computational Linguistics* (*COLING*), pages 3866–3878, 2018. 5.1

[182] Nan Niu, Anas Mahmoud, Zhangji Chen, and Gary Bradshaw. Departures from optimality: understanding human analyst's information foraging in assisted requirements tracing. In David Notkin, Betty H. C. Cheng, and Klaus Pohl, editors, *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 572–581. IEEE Computer Society, 2013. doi: 10.1109/ICSE.2013.6606603. URL `https://doi.org/10.1109/ICSE.2013.6606603`. 2.2

[183] Maxwell I. Nye, Luke B. Hewitt, Joshua B. Tenenbaum, and Armando Solar-Lezama. Learning to infer program sketches. 97:4861–4870, 2019. URL `http://proceedings.mlr.press/v97/nye19a.html`. 6.2

[184] Maxwell I. Nye, Yewen Pu, Matthew Bowers, Jacob Andreas, Joshua B. Tenenbaum, and Armando Solar-Lezama. Representing partial programs with blended abstract semantics. 2021. URL `https://openreview.net/forum?id=mCtadqIxOJ`. 6.2

[185] Janet Nykaza, Rhonda Messinger, Fran Boehme, Cherie L. Norman, Matthew Mace, and Manuel Gordon. What programmers really want: results of a needs assessment for SDK documentation. *Proceedings of the 20th annual international conference on Computer documentation - SIGDOC '02*, pages 133–141, 2002. doi: 10.1145/584955.584976. 2.1, 3.1

[186] Augustus Odena, Kensen Shi, David Bieber, Rishabh Singh, Charles Sutton, and Hanjun Dai. BUSTLE: bottom-up program synthesis through learning-guided exploration. 2021. URL `https://openreview.net/forum?id=yHeg4PbFHh`. 6.2

[187] Stephen Oney and Joel Brandt. Codelets: linking interactive documentation and example code in the editor. In Joseph A. Konstan, Ed H. Chi, and Kristina Höök, editors, *CHI Conference on Human Factors in Computing Systems, CHI '12, Austin, TX, USA - May 05 - 10, 2012*, pages 2697–2706. ACM, 2012. doi: 10.1145/2207676.2208664. URL `https://doi.org/10.1145/2207676.2208664`. 2.1

[188] Amantia Pano, Daniel Graziotin, and Pekka Abrahamsson. Factors and actors leading to the adoption of a javascript framework. *Empir. Softw. Eng.*, 23(6):3503–3534, 2018. doi: 10.1007/S10664-018-9613-X. URL `https://doi.org/10.1007/s10664-018-9613-x`. 3.1

[189] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis. 2017. URL `https://openreview.net/forum?id=rJ0JwFcex`. 6.2

[190] Chris Parnin, Christoph Treude, Lars Grammel, and Margaret-Anne Storey. Crowd documentation: Exploring the coverage and the dynamics of api discussions on stack overflow. Technical Report GIT-CS-12-05, Georgia Institute of Technology, 2012. 2.4, 5.1

[191] Nate Parsons. Boilerplate code definition of stackoverflow. `https://stackoverflow.com/questions/3992199/what-is-boilerplate-code`. 4.1, 4.2.2, 4.2.2

[192] Mateusz Pawlik and Nikolaus Augsten. Tree edit distance: Robust and memory-efficient. *Information Systems*, 56:157 – 173, 2016. 4.3.3

[193] David N Perkins, Chris Hancock, Renee Hobbs, Fay Martin, and Rebecca Simmons. Conditions of learning in novice programmers. *Journal of Educational Computing Research*, 2 (1):37–55, 1986. 7.5.3

[194] Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. Do users write more insecure code with AI assistants? *CoRR*, abs/2211.03622, 2022. doi: 10.48550/arXiv.2211.03622. URL `https://doi.org/10.48550/arXiv.2211.03622`. 2.4

