Thesis Proposal: Context-Aware Information Support for Developers

by

Daye Nam

Committee in charge:

Bogdan Vasilescu, Co-chair
Brad Myers, Co-chair
Vincent Hellendoorn, Co-chair
Baishakhi Ray, Columbia University
Andrew Macvean, Google, Inc.

Spring 2024

Abstract

Thesis Proposal: Context-Aware Information Support for Developers

by

Daye Nam

Doctor of Philosophy in Software Engineering

Carnegie Mellon University

Bogdan Vasilescu, Co-chair
Brad Myers, Co-chair
Vincent Hellendoorn, Co-chair
Baishakhi Ray, Columbia University
Andrew Macvean, Google, Inc.

Software engineering is an information-intensive discipline. To build a software system, developers need to consider many dimensions, from architecture to implementation bugs, each requiring information with different scopes and abstraction levels. However, information-seeking is not easy in software engineering. Much information about software and systems needs to be clearly documented or spread across various mediums in different formats, often becoming outdated as software evolves. Moreover, most existing information support tools in software engineering were made with a single "average" developer in mind.

This thesis proposes context-aware information support tools in software engineering to address this issue. Specifically, I propose to use learning-based natural language processing approaches to extract or generate information needed by developers and present such information considering developers' specific context, such as their current tasks or developers' characteristics. I first present an exploratory, mixed-methods empirical study on documentation page-view logs, revealing discernible documentation page visit patterns. The study shows that developers use documentation differently, and developers' contextual factors correlate with their documentation usage, confirming the potential of context-aware information support. I then introduce three prototype information support tools designed to assist developers in writing code when working with unfamiliar libraries. Evaluations of these approaches demonstrate the effectiveness of learning-based approaches, 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, I intend to develop an information support tool that broadens the context scope. It will incorporate

a personalization feature, which considers developers' experience in the domain and their specific information support needs related to APIs.

The expected outcome of my thesis is to demonstrate the effectiveness of context utilization in information support, leading to better developer productivity and software quality. The methodologies proposed to build context-aware information support tools, along with the insights gained from user studies in this thesis, will further our understanding of how developers with varying contexts seek and use information. Taken together, the thesis will shed light on the design of future development tools, especially those that will be built for a new programming paradigm that relies heavily on AI-based support, such as CoPilot or ChatGPT.

Contents

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. For that, developers spend a significant amount of time searching and foraging for the information they need and organizing and digesting the information they find [27, 63, 31, 53, 35, 48, 44] This information is often scattered across various sources, making it difficult for developers to find what they need. Developers face the challenge of accessing a wide range of information, from architectural considerations (e.g., Why was this code implemented this way? [28]) to implementation bugs (e.g., How did this runtime state occur? [34]).

It is more challenging when a developer needs to work with unfamiliar code or libraries that require them to learn new concepts or frameworks. Of course, there are a lot of documentation and supporting materials available on the Web. However, when a developer is not familiar with the domain and the environment, they may lack the necessary knowledge and experience to effectively navigate and understand the codebase [28]. Thus, developers 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, as much material is written by someone who is already familiar with the system. Thus, for decades, researchers and practitioners have looked for ways to improve developers' information seeking for learning, by cataloguing problems in existing learning resources [11, 70, 72], identifying desirable quality attributes for such resources [5, 14], and recommending best practices [90, 70, 72].

1.1 Challenges and Potential Solutions in Information Seeking for Learning

Despite these efforts, developers still face challenges in understanding and learning new libraries and code. In my thesis, I point out the following two main challenges and propose techniques for addressing each of them.

Hidden or Missing Information

For software engineering, 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). In many cases, they document information that can be useful for themselves or for others to reuse and maintain the software. However, as it is practically impossible and inefficient to document every piece of information possible, such documentation often ends up being incomplete. Furthermore, as software evolves, the information gets outdated or obsolete. The multi-modal nature (code + text) of software engineering makes it even harder for developers to collect relevant information in one place, as it can be dispersed across various mediums and formats.

To address this limitation, automated information support tools have been developed to supplement existing learning materials such as documentation. These tools extract implicit and unstructured information [84, 37, 26, 91] from diverse software repositories so that users accessing the learning materials like documentation can easily discover richer information. However, despite the proven usefulness of such tools, the types of information existing tools provide are limited, because most of them employ rule-based approaches, by employing a predetermined set of syntactic patterns that are likely to that are typically derived from manual inspection. This type of approach has the benefit of simplicity, but generally suffers from low recall when relations 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.

Solution1: Learning-based Information Extraction To provide developers with appropriate information, it is essential to have a well-prepared knowledge base. To overcome the aforementioned limitations of pattern-matching-based extraction approaches, I propose to use **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.

Solution2: Information Generation with Language Models However, not every piece of information is documented and can be extracted, so sometimes, it should be inferred based on what is already available. Ideally, if we can **generate information with large language models** that learned the patterns already and has a good enough understanding [**?** 83, 93], it can fill in such holes, by adapting information included in the documentation of similar libraries or sub-packages. The generation will also allow the system to provide information that fits into users' needs, instead of providing existing information that is most similar to the users' needs.

Difficulties in Spotting Needed Information

Even when all necessary information is available, conveying it effectively to developers poses challenges. The volume of information required for learning can overwhelm developers if presented all at once, leading to information overload and difficulty in finding specific information. On the other hand, presenting smaller pieces of information upon request may result in critical information being missed, as developers 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.

Solution: Context-aware Information Presentation. To address these challenges, I propose the use of context-aware information presentation. Context, defined as all factors influencing a computation except explicit input and output, has proven effective in other domains such as media streaming and search engines [81]. 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 [95]. Therefore, previous research in web search has made strides in understanding users' context using both both implicit (e.g., dwell time) [95, 96] and explicit (e.g., item rating) [2, 3] feedback mined from historical interaction data.

Similar to the users of other domains, developers 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 [20, 30] in software maintenance tasks [71, 75]. 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 developed by a large software company [57], I also showed that developers have discernible usage patterns when they use documentation, and developers' contextual factors, such as past experience with a specific product, do correlate with which documentation pages developers visit. Thus, similar to the web search systems, I believe that developers' information seeking can be improved by providing them with more context-aware information support, and I propose to use multiple sources to identify developers' personal and task contexts, such as software repositories (e.g., GitHub), profiles (e.g., LinkedIn profiles), task context (e.g., code the developer is working on) and local interaction data (e.g., search queries).