[195] Dimitrios Pierrakos, Georgios Paliouras, Christos Papatheodorou, and Constantine D. Spyropoulos. Web usage mining as a tool for personalization: A survey. *User Model. User Adapt. Interact.*, 13(4):311–372, 2003. doi: 10.1023/A:1026238916441. URL `https://doi.org/10.1023/A:1026238916441`. 3.5.1

[196] David Piorkowski, Austin Z. Henley, Tahmid Nabi, Scott D. Fleming, Christopher Scaffidi, and Margaret Burnett. Foraging and navigations, fundamentally: developers' predictions of value and cost. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 97–108, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342186. doi: 10.1145/2950290.2950302. URL `https://doi.org/10.1145/2950290.2950302`. 1, 7.1

[197] David J Piorkowski, Scott D Fleming, Irwin Kwan, Margaret M Burnett, Christopher Scaffidi, Rachel K E Bellamy, and Joshua Jordahl. The whats and hows of programmers' foraging diets. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 3063–3072, 2013. doi: 10.1145/2470654.2466418. 8.1

[198] Peter Pirolli and Stuart Card. Information foraging. *Psychological review*, 106(4):643, 1999. 2.2

[199] Peter L. T. Pirolli. *Information Foraging Theory: Adaptive Interaction with Information*. Oxford University Press, 05 2007. ISBN 9780195173321. doi: 10.1093/acprof:oso/9780195173321.001.0001. URL `https://doi.org/10.1093/acprof:oso/9780195173321.001.0001`. 3.5.2

[200] Reinhold Plösch, Andreas Dautovic, and Matthias Saft. The value of software documentation quality. In *2014 14th International Conference on Quality Software, Allen, TX, USA, October 2-3, 2014*, pages 333–342. IEEE, 2014. doi: 10.1109/QSIC.2014.22. URL `https://doi.org/10.1109/QSIC.2014.22`. 3.1

[201] Luca Ponzanelli, Alberto Bacchelli, and Michele Lanza. Seahawk: Stack overflow in the ide. In *International Conference on Software Engineering* (*ICSE*), pages 1295–1298. IEEE, 2013. 2.4

[202] Luca Ponzanelli, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Michele Lanza. Mining stack overflow to turn the ide into a self-confident programming prompter. In *International Conference on Mining Software Repositories* (*MSR*), pages 102–111, 2014.

2.4

[203] Christi-Anne Postava-Davignon, Candice Kamachi, Cory Clarke, Gregory Kushmerek, Mary Beth Rettger, Pete Monchamp, and Rich Ellis. Incorporating usability testing into the documentation process. *Technical Communication*, 51:36–44, 02 2004. 2.2, 3.4.1

[204] Sruti Srinivasa Ragavan, Mihai Codoban, David Piorkowski, Danny Dig, and Margaret Burnett. Version control systems: An information foraging perspective. *IEEE Trans. Software Eng.*, 47(8):1644–1655, 2021. doi: 10.1109/TSE.2019.2931296. URL `https://doi.org/10.1109/TSE.2019.2931296`. 2.2

[205] Nikitha Rao, Chetan Bansal, Thomas Zimmermann, Ahmed Hassan Awadallah, and Nachiappan Nagappan. Analyzing web search behavior for software engineering tasks. In Xintao Wu, Chris Jermaine, Li Xiong, Xiaohua Hu, Olivera Kotevska, Siyuan Lu, Weija Xu, Srinivas Aluru, Chengxiang Zhai, Eyhab Al-Masri, Zhiyuan Chen, and Jeff Saltz, editors, *2020 IEEE International Conference on Big Data (IEEE BigData 2020), Atlanta, GA, USA, December 10-13, 2020*, pages 768–777. IEEE, 2020. doi: 10.1109/BIGDATA50022.2020.9378083. URL `https://doi.org/10.1109/BigData50022.2020.9378083`. 2.1, 3.1, 3, 8.1

[206] Irum Rauf, Pekka Perälä, Jouni Huotari, and Ivan Porres. Perceived Obstacles by Novice Developers Adopting User Interface APIs and Tools. *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 223–227, 2016. doi: 10.1109/vlhcc.2016.7739689. 4, 3.5.2

[207] Martin Reddy. *API Design for C++*. Elsevier, 2011. 4.1, 4.1, 4.2, 4.2.1, 4.2.2, 4.2.2

[208] Xiaoxue Ren, Zhenchang Xing, Xin Xia, Guoqiang Li, and Jianling Sun. Discovering, explaining and summarizing controversial discussions in community q&a sites. In *International Conference on Automated Software Engineering (ASE)*, pages 151–162. IEEE, 2019. 2.4

[209] Xiaoxue Ren, Jiamou Sun, Zhenchang Xing, Xin Xia, and Jianling Sun. Demystify official api usage directives with crowdsourced api misuse scenarios, erroneous code examples, and patches. In *International Conference on Software Engineering (ICSE)*, pages 925–936, 2020. 2.4, 5.1

[210] Martin P. Robillard. What makes apis hard to learn? answers from developers. *IEEE Softw.*, 26(6):27–34, 2009. doi: 10.1109/MS.2009.193. URL `https://doi.org/10.1109/MS.2009.193`. 2.1, 3.1

[211] Martin P. Robillard and Yam B. Chhetri. Recommending reference API documentation. *Empir. Softw. Eng.*, 20(6):1558–1586, 2015. doi: 10.1007/s10664-014-9323-y. URL `https://doi.org/10.1007/s10664-014-9323-y`. 2.3

[212] Martin P. Robillard and Robert DeLine. A field study of API learning obstacles. *Empir. Softw. Eng.*, 16(6):703–732, 2011. doi: 10.1007/S10664-010-9150-8. URL `https://doi.org/10.1007/s10664-010-9150-8`. 2.1, 3.1, 5, 5.1

[213] Martin P Robillard, Wesley Coelho, and Gail C Murphy. How effective developers investigate source code: An exploratory study. *IEEE Transactions on software engineering*, 30 (12):889–903, 2004. 1.1.1

[214] Martin P Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. Automated API property inference techniques. *Transactions on Software Engineering*, 39

(5):613–637, 2013. 4.1

[215] Martin P. Robillard, Andrian Marcus, Christoph Treude, Gabriele Bavota, Oscar Chaparro, Neil A. Ernst, Marco Aurélio Gerosa, Michael W. Godfrey, Michele Lanza, Mario Linares Vásquez, Gail C. Murphy, Laura Moreno, David C. Shepherd, and Edmund Wong. On-demand developer documentation. In *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017*, pages 479–483. IEEE Computer Society, 2017. doi: 10.1109/ICSME.2017.17. URL `https://doi.org/10.1109/ICSME.2017.17`. 3.1, 8.1

[216] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. Learning syntactic program transformations from examples. pages 404–415, 2017. doi: 10.1109/ICSE.2017.44. URL `https://doi.org/10.1109/ICSE.2017.44`. 6.2

[217] Charles Romesburg. *Cluster analysis for researchers*. Lulu. com, 2004. 3.3.1

[218] Christoffer Rosen and Emad Shihab. What are mobile developers asking about? a large scale study using stack overflow. *Empirical Softw. Engg.*, 21(3):1192–1223, jun 2016. ISSN 1382-3256. doi: 10.1007/s10664-015-9379-3. URL `https://doi.org/10.1007/s10664-015-9379-3`. 2.1

[219] Steven I. Ross, Fernando Martinez, Stephanie Houde, Michael J. Muller, and Justin D. Weisz. The programmer's assistant: Conversational interaction with a large language model for software development. In *Proceedings of the 28th International Conference on Intelligent User Interfaces, IUI 2023, Sydney, NSW, Australia, March 27-31, 2023*, pages 491–514. ACM, 2023. doi: 10.1145/3581641.3584037. URL `https://doi.org/10.1145/3581641.3584037`. 2.4, 7.3, 8.1