1.2 Thesis

Combining the two solutions I proposed, my thesis aims to investigate the following claim:

By leveraging learning-based NLP techniques to extract or generate knowledge from software artifacts and by presenting this knowledge in a context-aware manner, we can enhance the success of developers' information-seeking for learning.

1.3 Outline

To explore this claim, I conducted a step-by-step investigation so far, and propose one further project to test the thesis claim. In Chapter 2, I discuss the history of studies of developers' information-seeking and information-support tools. In Chapter 3, I report on my study that confirms that developers do use documentation differently depending on their user characteristics, which was missing from the literature.

Then, in the following chapters, I discuss several information support tools of my own, each of which was built to test the feasibility and usefulness of the two solutions I proposed. Table 1.1 summarizes the tools included in this proposal. The first two tools, in Chapter 4 and Chapter 5, were designed considering a specific user task, where users must write code using unfamiliar libraries. I implemented the two tools using two different information preparation approaches, to show the feasibility of using a learning-based information extraction approach (compared with pattern-matching-based approaches), and using a generative language model. Then, in Chapter 6, I test the benefit of generating information suited for the users' task context, by considering the code they are working on in information support. Finally, in Chapter 7, I propose a study to test the benefit of utilizing user characteristics in preparing information.

Table 1.1: Overview of the information support tools included in this proposal.

Chapt.	**Type of Support**	**Contexts Used for Info. Support**	**Information Preparation**	**Information Presentation**
4	Comparable API Methods	User task	Extraction (Learning-based)	Chrome plugin
5	API methods Sequence	User task	Generation (Language model)	-
6	API-relevant information, Free-form response	User task, Task context	Generation (LLM)	In-IDE plugin
7	API-relevant information, Free-form response	User task, Task context, User characteristics	Generation (LLM)	In-IDE plugin

For each chapter, I provide guiding questions to describe the goal of each work. 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, especially how they search for and acquire information, as developers need a variety of kinds of knowledge [28, 47, 76, 82, 97]. There is some prior work on the information-seeking strategies of developers, mostly in general software maintenance [28, 36, 21] or web search settings [8, 67]. Efforts to understand and improve developers' information-seeking in learning new systems or libraries were mainly made around software documentation, which is the main source of information for developers. Researchers have catalogued problems developers face when using documentation [11, 70, 72], identified desirable quality attributes for documentation [5, 14], and recommended best practices [90, 70, 72]. Some of these studies provided concrete insights into what developers need from the documentation. For example, developers have expressed the need for complete and up-to-date documentation [4], because many developers rely on API reference information and code examples [61, 51] when they approach documentation with a problem or task in mind [51]. Developers also asked for a concise overview of the documentation, more rationale, and adequate explanation for code examples [70, 72, 89, 51]. 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., [7, 42, 55, 45, 1, 85].

To overcome some of these difficulties, 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., [62, 84, 79, 25, 68]. Researchers have also been developing tools to more closely integrate such knowledge into the development workflow, e.g., [64, 65, 79, 88]. Many types of knowledge have been in focus, including common use and misuse patterns [6, 98, 87, 68, 88], caveats [84, 39, 37], opinions on different quality attributes (e.g., usability) [86, 41, 10, 40, 69], or more generally any Stack Overflow posts discussing some given API methods, such as those invoked in the developer's local IDE context [73].

Most of these words to improve developers' information seeking for learning, however,

have mostly focused on the learning materials and the tools, rather than the developers who learn libraries and systems, despite many studies in developers' general information seeking reporting showing that developers' information-seeking may vary with contextual factors, including their experience [30, 18, 49, 35, 38, 18, 66], roles, and learning styles [38, 13, 18, 52]. For example, Costa *et al.* [13] 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. Similarly, in the literature, programmers are sometimes categorized into three personas that summarize their information-seeking and problem-solving strategies – systematic, opportunistic, and pragmatic [12] – that reportedly also correlate with documentation use [52].

In my thesis, I aim to fill in this gap by understanding how different contextual factors affect developers' information seeking for learning, and build information support tools for their learning with their contextual factors taken into account.

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, by answering the following guiding questions:

- RQ3.1: What are the different documentation usage patterns?
- RQ3.2: To what extent do documentation visit logs correlate with user characteristics (e.g., experience)?

3.1 Page-view Log Analysis

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

Product Selection. We selected four popular Google Cloud products 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 each product).

Preprocessing. For each of the four products, we had access to pseudonymized documentation page-view logs [43] for users who visited the documentation from May 1, 2020 to May 31, 2020, UTC, which include the specific documentation pages visited by someone, as

[1]This chapter is adapted from Nam *et al.* [57]

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

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

well as the timestamps and dwell times for each visit. We use *dwell time* to estimate user engagement with the content, following prior work [19, 95], after aggregating them at month-level. 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* [18]) summarized in Table 3.1. We also collected pseudonymized user-level data informed by the literature on programmer information seeking [29, 21, 28, 36] to help us understand how the following user factors are associated with documentation use, including Experience, Documentation Type Predisposition, and Possible Intent. 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), broken down by documentation page type as above. We measured the documentation users' experience level using two variables: *overall platform experience* and *specific product experience*. We defined the experience with the platform as the user account age, i.e., years passed since signing the platform terms and conditions. We defined 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). As a proxy for possible user intent when accessing the documentation, we recorded the *average per-page dwell time*, based on Brandt *et al.* [8]. We 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.* [8].

Clustering Analysis

As an initial investigation, we conducted a cluster analysis to explore patterns of documentation usage discernible in the page-view log data. For clustering, we represented each user's documentation visit profile as a vector of 11 elements, capturing the total times spent across each page type that month. Then, we adopted a protocol proposed by Zhao *et al.* [99], which is particularly well suited for large datasets. Applying this two-step protocol to our data resulted in 320 clusters.

Exploration of Clustering Results. We qualitatively explored the results of our clustering analysis, to get a better sense of in what ways the clusters differed, and how the differences might be associated with user characteristics and behaviors. We formulated preliminary hypotheses based on large clusters (> 1000 users) that displayed distinctive dimensions of usage data compared to the "average user" in the entire dataset. To help with our exploration and comparison of clusters, we first visualized each cluster as its "average user," i.e., a fictional user whose dwell time values across the 11 documentation types were equal to the average values of all users assigned to that cluster. To ease interpretation, we further 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%). Then, for each cluster, we qualitatively compared the resulting distributions of values across four dimensions (the product itself, the users' overall platform and specific product experience, and the average page dwell time as a proxy for possible intent; recall the discussion in Section 3.1 above) to the "average user" across all clusters, to identify clear differences in user characteristics.