[220] Chanchal Kumar Roy and James R Cordy. A survey on software clone detection research. *Queen's School of Computing TR*, 541(115):64–68, 2007. 4.2.3, 4.2.3

[221] Riccardo Rubei, Claudio Di Sipio, Phuong T Nguyen, Juri Di Rocco, and Davide Di Ruscio. Postfinder: Mining stack overflow posts to support software developers. *Information and Software Technology*, 127:106367, 2020. 2.4

[222] Advait Sarkar, Andrew D Gordon, Carina Negreanu, Christian Poelitz, Sruti Srinivasa Ragavan, and Ben Zorn. What is it like to program with artificial intelligence? *arXiv*, 2022. doi: 10.48550/arxiv.2208.06213. 2.4

[223] Sami Sarsa, Paul Denny, Arto Hellas, and Juho Leinonen. Automatic generation of programming exercises and code explanations using large language models. In *ACM Conference on International Computing Education Research (ICER 2022), Lugano and Virtual Event, Switzerland, August 7 - 11, 2022, Volume 1*, pages 27–43. ACM, 2022. doi: 10.1145/3501385.3543957. URL `https://doi.org/10.1145/3501385.3543957`. 2.4

[224] M. Schuster and K.K. Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, 1997. doi: 10.1109/78.650093. 18

[225] SeanCurtis-TRI. Collect GeometrySystem → drake_visualizer boilerplate by SeanCurtis-TRI pull request #8526 RobotLocomotion/drake. `https://github.com/RobotLocomotion/drake/pull/8526`. 4.1, 4.2

[226] David E. Shaw, William R. Swartout, and C. Cordell Green. Inferring lisp programs from examples. In *Proceedings of the 4th International Joint Conference on Artificial Intelligence*

- *Volume 1*, IJCAI'75, pages 260–267, San Francisco, CA, USA, 1975. Morgan Kaufmann Publishers Inc. 6.2

[227] Mary Shaw. Prospects for an engineering discipline of software. *IEEE Software*, 7(6):15–24, 1990. 5.1

[228] Xuehua Shen, Bin Tan, and ChengXiang Zhai. Implicit user modeling for personalized search. *Proceedings of the 14th ACM international conference on Information and knowledge management - CIKM '05*, pages 824–831, 2005. doi: 10.1145/1099554.1099747. 3.5.3

[229] Kensen Shi, David Bieber, and Rishabh Singh. Tf-coder: Program synthesis for tensor manipulations. *CoRR*, abs/2003.09040, 2020. URL `https://arxiv.org/abs/2003.09040`. (document), 6.1, 6.1, 6.1, 6.1, 6.2, 6.5, 6.6.1, 6.6.3, B.2

[230] Peng Shi and Jimmy Lin. Simple bert models for relation extraction and semantic role labeling. *arXiv preprint arXiv:1904.05255*, 2019. 2.3

[231] Richard Shin, Illia Polosukhin, and Dawn Song. Improving neural program synthesis with inferred execution traces. pages 8931–8940, 2018. URL `https://proceedings.neurips.cc/paper/2018/hash/7776e88b0c189539098176589250bcba-Abstract.html`. 6.2

[232] Richard Shin, Neel Kant, Kavi Gupta, Chris Bender, Brandon Trabucco, Rishabh Singh, and Dawn Song. Synthetic datasets for neural program synthesis. 2019. URL `https://openreview.net/forum?id=ryeOSnAqYm`. 6.3

[233] Jonathan Sillito, Kris De Voider, Brian Fisher, and Gail Murphy. Managing software change tasks: An exploratory study. In *2005 International Symposium on Empirical Software Engineering, 2005.*, pages 10–pp. IEEE, 2005. 1.1.1

[234] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. Asking and answering questions during a programming change task. *IEEE Transactions on Software Engineering*, 34(4): 434–451, 2008. doi: 10.1109/TSE.2008.26. 2.1, 3.1

[235] Calvin Smith and Aws Albarghouthi. Mapreduce program synthesis. pages 326–340, 2016. doi: 10.1145/2908080.2908102. URL `https://doi.org/10.1145/2908080.2908102`. 6.2

[236] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. pages 404–415, 2006. doi: 10.1145/1168857.1168907. URL `https://doi.org/10.1145/1168857.1168907`. 6.2

[237] Nitish Srivastava, Geoffrey E. Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, 2014. doi: 10.5555/2627435.2670313. URL `https://dl.acm.org/doi/10.5555/2627435.2670313`. 5.4.2

[238] Cuyler Stuwe. Boilerplate code definition of quora. `https://www.quora.com/What-is-boilerplate-code`. 4.2.2

[239] Jeffrey Stylos and Brad A Myers. The implications of method placement on API learnability. In *International Symposium on Foundations of software engineering*, pages 105–112. ACM, 2008. 4.1, 5, 5.1

[240] Ning Su, Jiyin He, Yiqun Liu, Min Zhang, and Shaoping Ma. User intent, behaviour, and perceived satisfaction in product search. In Yi Chang, Chengxiang Zhai, Yan Liu, and Yoelle Maarek, editors, *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining, WSDM 2018, Marina Del Rey, CA, USA, February 5-9,*

2018, pages 547–555. ACM, 2018. doi: 10.1145/3159652.3159714. URL https://doi.org/10.1145/3159652.3159714. 3.1

[241] Harsh Suri. Purposeful sampling in qualitative research synthesis. *Qualitative research journal*, 11(2):63–75, 2011. ISSN 1443-9883. 3.2.1

[242] Jeniya Tabassum, Mounica Maddela, Wei Xu, and Alan Ritter. Code and named entity recognition in Stack Overflow. In *Annual Meeting of the Association for Computational Linguistics* (*ACL*), pages 4913–4926. Association for Computational Linguistics, 2020. 5.4, 14

[243] Hengzhu Tang, Yanan Cao, Zhenyu Zhang, Jiangxia Cao, Fang Fang, Shi Wang, and Pengfei Yin. Hin: Hierarchical inference network for document-level relation extraction. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 197–209. Springer, 2020. 5.4.7

[244] taynaud. Community detection package. https://python-louvain.readthedocs.io. 4.4.1

[245] Kyle Thayer, Sarah E Chasins, and Amy J Ko. A Theory of Robust API Knowledge. *ACM Transactions on Computing Education*, 21(1):1–32, 2021. doi: 10.1145/3444945. 3.1, 5.1, 7.2.2, 7.3

[246] Haoye Tian, Weiqi Lu, Tsz On Li, Xunzhu Tang, Shing-Chi Cheung, Jacques Klein, and Tegawendé F. Bissyandé. Is chatgpt the ultimate programming assistant – how far is it?, 2023. Retrieved from https://arxiv.org/abs/2304.11938. 2.4

[247] Yuan Tian, Ke Zhou, and Dan Pelleg. Characterization and prediction of mobile tasks. *ACM Trans. Inf. Syst.*, 41(1):13:1–13:39, 2023. doi: 10.1145/3522711. URL https://doi.org/10.1145/3522711. 3.3.2

[248] Tim. How to avoid writing duplicate boilerplate code for requesting permissions? https://stackoverflow.com/questions/39080095/how-to-avoid-writing-duplicate-boilerplate-code-for-requesting-permissions. 4.2.2

[249] Nava Tintarev and Judith Masthoff. Evaluating the effectiveness of explanations for recommender systems - methodological issues and empirical studies on the impact of personalization. *User Model. User Adapt. Interact.*, 22(4-5):399–439, 2012. doi: 10.1007/s11257-011-9117-5. URL https://doi.org/10.1007/s11257-011-9117-5. 3.5.3