Hypotheses. In the end, we built the following 5 hypotheses from the qualitative exploration of the clusters:

$\mathbf{H}_1$. *High platform experience levels are positively associated with viewing documentation that provides system-level protocols (typically guide-genre pages) and negatively with other documentation types, especially landing and marketing pages.*

$\mathbf{H}_2$. *Documentation usage of large-scale infrastructural APIs differs from that of application APIs.*

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

$\mathbf{H}_4$. *Users with long average page dwell times are more likely to visit tutorials and how-to documentation, while users with short average page dwell times are more likely to visit reference pages, marketing, and admin documentation.*

$\mathbf{H}_5$. *High product experience levels are positively associated with accessing documentation covering implementation details (dev and guide genres) over other types of documentation.*

Regression Analysis

We formally tested the hypotheses above on our entire sample to assess how well they are supported beyond the specific clusters where we qualitatively derived them. We used multiple regression to test how much the various user-level characteristics we hypothesized about in $\mathbf{H}_1$–$\mathbf{H}_5$ can explain people's logged documentation visits to pages in each of our four genres (recall Table 3.1).

Modeling Considerations. 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), $\mathbf{H}_2$ (product), $\mathbf{H}_3$ (page views in the previous

three months), $\mathbf{H}_4$ (average page dwell time), and $\mathbf{H}_5$ (specific product experience). 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 the likelihood of accessing documentation genres being modeled. Similarly, we could test for correlations between platform experience and the likelihood of accessing documentation pages from the other three genres with the other three models.

Results

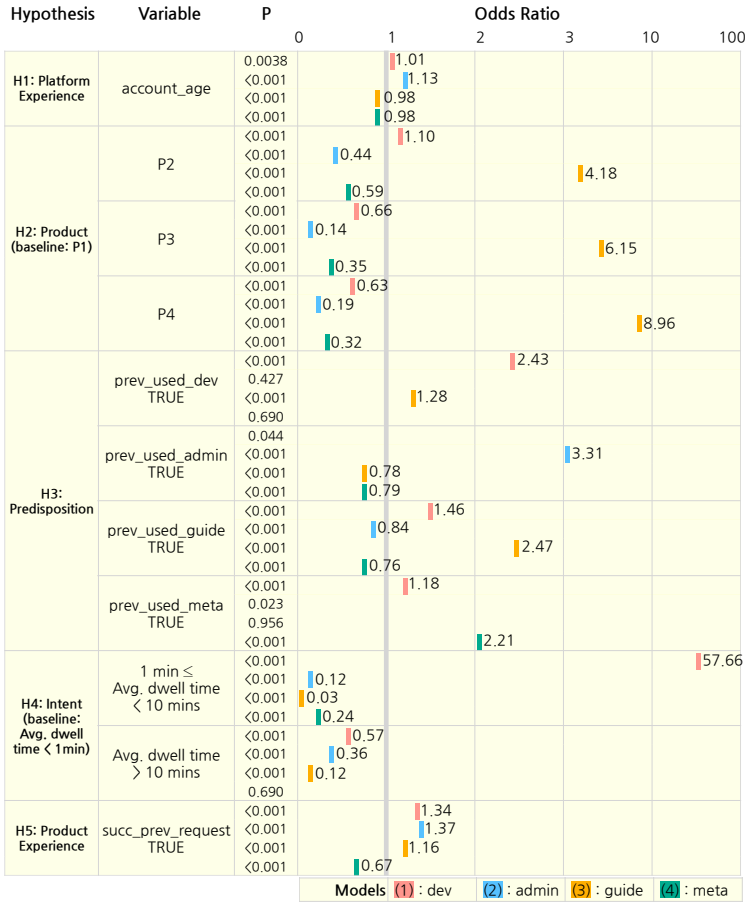

Figure 3.1: Estimated odds ratios from the regression modeling *dwell time* > 0 for our four documentation genres. Variables without statistically significant coefficients (adjusted $p \geq 0.01$) are omitted.

Figure 3.1 summarizes the documentation-access logistic regression results across the four models we estimated (one per genre) to test $\mathbf{H}_1$–$\mathbf{H}_5$. We present our results in terms

of odds ratios (OR) instead of regression coefficients to ease interpretation. All four models are plausible, with Nagelkerke [56] pseudo R^2 values (deviance explained) of 74% for dev, 16% for admin, 44% for guide, and 55% for the meta documentation genre.

H_1 (platform experience): only partially supported. Focusing on the model for guide-genre documentation in Figure 3.1, we observe a negative but small correlation between platform experience and the likelihood of accessing tutorials, how-to, quickstart, or concept guides: the odds of accessing such pages are 0.98 times as high among users with one extra year of platform experience, contrary to our hypothesis. The meta-genre model supports our hypothesis, given the negative correlation between platform experience and the likelihood of accessing landing and marketing pages (OR = 0.98). Beyond our specific hypothesis, we also note a positive correlation in the admin-genre model—the odds of accessing pricing and other admin pages are higher among users with more platform experience—perhaps indicative of higher conscientiousness among more experienced users when it comes to pricing.

H_2 (product type): supported. All four models support the hypothesis: taking *Translation AI* as the reference, the magnitude of differences between *Translation AI* and *Natural Language AI* is consistently smaller than either *Translation AI* and *Cloud Logging* or *Translation AI* and *Pub/Sub*, 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 *Natural Language AI* documentation compared to *Translation AI*, but 0.66 and 0.63 times as high among visitors to *Cloud Logging* and *Pub/Sub* compared to *Translation AI*.

H_3 (documentation type predisposition): supported. All models show strong effects of documentation type consistency: in general, past access to pages of some type is the strongest predictor of future visits to those page types. E.g., 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.

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.

$\mathbf{H}_5$ (product experience): supported. Results from the dev, guide, 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 (1.16 times higher for tutorials, how-to, and other guide pages) among people with prior experience with the products, i.e., those who made successful API requests in the past, compared to those without. Similarly, the odds of accessing marketing and other meta documentation are lower (OR = 0.67) among people with prior experience with the products.

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.

3.2 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 ($\mathbf{H}_1$, $\mathbf{H}_5$), 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

Information Support for Programming with Unfamiliar Libraries[2]

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* [72, 78, 17]). 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.

Overall, in this section, we focused on answering the following research questions:

- RQ4.1: Does providing comparable API methods proactively help developers learn

[2]This chapter is adapted from Nam *et al.* [58]

Table 4.1: 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

unfamiliar libraries?

- RQ4.2: How well can a learning-based approach extract comparable API methods from Stack Overflow answers compared to pattern matching-based approaches?