[250] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models, 2023. 1.1.2, 2.3

[251] Christoph Treude and Martin P Robillard. Augmenting API documentation with insights from stack overflow. *ICSE*, 2016. doi: 10.1145/2890000/2884800/p392-treude.pdf. 1.1.2, 2.4, 3.1, 5.1

[252] Christoph Treude, Ohad Barzilay, and Margaret-Anne Storey. How do programmers ask and answer questions on the web? (nier track). In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 804–807, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450304450. doi: 10.1145/1985793.1985907. URL `https://doi.org/10.1145/1985793.1985907`. 2.1, 8.2.1

[253] Jaroslav Tulach. *Practical API Design: Confessions of a Java Framework Architect*. Apress, September 2008. 4.1

[254] Gias Uddin and Foutse Khomh. Opiner: an opinion search and summarization engine for apis. In Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen, editors, *International Conference on Automated Software Engineering* (*ASE*), pages 978–983. IEEE, 2017. 2.4

[255] Gias Uddin and Martin P. Robillard. How API documentation fails. *IEEE Softw.*, 32(4): 68–75, 2015. doi: 10.1109/MS.2014.80. URL `https://doi.org/10.1109/MS.2014.80`. 2.1, 8.1

[256] Gias Uddin, Olga Baysal, Latifa Guerrouj, and Foutse Khomh. Understanding how and why developers seek and analyze api-related opinions. *IEEE Transactions on Software Engineering*, 47(4):694–735, 2019. 2.1

[257] Gias Uddin, Foutse Khomh, and Chanchal K Roy. Mining api usage scenarios from stack overflow. *Information and Software Technology*, 122:106277, 2020. 2.4

[258] Gias Uddin, Foutse Khomh, and Chanchal K Roy. Automatic api usage scenario documentation from technical q&a sites. *ACM Transactions on Software Engineering and Methodology* (*TOSEM*), 30(3):1–45, 2021. 2.4, 5.4.6

[259] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M. K. Martin, and Rajeev Alur. TRANSIT: specifying protocols with concolic snippets. pages 287–296, 2013. doi: 10.1145/2491956.2462174. URL `https://doi.org/10.1145/2491956.2462174`. 6.1, 6.2, 6.5

[260] user2999943. How to avoid boilerplate code when loading images with picasso library. `https://stackoverflow.com/questions/32167948/how-to-avoid-boilerplate-code-when-loading-images-with-picasso-library`. 4.4.2

[261] Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. *CHI Conference on Human Factors in Computing Systems Extended Abstracts*, pages 1–7, 2022. doi: 10.1145/3491101.3519665. 2.4, 7.1, 7.8

[262] Enrique Larios Vargas, Maurício Finavaro Aniche, Christoph Treude, Magiel Bruntink, and Georgios Gousios. Selecting third-party libraries: the practitioners' perspective. In Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann, editors, *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pages 245–256. ACM, 2020. doi: 10.1145/3368089.3409711. URL `https://doi.org/10.1145/3368089.3409711`. 4, 3.5.2

[263] Marcello Visconti and Curtis R. Cook. Assessing the state of software documentation practices. In Frank Bomarius and Hajimu Iida, editors, *Product Focused Software Process Improvement, 5th International Conference, PROFES 2004, Kausai Science City, Japan, April 5-8, 2004, Proceedings*, volume 3009 of *Lecture Notes in Computer Science*, pages 485–496. Springer, 2004. doi: 10.1007/978-3-540-24659-6_35. URL `https://doi.org/10.1007/978-3-540-24659-6_35`. 3.1

[264] Han Wang, Chunyang Chen, Zhenchang Xing, and John Grundy. DiffTech: a tool for differencing similar technologies from question-and-answer discussions. *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1576–1580, 2020. doi: 10.1145/3368089.3417931. (document), 1.1.2, 5.1, 5.3, 5.4.4, 5.4.4, 5.4.4, 5.4.4

[265] Roger C S Wernersson. Can java help me avoid boilerplate code in equals()? `https://stackoverflow.com/questions/25183872/can-java-help-me-avoid-boilerplate-code-in-equals`. 4.1, 4.2

[266] Jake Wharton. Butterknife. `https://jakewharton.github.io/butterknife/`. 4.4.2

[267] Ronald J. Williams and David Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural Comput.*, 1(2):270–280, 1989. doi: 10.1162/neco.1989.1.2.270. URL `https://doi.org/10.1162/neco.1989.1.2.270`. 6.1

[268] Alexandra Wood, Micah Altman, Aaron Bembenek, Mark Bun, Marco Gaboardi, James Honaker, Kobbi Nissim, David R O'Brien, Thomas Steinke, and Salil Vadhan. Differential privacy: A primer for a non-technical audience. *Vanderbilt journal of entertainment and technology law*, 21(1):209–, 2018. ISSN 1942-678X. 3.2.3

[269] Yuhao Wu, Shaowei Wang, Cor-Paul Bezemer, and Katsuro Inoue. How do developers utilize source code from stack overflow? *Empirical Software Engineering*, 24:637–673, 2019. 7.8

[270] Xin Xia, Lingfeng Bao, David Lo, Pavneet Singh Kochhar, Ahmed E Hassan, and Zhenchang Xing. What do developers search for on the web? *Empirical Software Engineering*, 22:3149–3185, 2017. 1, 7.1, 7.6.2, 8.1

[271] Tao Xiao, Christoph Treude, Hideaki Hata, and Kenichi Matsumoto. Devgpt: Studying developer-chatgpt conversations. *CoRR*, abs/2309.03914, 2023. doi: 10.48550/ARXIV.2309.03914. URL `https://doi.org/10.48550/arXiv.2309.03914`. 8.2.1

[272] Frank F Xu, Uri Alon, Graham Neubig, and Vincent J Hellendoorn. A Systematic Evaluation of Large Language Models of Code. *arXiv*, 2022. doi: 10.48550/arxiv.2202.13169. 1.1.2, 2.4

[273] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. Sqlizer: query synthesis from natural language. *Proc. ACM Program. Lang.*, 1(OOPSLA):63:1–63:26, 2017. doi: 10.1145/3133887. URL `https://doi.org/10.1145/3133887`. 6.2

[274] Navid Yaghmazadeh, Xinyu Wang, and Isil Dillig. Automated migration of hierarchical data to relational tables using programming-by-example. *Proc. VLDB Endow.*, 11(5):580–593, 2018. doi: 10.1145/3187009.3177735. URL `http://www.vldb.org/pvldb/vol11/p580-yaghmazadeh.pdf`. 6.2

[275] Litao Yan, Alyssa Hwang, Zhiyuan Wu, and Andrew Head. Ivie: Lightweight anchored explanations of just-generated code. *arXiv preprint arXiv:2403.02491*, 2024. 9.2

[276] Yuan Yao, Deming Ye, Peng Li, Xu Han, Yankai Lin, Zhenghao Liu, Zhiyuan Liu, Lixin Huang, Jie Zhou, and Maosong Sun. DocRED: A large-scale document-level relation extraction dataset. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 764–777. Association for Computational Linguistics, 2019. 5.4.1

[277] Xing Yi, Liangjie Hong, Erheng Zhong, Nanthan Nan Liu, and Suju Rajan. Beyond clicks: Dwell time for personalization. In *Proceedings of the 8th ACM Conference on Recommender Systems*, RecSys '14, pages 113–120, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450326681. doi: 10.1145/2645710.2645724. URL `https://doi.org/10.1145/2645710.2645724`. 1.1.1, 3.2.2, 3.5.3