4.1 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 4.1 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.[3]

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 authors 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,

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

both by the free-response kappa proponents [9] and the interpretation guideline for Cohen's kappa [32], we considered the instructions adequate at that point. After this, a single author annotated the remaining documents alone following the final instructions.

Resulting Dataset. Table 4.1 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.

4.2 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 4.1) 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.

Figure 4.1: 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, 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.

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 further recruited 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 [22] (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.

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.

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[4] 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.

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

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).

4.3 Learning-based Information Extraction

To test the effectiveness of utilizing a learning-based information extraction approach, we built SOREL (Stack Overflow RELation extractor), an ML model that collects comparable API methods and relevant supporting natural language sentences from Stack Overflow answers. We aimed 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 identified one or more sentences from the same answer, that offer the evidence supporting each identified relation, providing a type of (extractive) summary for each relation. Therefore, our task was divided into two sub-problems:

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

Model Architecture

Figure 4.2 shows the architecture of our model, which was inspired by Yao et al.'s [94] DocRED. The model consists of three main components:

BERT. We used a language representation model called BERT (Bidirectional Encoder Representations from Transformers) [15] 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 provided via huggingface.[5,6] 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 [24, 23].

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

[6]We also considered BertOverflow [80], which was pre-trained with a Stack Overflow corpus, but found the original BERT to work better.

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 of 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 [94]. 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 [94] for details.

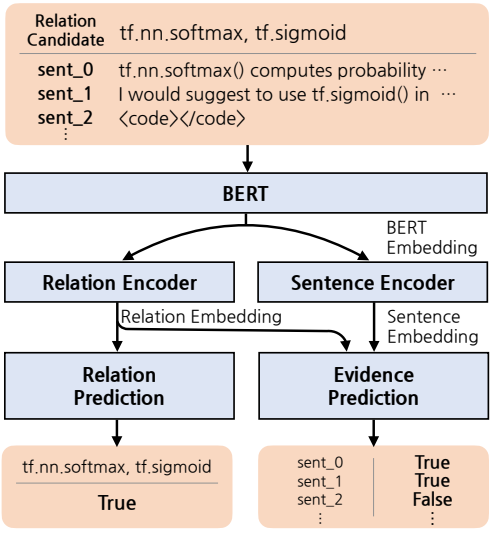

Figure 4.2: The architecture of SOREL, which learns to infer the comparison relation and the supporting evidence.

Sentence Encoder generates contextualized representations for each sentence in the input document. We used 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 backward, 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 4.2). A single-layer, low-dimensional BiLSTM contains comparatively far fewer parameters to calibrate. To obtain a sentence embedding, we used 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.

Model Training

As RE and SEP are highly connected tasks, we trained 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 4.1 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. 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

Table 4.2: Overall performance on each subtask (%).

	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

Table 4.3: Recall comparison with DiffTech [92] and APIComp [46]), on test set (%) after excluding deprecated API methods.

	SOREL	DiffTech	APIComp
RE	55.3	33.3	16.7
SEP	55.2	5.7	26.4

40% of tokens from the input at training time [77]. 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.

Evaluations with Test Data

Table 4.2 summarizes the train / (average) validation / test performance scores. 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.

Comparison with Pattern Matching-based Information Extraction Approaches

To test the usefulness of learning-based information extraction compared to pattern-matching based approaches, we compared with two pattern-matching-based approaches: DiffTech and APIComp. Table 4.3 summarizes the results.

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 [54] 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. For RE, 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. For SEP task, DiffTech [92], 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 [46] 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 previous pattern-matching-based approaches 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.

4.4 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 5

Testing the Feasibility of Generation-based Information Support[7]

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. By using a language model for generation, we can provide information that is suited to the user's needs, as long as the requirement is clearly elaborated. 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 answering the following research question:

- RQ5: How well does a generation-based approach predict a sequence of API methods?

5.1 Information Generation Using Language Model

We designed the language model based on the intuition that for many common API methods, their behavior—the relationship of output to inputs—has simple patterns, and a feed-forward neural network can be trained to predict likely functions—as in a multi-classification problem—from a representation of the input/output data when it is trained on large amounts of input-output examples and their known (ground truth).

[7]This chapter is adapted from Nam *et al.* [59]

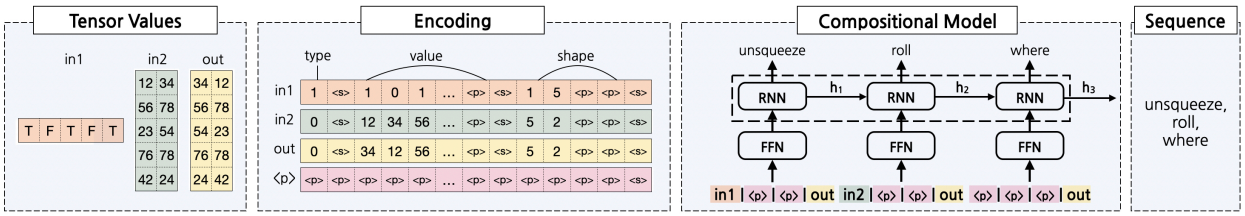

Figure 5.1: Illustration of Compositional Model on an example. The inputs are in the Tensor Values box, and the expected prediction is shown in the Sequence box.

Model Architecture

Figure 5.1 shows three units of the model using input and output examples. We train a model to predict the sequence of API methods $s_f = [f_1, ..., f_n]$, given a task specification $\phi = \{inp, out\}$, where inp is a list of input tensors that have gone through s_f, and out is the final output tensor.

Before passing the input/output tensors to the models, we encode them into a fixed-length vector (Figure 5.1-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., X = type <s> shape <s> 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. 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 fewer than 3 inputs to keep the model input size the same for all examples.

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 methods.

Model Training

Table 5.1: Statistics of the dataset used in this study. Numbers in parentheses indicate the length of the sequences.

	Synthetic			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

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. We synthesized 202 unique sequences by using the exhaustive combination of 16 PyTorch API methods, with 1 or 2-length sequences. For each API method 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). 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. We created the dataset with 100K input/output pairs for each unique API sequence (Table 5.1-Synthetic), and split it into training, validation, and test sets. Each included all 202 API sequences, but the input/output values were not overlapped across the datasets.

Table 5.2: Model accuracy for unseen input/output values.

Synthetic-Test	**Stack Overflow**	
Top-1	**Top-1**	**Top-3**
79.36%	35.29%	76.47%

Evaluation

We evaluated the effectiveness of our approaches with a subset of TF-Coder's SO benchmarks [74]. These benchmarks contain 50 tensor manipulation examples collected from SO, each containing input/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 methods needed for the remaining 36 benchmarks, we selected 16 methods covering 18 benchmarks (Table 5.1-Stack Overflow) from the core utility that modify 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 method solely from input/output pairs (i.e., no side effects).

We trained both the model 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 5.2 shows the result. The model's top-1 testing accuracies of the 10K synthetic test set are ∼79%. Among 18 SO benchmarks, the model found 13 sequences are in the top-3 (72.22%), and among them 6 are in the top-1 (33.33%).

5.2 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 6

Generation-based Information Support Considering Developer's Task Context[8]

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. We utilized the code on which developers are working as their task context for the information support, leading to the main guiding research question of the chapter:

- RQ6: How does the generation-based information support tool that considers their task context affect developers' productivity in working with unfamiliar code?

We investigate this by providing such a tool, backed by powerful Large Language Models (LLMs), to developers tasked with comprehending and extending unfamiliar code that involves new domain concepts and APIs – a challenging task. We developed a VS Code extension prototype that allows developers to interact with LLMs within their IDE. To simulate scenarios where developers are confronted with unfamiliar code, we designed two programming tasks that utilized Python libraries in the domains of data visualization and 3D rendering. We then conducted user studies with 32 participants with varying levels of experience in programming and knowledge in the task application domains. Further details on each step are provided in the following sections.

[8]This chapter is adapted from Nam *et al.* [60]

6.1 Information Support Using LLM

We developed our prototype to explore the user experience of interacting with the LLM information support tool. We aimed to provide multiple ways of using it to identify the most natural and preferable way of interacting with an LLM for information support. We intentionally did not integrate a code generation feature in the prototype as we wanted to focus on how developers *understand* code, which is needed regardless of *who* wrote the code.

In-IDE extension. To enhance the user experience, we designed the prototype as an in-IDE extension. We hypothesized that in-IDE explanation generation would be more effective than using a standalone tool as it reduces context switching. Additionally, by making it an in-IDE extension, we can easily provide the code context into the LLM. Participants could select code to use as part of the context for a query.

Pre-generated prompts. To make the tool accessible to novice programmers or those unfamiliar with the APIs/domains or the LLM, we implemented the tool to support prompt-less LLM triggering. We designed buttons that do not require user prompting by querying the LLM with pre-generated prompts (to ask about an **API**, a **Concept**, or **Usage** examples, as shown in Figure 6.1-4) for commonly needed information to learn APIs based on API learning theory [81, 50, 33, 16].

Unrestricted textual queries. The tool also allows the developer to **Ask a question...** to the LLM (see Figure 6.1-5), in which case our tool 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 the pre-generated template, and request the LLM to respond based on the code context.

Need-based explanation generation. Our tool was designed as a pull system; 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 believe that once enough context can be extracted and utilized from the IDE, making a hybrid (pull + push) will be possible, but this would require additional study.

Iterative design updates. We updated the design of our extension iteratively based on the feedback we received from pilot studies. For example, we updated the tool to provide 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 AI model prompting to support 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 in the pilot studies that the response time was too slow with GPT-4.

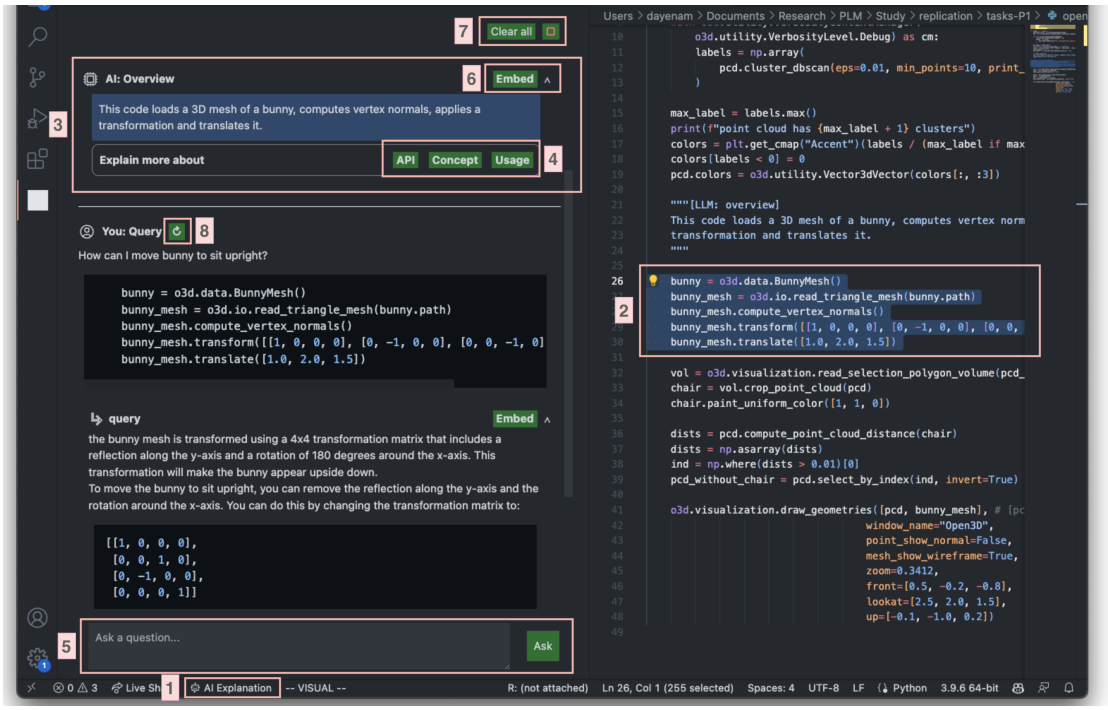

Figure 6.1: Overview of our prototype. (1) A trigger button; (2) code used as context when prompting LLM; (3) code summary (no-prompt trigger); (4) buttons for further details; (5) queries prompts; (6) options to embed information to code (Embed) and a hide/view button; (7) options to clear the panel (Clear all) and an abort LLM button; (8) a refresh button.

Features

There are two ways to trigger the prototype. First, users can select parts of their code and trigger the tool by clicking an "AI Explanation" button on the bottom bar (Figure 6.1-1), or using "alt/option + a" as a short-cut, to receive a summary description of the highlighted code (`Overview`: Figure 6.1-3). They can then explore further by clicking on buttons (Figure 6.1-4) for API (`API`), domain-specific concepts (`Concept`), and usage examples (`Usage`), which will 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 ask a specific question to the model directly to LLM (`Prompt`: Figure 6.1-5). If no code is selected, the entire source code is provided as context. Users can also prompt the LLM with the relevant code highlighted before sending their query (`Prompt-context`). The model will then answer the question with that code as context. Our LLM information support tool 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.