[278] Peifeng Yin, Ping Luo, Wang-Chien Lee, and Min Wang. Silence is also evidence: interpreting dwell time for recommendation from psychological perspective. In Inderjit S. Dhillon, Yehuda Koren, Rayid Ghani, Ted E. Senator, Paul Bradley, Rajesh Parekh, Jingrui He, Robert L. Grossman, and Ramasamy Uthurusamy, editors, *The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2013, Chicago, IL, USA, August 11-14, 2013*, pages 989–997. ACM, 2013. doi: 10.1145/2487575.2487663. URL `https://doi.org/10.1145/2487575.2487663`. 1.1.1, 3.5.3

[279] Pengcheng Yin and Graham Neubig. A syntactic neural model for general-purpose code generation. pages 440–450, 2017. doi: 10.18653/v1/P17-1041. URL `https://doi.org/10.18653/v1/P17-1041`. 6.2

[280] J. D. Zamfirescu-Pereira, Richmond Y. Wong, Bjoern Hartmann, and Qian Yang. Why johnny can't prompt: How non-ai experts try (and fail) to design LLM prompts. In Albrecht Schmidt, Kaisa Väänänen, Tesh Goyal, Per Ola Kristensson, Anicia Peters, Stefanie Mueller, Julie R. Williamson, and Max L. Wilson, editors, *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems, CHI 2023, Hamburg, Germany, April 23-28, 2023*, pages 437:1–437:21. ACM, 2023. doi: 10.1145/3544548.3581388. URL `https://doi.org/10.1145/3544548.3581388`. 7.4.1, 8.1

[281] Tianyi Zhang, Ganesha Upadhyaya, Anastasia Reinhardt, Hridesh Rajan, and Miryung Kim. Are code examples on an online q&a forum reliable?: A study of api misuse on stack overflow. In *International Conference on Software Engineering* (*ICSE*), pages 886–896. IEEE, 2018. 2.4, 4.1, 5.1, 5.5, 6.8, 7.8

[282] Tianyi Zhang, Björn Hartmann, Miryung Kim, and Elena L. Glassman. Enabling data-driven API design with community usage data: A need-finding study. In Regina Bernhaupt, Florian 'Floyd' Mueller, David Verweij, Josh Andres, Joanna McGrenere, Andy Cockburn, Ignacio Avellino, Alix Goguey, Pernille Bjøn, Shengdong Zhao, Briane Paul Samson, and Rafal Kocielnik, editors, *CHI '20: CHI Conference on Human Factors in Computing Systems, Honolulu, HI, USA, April 25-30, 2020*, pages 1–13. ACM, 2020. doi: 10.1145/3313831.3376382. URL `https://doi.org/10.1145/3313831.3376382`. 3.1

[283] Sha Zhao, Julian Ramos, Jianrong Tao, Ziwen Jiang, Shijian Li, Zhaohui Wu, Gang Pan, and Anind K. Dey. Discovering different kinds of smartphone users through their application usage behaviors. In Paul Lukowicz, Antonio Krüger, Andreas Bulling, Youn-Kyung Lim, and Shwetak N. Patel, editors, *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing, UbiComp 2016, Heidelberg, Germany, September 12-16, 2016*, pages 498–509. ACM, 2016. doi: 10.1145/2971648.2971696.

URL https://doi.org/10.1145/2971648.2971696. 3.3.2, 3.3.2

[284] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging llm-as-a-judge with mt-bench and chatbot arena. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine, editors, *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023. URL http://papers.nips.cc/paper_files/paper/2023/hash/91f18a1287b398d378ef22505bf41832-Abstract-Datasets_and_Benchmarks.html. 8.3.1, 8.3.2

[285] Albert Ziegler, Eirini Kalliamvakou, Shawn Simister, Ganesh Sittampalam, Alice Li, Andrew Rice, Devon Rifkin, and Edward Aftandilian. Productivity Assessment of Neural Code Completion. *arXiv*, 2022. doi: 10.48550/arxiv.2205.06537. 2.4, 7.1