Human Study Design

Participants. After advertising our IRB-approved study widely within the university community and to the public, we recruited 33 participants. To minimize the possibility of participants knowing solutions, we specifically sought out participants who had not used the libraries included in our study.

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. We also included descriptions of each sub-task. 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. We chose two domains, data visualization and 3D rendering, to cover both a common and a less common domain that a given Python developer might encounter in their work, and to allow participants to easily check their progress. When selecting libraries, we intentionally chose ones that are not the most common in their respective domains to minimize participants' dependence on their knowledge of the libraries.

Experimental Design. We chose a within-subjects design, with participants using both our LLM-powered in-IDE support tool (treatment) and 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.

In 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 for code understanding, and read any web pages. In the treatment condition, participants were encouraged to primarily use our prototype for information support.

Analysis

We compared the effectiveness of using a LLM information support tool with traditional search engines for completing programming tasks. We employed regression models to estimate the effects of the LLM information support tool usage on three outcome variables. For the task progress and the code understanding, we employed 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 analysis. Finally, we used a dummy variable to indicate the condition (using the LLM information support tool vs. using search engines). We considered mixed-effects regression but used fixed effects only, since

Table 6.1: Task Performance Models

	Progress	Time	Understanding	Progress	
	(1)	(2)	(3)	Professionals	Students
Constant	0.41	312.65	-1.81**	-0.38	1.82**
	(0.49)	(185.33)	(0.89)	(0.68)	(0.83)
Uses extension	0.47***	-9.10	0.29	0.57**	0.29
	(0.16)	(57.26)	(0.28)	(0.22)	(0.25)
Task experience	0.13*	23.14	0.41***	0.16	0.04
	(0.07)	(25.40)	(0.12)	(0.09)	(0.11)
Programming ex.	-0.10	-23.67	0.20	0.01	-0.37*
	(0.12)	(43.53)	(0.22)	(0.17)	(0.21)
LLM familiarity	-0.01	7.70	-0.09	0.07	-0.10
	(0.07)	(27.04)	(0.14)	(0.11)	(0.10)
R^2	0.173	0.022	0.202	0.341	0.137
Adjusted R^2	0.117	-0.046	0.148	0.243	0.010

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

each participant and task appear only once in the two conditions (with and without the LLM information support tool). For example, for the task completion time response, we estimate the model:

```
completion_time ~ programming_experience + task_experience + LLM_knowledge + uses_tool
```

The estimated coefficient for the tool usage variable indicates the effect of using the LLM information support tool while holding fixed the effects of programming experience, task experience, and LLM knowledge.

Results

Table 6.1 (1)-(3) columns display the regression results for three response variables. The task progress model (1) shows a significant difference between the two conditions, with participants in the LLM information support tool condition completing statistically significantly more sub-tasks than those who used search engines. Controlling for experience levels and AI tool familiarity, participants completed 0.47 more subtasks in the LLM information support tool condition compared to the control condition. This indicates that the LLM information support tool may assist users in making more progress in their tasks than 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 the LLM information support tool condition and the control group. This suggests that users in the LLM information support tool condition do not complete their tasks at a sufficiently different speed or have a suffi-

ciently different level of understanding than those in the control group, given the statistical power of our experiment.

From the qualitative analysis, participants also appreciated the within-context setting a lot. Participants (e.g., P4, P31) valued the ability to prompt 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 LLM with just code context, as it allowed them to bypass the need to write proficient queries, a well-known challenge in information retrieval. P15 mentioned "*It's nice not to need to know anything about the context before being effective in your search strategy.*"

6.2 Summary

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

Chapter 7

Generation-based Information Support Considering Developer's Characteristics

Building upon previous research, this thesis proposal aims to broaden the context scope, and develop an information support tool that aids developers working with unfamiliar libraries or domains, with individual developers' contexts taken into account. The proposed work consists of two main stages: adding a personalization feature to GILT (see Chapter 6), and conducting a user study with developers to measure the effectiveness of personalization. Among many personal factors that might affect developers' information seeking and program comprehension, we plan to focus on developers' expertise in programming and the application domain to personalize information support.

Personalization Implementation

To measure the users' experience levels in programming and the application domain, we plan to employ two methods: (1) manual inquiries using a questionnaire and (2) user repository mining. The questionnaire will seek information on the general levels of experience in programming languages used in the user's code (e.g., Python), the libraries utilized in their code (e.g., PyTorch), and their familiarity with the application domain (e.g., Machine Learning). Through repository mining, we will delve into more specific details, including whether the user has worked on projects within the same application domain, the relevant libraries employed in that domain (e.g., Tensorflow), and a comprehensive list of API methods they have utilized.

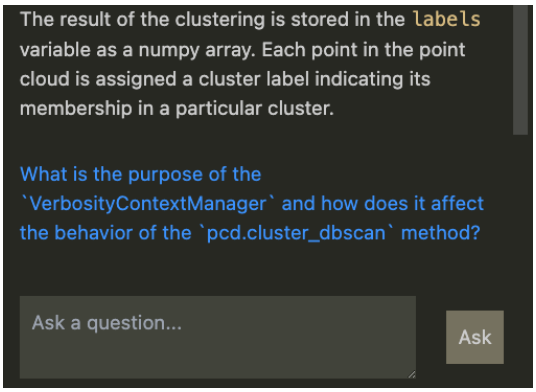

Figure 7.1: Prompt suggestion.

Next, prompt engineering will be employed to incorporate user characteristics into the information generation process, tailoring the content to individual users' experience levels. For example, when a user is new to library A, our prototype tool will internally add context by stating, "For a developer who is new to [library A]," before the user query. When available, the list of API methods the user has previously used, will be also used as context to generate information, by using a prompt like, "For a developer who knows [library A-method 1] and [library A-method 2], explain the details of [library B-method 1']" when a user seeks information about a method similar to one from another library they have previously used. As we will have users' experience levels on multiple dimensions, further exploration will be needed to effectively provide contextual information across these dimensions.

Another approach to offering personalized information support is by providing prompt suggestions, as illustrated in Figure 7.1. Most conventional information retrieval methods, including GILT, operate on a pull-based model, where users initiate information seeking by submitting search queries. While this approach helps mitigate information overload associated with push-based interactions, research [30] indicates that developers who lack familiarity with the application domain or programming environment often struggle to identify a starting point for their information search. This is primarily because they may not possess the necessary concepts and keywords to formulate effective search queries. To address the challenges of prompt formulation without overwhelming users with excessive information, we propose suggesting prompts tailored to developers' experience levels, but only provide responses to these prompts when developers initiate the system with such queries. In the process of generating these prompts, we will leverage LLMs to create a list of pertinent questions. For instance, we might use a prompt like "What are some key questions to ask when trying to comprehend the following code, when a developer is new to [library A]?"

Human Study Design

Once the prototype is complete, a human study will be conducted with developers to evaluate the overall effectiveness of personalization, as well as the benefits brought by each personalization dimension. The research aims to answer the following research questions:

- RQ7.1: How does personalization in the LLM-powered information support tool affect the code comprehension and the completion of tasks involving unfamiliar code?
- RQ7.2: Which user characteristic's personalization is most effective for efficient learning?
- RQ7.3: How do developers perceive the usefulness of personalized information generation?

Participants. We will advertise the study widely within the university community and to the public. During the recruitment process, we will ask about participants' programming

backgrounds and exclude individuals who indicate minimal experience. To understand participants' programming and domain expertise and ensure a balanced representation, we will pose questions regarding their proficiency in programming languages, experience in the specific domains relevant to our tasks, familiarity with libraries pertinent to our study, and, in the case of students, their degree programs. If participants possess expertise in the domains or libraries relevant to our tasks, we will also request that they provide links to their GitHub repositories. This will allow us to extract additional details regarding their experience, such as a list of previously used API methods.

Experimental Design. We will employ a within-subjects design, with participants using both GILT and GILT with personalization. This will allow us to ask participants to rate both conditions and provide comparative feedback about both treatments. We will prepare four tasks, that require a certain degree of understanding of domain-specific knowledge (e.g., security) in uncommon libraries. To identify the effectiveness of each personalization dimension (RQ7.2), each participant will engage in these tasks under four different conditions, and we will then compare the performance of the users under these conditions:

- Without considering the developer's context (GILT)
- Considering only the developer's domain knowledge (GILT + domain)
- Considering only the developer's experience with libraries (GILT + library)
- Considering both dimensions (GILT + domain + library)

Data Collection. To evaluate the effectiveness of considering developer characteristics in the information support tool, participants' performance will be quantitatively measured. This assessment will include factors such as completion time, number of queries made, task completeness, and correctness of quiz question responses.

Study Protocol. Prior to the study sessions, we will collect participants' experience levels, through questionnaires and mining their repositories. Similar to the previous study, we will conduct the study via a video conferencing tool and with the web-based VS Code IDE hosted on GitHub CodeSpaces. For personalization, we will pre-configure the plugin with each participant's profile to save time during the study setup. During the study, we will start each session with a demonstration task, illustrating how to use our plugin. Then, for each task, participants will be given an initial 5 minutes to comprehend the provided code for the tasks. This is intended to encourage participants to invest sufficient time in comprehending the code rather than solely relying on language models for task completion. Following this, we will assign tasks to participants that require code editing. Through this step, we aim to gauge the effectiveness of personalization in programming tasks by considering completion rates and completion time. Finally, at the end of the study, participants will be requested to fill out a post-study survey evaluating the perceived usefulness and ease of use of the personalization, and addressing any concerns they may have.

Chapter 8

Proposed Contributions

My thesis is expected to make a number of contributions to enhancing information support for developers' learning, including:

- Evidence that developers' different use of documentation depends on their user characteristics (e.g., experience level with the API).
- Evidence that push-based comparable API methods information support can enhance discoverability issues in learning new libraries.
- Evidence that context-aware generation-based information support using LLM can help developers complete more tasks.
- Three information support tool prototypes: one Chrome plug-in and two VS Code plug-ins.
- SOREL, a learning-based model extracting comparable API methods and the support evidence from Stack Overflow posts.
- A language model, which can predict a sequence of API methods given input and output value pairs.

Chapter 9

Proposed Timeline

	Tasks	Prior	2023					2024			
			Aug	Sep	Oct	Nov	Dec	Jan	Feb	Mar	Apr
Chapter 3	Paper Revision & Resubmission										
Chapter 4											
Chapter 5											
Chapter 6	Paper Revision & Resubmission										
Chapter 7	Prototype Development										
	Task Design										
	Pilot Study & Participant Recruitment										
	User Study										
	Analysis										
Dissertation	Dissertation Writing										

Figure 9.1: Proposed timeline. Green indicates completed tasks, orange indicates tasks that are in progress, and navy indicates tasks that are planned.

There is one final study left for this thesis: Generation-based Information Support Considering Developer's Characteristics (Chapter 7). The final study consists of four main parts: Prototype Development, Study Design, User Study, and Analysis. I plan to work on the first two parts simultaneously. Although they will require multiple iterations based on the pilot study feedback, I estimate that it will take about two months overall. Once I finish the study design, I estimate the participant recruitment and actual user study to take two months, and I will analyze the collected data by the January of 2024. This will give a reasonable buffer for writing, paper submissions or revisions, and presentation preparation culminating in a thesis defense in April 2024.

Bibliography

[1] 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] 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.

[3] Deepak Agarwal, Bee-Chung Chen, Pradheep Elango, and Raghu Ramakrishnan. Content recommendation on web portals. *Commun. ACM*, 56(6):92–101, 2013.

[4] Emad Aghajani, Csaba Nagy, Olga Lucero Vega-Márquez, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, and Michele Lanza. Software documentation issues unveiled. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1199–1210. IEEE, 2019.

[5] James D Arthur and K Todd Stevens. Document quality indicators: A framework for assessing documentation adequacy. *Journal of Software Maintenance: Research and Practice*, 4(3):129–142, 1992.

[6] 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.

[7] 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.

[8] 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 *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1589–1598, 2009.

[9] 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.

[10] 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.

[11] Jie-Cherng Chen and Sun-Jen Huang. An empirical analysis of the impact of software development problem factors on software maintainability. *Journal of Systems and Software*, 82(6):981–992, 2009.

[12] Steven Clarke. What is an end user software engineer? In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2007.

[13] 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.

[14] Andreas Dautovic. Automatic assessment of software documentation quality. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 665–669. IEEE, 2011.

[15] 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.

[16] 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.

[17] 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.

[18] Ralph H Earle, Mark A Rosso, and Kathryn E Alexander. User preferences of software documentation genres. In *Proceedings of the 33rd Annual International Conference on the Design of Communication*, pages 1–10, 2015.

[19] Steve Fox, Kuldeep Karnawat, Mark Mydland, Susan Dumais, and Thomas White. Evaluating implicit measures to improve web search. *ACM Transactions on Information Systems (TOIS)*, 23(2):147–168, 2005.

[20] 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.

[21] Luanne Freund. Contextualizing the information-seeking behavior of software engineers. *Journal of the Association for Information Science and Technology*, 66(8):1594–1605, 2015.

[22] Klaus Hinkelmann and Oscar Kempthorne. *Design and analysis of experiments, volume 1: Introduction to experimental design*, volume 1. John Wiley & Sons, 2007.

[23] 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.

[24] Jeremy Howard and Sebastian Ruder. Universal language model fine-tuning for text classification. *arXiv preprint arXiv:1801.06146*, 2018.

[25] 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.

[26] 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.

[27] Andrew J. Ko, Robert DeLine, and Gina Venolia. Information Needs in Collocated Software Development Teams. *29th International Conference on Software Engineering (ICSE'07)*, pages 1–10, 2007.

[28] Andrew J Ko, Robert DeLine, and Gina Venolia. Information needs in collocated software development teams. In *29th International Conference on Software Engineering (ICSE'07)*, pages 344–353. IEEE, 2007.

[29] Andrew 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 Transactions on software engineering*, 32(12):971–987, 2006.

[30] Andrew J Ko and Yann Riche. The role of conceptual knowledge in api usability. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 173–176. IEEE, 2011.

[31] Andrew Jensen 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 Transactions on Software Engineering*, 32(12):971 – 987, 11 2006.

[32] J Richard Landis and Gary G Koch. The measurement of observer agreement for categorical data. *biometrics*, pages 159–174, 1977.

[33] 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.

[34] 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.

[35] Thomas D LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th international conference on Software engineering*, pages 492–501, 2006.

[36] Joseph Lawrance, Christopher Bogart, Margaret Burnett, Rachel Bellamy, Kyle Rector, and Scott D Fleming. How programmers debug, revisited: An information foraging theory perspective. *IEEE Transactions on Software Engineering*, 39(2):197–215, 2010.

[37] 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.

[38] Hongwei Li, Zhenchang Xing, Xin Peng, and Wenyun Zhao. What help do developers seek, when and how? In *2013 20th working conference on reverse engineering (WCRE)*, pages 142–151. IEEE, 2013.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] Bing Liu. *Web data mining: exploring hyperlinks, contents, and usage data*. Springer Science & Business Media, 2007.

[44] 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, page 67–80, New York, NY, USA, 2019. Association for Computing Machinery.

[45] 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.

[46] 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.

[47] Walid Maalej and Martin P Robillard. Patterns of knowledge in api reference documentation. *IEEE Transactions on Software Engineering*, 39(9):1264–1282, 2013.

[48] Walid Maalej, Rebecca Tiarks, Tobias Roehm, and Rainer Koschke. On the Comprehension of Program Comprehension. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(4):31 – 37, 09 2014.

[49] Walid Maalej, Rebecca Tiarks, Tobias Roehm, and Rainer Koschke. On the comprehension of program comprehension. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(4):1–37, 2014.

[50] 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, 07 2017.

[51] 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.

[52] Michael Meng, Stephanie Steinhardt, and Andreas Schubert. How developers use api documentation: an observation study. *Communication Design Quarterly Review*, 7(2):40–49, 2019.

[53] 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.

[54] 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.

[55] 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.

[56] Nico JD Nagelkerke et al. A note on a general definition of the coefficient of determination. *Biometrika*, 78(3):691–692, 1991.

[57] Daye Nam, Andrew Macvean, Brad Myers, and Bogdan Vasilescu. Understanding documentation use through log analysis: A case study of four cloud services. Under submission.

[58] Daye Nam, Brad A. Myers, Bogdan Vasilescu, and Vincent 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*, 2023.

[59] 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.

[60] Daye Nam, Baishakhi Ray, Seohyun Kim, Xianshan Qu, and Satish Chandra. Predictive synthesis of api-centric code. *CoRR*, abs/2201.03758, 2022.

[61] 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. In *Proceedings of the 20th annual international conference on Computer documentation*, pages 133–141, 2002.

[62] 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.

[63] 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. the 2016 24th ACM SIGSOFT International Symposium, pages 97 – 108, 2016.

[64] 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.

[65] 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.

[66] 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(1):36–44, 2004.

[67] Nikitha Rao, Chetan Bansal, Thomas Zimmermann, Ahmed Hassan Awadallah, and Nachiappan Nagappan. Analyzing web search behavior for software engineering tasks. In *2020 IEEE International Conference on Big Data (Big Data)*, pages 768–777. IEEE, 2020.

[68] 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.

[69] 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.

[70] Martin P Robillard. What makes apis hard to learn? answers from developers. *IEEE software*, 26(6):27–34, 2009.

[71] 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.

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

[73] 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.

[74] Kensen Shi, David Bieber, and Rishabh Singh. Tf-coder: Program synthesis for tensor manipulations. *CoRR*, abs/2003.09040, 2020.

[75] 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.

[76] 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.

[77] 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.

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

[79] Siddharth Subramanian, Laura Inozemtseva, and Reid Holmes. Live api documentation. In *International Conference on Software Engineering (ICSE)*, pages 643–652, 2014.

[80] 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.

[81] 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.

[82] Kyle Thayer, Sarah E Chasins, and Amy J Ko. A theory of robust api knowledge. *ACM Transactions on Computing Education (TOCE)*, 21(1):1–32, 2021.

[83] 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.

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

[85] 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.

[86] 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.

[87] Gias Uddin, Foutse Khomh, and Chanchal K Roy. Mining api usage scenarios from stack overflow. *Information and Software Technology*, 122:106277, 2020.

[88] 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.

[89] Gias Uddin and Martin P Robillard. How api documentation fails. *IEEE Software*, 32(4):68–75, 2015.

[90] Marcello Visconti and Curtis R Cook. Assessing the state of software documentation practices. In *International Conference on Product Focused Software Process Improvement*, pages 485–496. Springer, 2004.

[91] 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.

[92] Han Wang, Chunyang Chen, Zhenchang Xing, and John Grundy. Difftech: Differencing similar technologies from crowd-scale comparison discussions. *IEEE Transactions on Software Engineering*, 2021.

[93] Frank F Xu, Uri Alon, Graham Neubig, and Vincent J Hellendoorn. A Systematic Evaluation of Large Language Models of Code. *arXiv*, 2022.

[94] 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.

[95] 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, page 113–120, New York, NY, USA, 2014. Association for Computing Machinery.

[96] 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.

[97] 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.

[98] 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.

[99] 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 *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, pages 498–509, 2016.